TFSnippet Documentation

Release 0.2.0a4

Haowen Xu

Sep 10, 2019

Contents

TFSnippet is a set of utilities for writing and testing TensorFlow models.

The design philosophy of TFSnippet is *non-interfering*. It aims to provide a set of useful utilities, possible to be used along with any other TensorFlow libraries and frameworks.

CHAPTER 1

Installation

```
pip install git+https://github.com/thu-ml/zhusuan.git
pip install git+https://github.com/haowen-xu/tfsnippet.git
```

CHAPTER 2

Documentation

2.1 API Docs

2.1.1 tfsnippet

tfsnippet Package

Functions

as_distribution(distribution)	Convert a supported type of *distribution* into *Distribution* type.
reduce_group_ndims(operation, tensor, ..., [, ...])	Reduce the last *group_ndims* dimensions in *tensor*, using *operation*.
summarize_variables(variables[, title, ...])	Get a formatted summary about the variables.
auto_batch_weight(*batch_arrays)	Automatically inspect the metric weight for an evaluation mini-batch.
merge_feed_dict(*feed_dicts)	Merge all feed dicts into one.
resolve_feed_dict(feed_dict[, inplace])	Resolve all dynamic values in *feed_dict* into fixed values.
elbo_objective(log_joint, latent_log_prob[, ...])	Derive the ELBO objective.
importance_sampling_log_likelihood(..., [, ...])	Compute $\log p(\mathbf{x})$ by importance sampling.
iwae_estimator(log_values, axis[, keepdims, ...])	Derive the gradient estimator for $\mathbb{E}_{q(\mathbf{z}^{(1:K)}\|\mathbf{x})}\left[\log \frac{1}{K}\sum_{k=1}^{K} f\left(\mathbf{x}, \mathbf{z}^{(k)}\right)\right]$, by IWAE (Burda, Y., Grosse, R.
monte_carlo_objective(log_joint, latent_log_prob)	Derive the Monte-Carlo objective.
nvil_estimator(values, latent_log_joint[, ...])	Derive the gradient estimator for $\mathbb{E}_{q(\mathbf{z}\|\mathbf{x})}\left[f(\mathbf{x}, \mathbf{z})\right]$, by NVIL (Mnih and Gregor, 2014) algorithm.

Continued on next page

Table 1 – continued from previous page

sgvb_estimator(values[, axis, keepdims, name])	Derive the gradient estimator for $\mathbb{E}_{q(\mathbf{z}\|\mathbf{x})}\big[f(\mathbf{x},\mathbf{z})\big]$, by SGVB (Kingma, D.P.
vimco_estimator(log_values, latent_log_joint)	Derive the gradient estimator for
get_config_defaults(config)	Get the default config values of *config*.
register_config_arguments(config, parser[, ...])	Register config to the specified argument parser.
model_variable(name[, shape, dtype, ...])	Get or create a model variable.
get_model_variables([scope])	Get all model variables (i.e., variables in *MODEL_VARIABLES* collection).
instance_reuse([method_or_scope, _sentinel, ...])	Decorate an instance method to reuse a variable scope automatically.
global_reuse([method_or_scope, _sentinel, scope])	Decorate a function to reuse a variable scope automatically.
add_histogram(tensor[, summary_name, ...])	Add the histogram of *tensor* to the default summary collector, and to *collections*.
add_summary(summary[, collections])	Add the summary to the default summary collector, and to *collections*.
default_summary_collector()	Get the *SummaryCollector* object at the top of context stack.

as_distribution

tfsnippet.**as_distribution**(*distribution*)

Convert a supported type of *distribution* into *Distribution* type.

Parameters **distribution** – A supported distribution instance. Supported types are: 1. *Distribution*, 2. zhusuan.distributions.Distribution.

Returns The wrapped distribution of *Distribution* type.

Return type *Distribution*

Raises TypeError – If the specified *distribution* cannot be converted.

reduce_group_ndims

tfsnippet.**reduce_group_ndims**(*operation*, *tensor*, *group_ndims*, *name=None*)

Reduce the last *group_ndims* dimensions in *tensor*, using *operation*.

In Distribution, when computing the (log-)densities of certain *tensor*, the last few dimensions may represent a group of events, thus should be accounted together. This method can be used to reduce these dimensions, for example:

```
log_prob = reduce_group_ndims(tf.reduce_sum, log_prob, group_ndims)
prob = reduce_group_ndims(tf.reduce_prod, log_prob, group_ndims)
```

Parameters

- **operation** – The operation for reducing the last *group_ndims* dimensions. It must receive *tensor* as the 1st argument, and *axis* as the 2nd argument.
- **tensor** – The tensor to be reduced.
- **group_ndims** – The number of dimensions at the end of *tensor* to be reduced. If it is a constant integer and is zero, then no operation will take place.

- **name** – TensorFlow name scope of the graph nodes. (default "reduce_group_ndims")

Returns The reduced tensor.

Return type tf.Tensor

Raises `ValueError` – If *group_ndims* cannot be validated by `validate_group_ndims()`.

summarize_variables

`tfsnippet.`**`summarize_variables`**(*variables*, *title='Variables Summary'*, *other_variables_title='Other Variables'*, *groups=None*, *sort_by_names=False*)

Get a formatted summary about the variables.

Parameters

- **variables** (*list[tf.Variable] or dict[str, tf.Variable]*) – List or dict of variables to be summarized.
- **title** (*str*) – Title of this summary.
- **other_variables_title** (*str*) – Title of the "Other Variables".
- **groups** (*None or list[str]*) – List of separated variable groups, each summarized in a table. (default `None`)
- **sort_by_names** (*bool*) – Whether or not to sort the variables within each group by their names? (if not `True`, will display the variables according to their natural order)

Returns Formatted summary about the variables.

Return type str

auto_batch_weight

`tfsnippet.`**`auto_batch_weight`**(**batch_arrays*)

Automatically inspect the metric weight for an evaluation mini-batch.

Parameters ***batch_arrays** – Mini-batch arrays. The `.size` of the first array will be used as the weight.

Returns The inspected weight, or 1. if any error occurs during inspection.

merge_feed_dict

`tfsnippet.`**`merge_feed_dict`**(**feed_dicts*)

Merge all feed dicts into one.

Parameters ****feed_dicts** – List of feed dicts. The later ones will override values specified in the previous ones. If a `None` is specified, it will be simply ignored.

Returns The merged feed dict.

resolve_feed_dict

tfsnippet.**resolve_feed_dict**(*feed_dict*, *inplace=False*)

Resolve all dynamic values in *feed_dict* into fixed values.

The supported dynamic value types and corresponding resolving method is listed as follows:

1. *ScheduledVariable*: `get()` will be called.
2. *DynamicValue*: `get()` will be called.
3. callable object: Will be called to get the value.

Parameters

- **feed_dict** (*dict[tf.Tensor, any]*) – The feed dict to be resolved.
- **inplace** (*bool*) – Whether or not to fill resolved values in the input *feed_dict* directly, instead of copying a new one? (default `False`)

Returns The resolved feed dict.

elbo_objective

tfsnippet.**elbo_objective**(*log_joint*, *latent_log_prob*, *axis=None*, *keepdims=False*, *name=None*)

Derive the ELBO objective.

$$\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\big[\log p_\theta(\mathbf{x},\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\big]$$

Parameters

- **log_joint** – Values of $\log p(\mathbf{z},\mathbf{x})$, computed with $\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})$.
- **latent_log_prob** – $q(\mathbf{z}|\mathbf{x})$.
- **axis** – The sampling dimensions to be averaged out. If `None`, no dimensions will be averaged out.
- **keepdims** (*bool*) – When *axis* is specified, whether or not to keep the averaged dimensions? (default `False`)
- **name** (*str*) – TensorFlow name scope of the graph nodes. (default "elbo_objective")

Returns The ELBO objective. Not applicable for training.

Return type tf.Tensor

importance_sampling_log_likelihood

tfsnippet.**importance_sampling_log_likelihood**(*log_joint*, *latent_log_prob*, *axis*, *keepdims=False*, *name=None*)

Compute $\log p(\mathbf{x})$ by importance sampling.

$$\log p(\mathbf{x}) = \log \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\Big[\exp\big(\log p(\mathbf{x},\mathbf{z}) - \log q(\mathbf{z}|\mathbf{x})\big)\Big]$$

Parameters

- **log_joint** – Values of $\log p(\mathbf{z},\mathbf{x})$, computed with $\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})$.
- **latent_log_prob** – $q(\mathbf{z}|\mathbf{x})$.

- **axis** – The sampling dimensions to be averaged out.
- **keepdims** (*bool*) – When *axis* is specified, whether or not to keep the averaged dimensions? (default `False`)
- **name** (*str*) – TensorFlow name scope of the graph nodes. (default "importance_sampling_log_likelihood")

Returns The computed $\log p(x)$.

iwae_estimator

`tfsnippet.`**`iwae_estimator`**(*log_values*, *axis*, *keepdims=False*, *name=None*)

Derive the gradient estimator for $\mathbb{E}_{q(\mathbf{z}^{(1:K)}|\mathbf{x})}\left[\log \frac{1}{K}\sum_{k=1}^{K} f\left(\mathbf{x}, \mathbf{z}^{(k)}\right)\right]$, by IWAE (Burda, Y., Grosse, R. and Salakhutdinov, R., 2015) algorithm.

$$\nabla \mathbb{E}_{q(\mathbf{z}^{(1:K)}|\mathbf{x})}\left[\log \frac{1}{K}\sum_{k=1}^{K} f\left(\mathbf{x}, \mathbf{z}^{(k)}\right)\right] = \nabla \mathbb{E}_{q(\epsilon^{(1:K)})}\left[\log \frac{1}{K}\sum_{k=1}^{K} w_k\right] = \mathbb{E}_{q(\epsilon^{(1:K)})}\left[\nabla \log \frac{1}{K}\sum_{k=1}^{K} w_k\right] =$$
$$\mathbb{E}_{q(\epsilon^{(1:K)})}\left[\frac{\nabla \frac{1}{K}\sum_{k=1}^{K} w_k}{\frac{1}{K}\sum_{i=1}^{K} w_i}\right] = \mathbb{E}_{q(\epsilon^{(1:K)})}\left[\frac{\sum_{k=1}^{K} w_k \nabla \log w_k}{\sum_{i=1}^{K} w_i}\right] = \mathbb{E}_{q(\epsilon^{(1:K)})}\left[\sum_{k=1}^{K} \widetilde{w}_k \nabla \log w_k\right]$$

Parameters

- **log_values** – Log values of the target function given z and x, i.e., $\log f(\mathbf{z}, \mathbf{x})$.
- **axis** – The sampling axes to be reduced in outputs.
- **keepdims** (*bool*) – When *axis* is specified, whether or not to keep the reduced axes? (default `False`)
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

The surrogate for optimizing the original target. Maximizing/minimizing this surrogate via gradient descent will effectively maximize/minimize the original target.

Return type tf.Tensor

monte_carlo_objective

`tfsnippet.`**`monte_carlo_objective`**(*log_joint*, *latent_log_prob*, *axis=None*, *keepdims=False*, *name=None*)

Derive the Monte-Carlo objective.

$$\mathcal{L}_K(\mathbf{x}; \theta, \phi) = \mathbb{E}_{\mathbf{z}^{(1:K)} \sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{1}{K}\sum_{k=1}^{K} \frac{p_\theta(\mathbf{x}, \mathbf{z}^{(k)})}{q_\phi(\mathbf{z}^{(k)}|\mathbf{x})}\right]$$

Parameters

- **log_joint** – Values of $\log p(\mathbf{z}, \mathbf{x})$, computed with $\mathbf{z} \sim q(\mathbf{z}|\mathbf{x})$.
- **latent_log_prob** – $q(\mathbf{z}|\mathbf{x})$.
- **axis** – The sampling dimensions to be averaged out.

- **keepdims** (*bool*) – When *axis* is specified, whether or not to keep the averaged dimensions? (default `False`)
- **name** (*str*) – TensorFlow name scope of the graph nodes. (default "monte_carlo_objective")

Returns The Monte Carlo objective. Not applicable for training.

Return type tf.Tensor

nvil_estimator

tfsnippet.**nvil_estimator**(*values*, *latent_log_joint*, *baseline=None*, *center_by_moving_average=True*, *decay=0.8*, *axis=None*, *keepdims=False*, *batch_axis=None*, *name=None*)

Derive the gradient estimator for $\mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\big[f(\mathbf{x},\mathbf{z})\big]$, by NVIL (Mnih and Gregor, 2014) algorithm.

$$\begin{aligned}\nabla \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\big[f(\mathbf{x},\mathbf{z})\big] &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\Big[\nabla f(\mathbf{x},\mathbf{z}) + f(\mathbf{x},\mathbf{z})\,\nabla \log q(\mathbf{z}|\mathbf{x})\Big] \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\Big[\nabla f(\mathbf{x},\mathbf{z}) + \big(f(\mathbf{x},\mathbf{z}) - C_\psi(\mathbf{x}) - c\big)\,\nabla \log q(\mathbf{z}|\mathbf{x})\Big]\end{aligned}$$

where $C_\psi(\mathbf{x})$ is a learnable network with parameter ψ, and c is a learnable constant. They would be learnt by minimizing $\mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\Big[\big(f(\mathbf{x},\mathbf{z}) - C_\psi(\mathbf{x}) - c\big)^2\Big]$.

Parameters

- **values** – Values of the target function given z and x, i.e., $f(\mathbf{z},\mathbf{x})$.
- **latent_log_joint** – Values of $\log q(\mathbf{z}|\mathbf{x})$.
- **baseline** – Values of the baseline function $C_\psi(\mathbf{x})$ given input x. If this is not specified, then this method will degenerate to the REINFORCE algorithm, with only a moving average estimated constant baseline c.
- **center_by_moving_average** (*bool*) – Whether or not to use the moving average to maintain an estimation of c in above equations?
- **decay** – The decaying factor for moving average.
- **axis** – The sampling axes to be reduced in outputs. If not specified, no axis will be reduced.
- **keepdims** (*bool*) – When *axis* is specified, whether or not to keep the reduced axes? (default `False`)
- **batch_axis** – The batch axes to be reduced when computing expectation over x. If not specified, all axes will be treated as batch axes, except the sampling axes.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

The *(surrogate, baseline cost)*.

surrogate is the surrogate for optimizing the original target. Maximizing/minimizing this surrogate via gradient descent will effectively maximize/minimize the original target.

baseline cost is the cost to be minimized for training baseline. It will be `None` if *baseline* is `None`.

Return type (tf.Tensor, tf.Tensor)

sgvb_estimator

tfsnippet.**sgvb_estimator**(*values*, *axis=None*, *keepdims=False*, *name=None*)

Derive the gradient estimator for $\mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\big[f(\mathbf{x},\mathbf{z})\big]$, by SGVB (Kingma, D.P. and Welling, M., 2013) algorithm.

$$\nabla \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\big[f(\mathbf{x},\mathbf{z})\big] = \nabla \mathbb{E}_{q(\epsilon)}\big[f(\mathbf{x},\mathbf{z}(\epsilon))\big] = \mathbb{E}_{q(\epsilon)}\big[\nabla f(\mathbf{x},\mathbf{z}(\epsilon))\big]$$

Parameters

- **values** – Values of the target function given z and x, i.e., $f(\mathbf{z},\mathbf{x})$.
- **axis** – The sampling axes to be reduced in outputs. If not specified, no axis will be reduced.
- **keepdims** (*bool*) – When *axis* is specified, whether or not to keep the reduced axes? (default False)
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

The surrogate for optimizing the original target. Maximizing/minimizing this surrogate via gradient descent will effectively maximize/minimize the original target.

Return type tf.Tensor

vimco_estimator

tfsnippet.**vimco_estimator**(*log_values*, *latent_log_joint*, *axis=None*, *keepdims=False*, *name=None*)

Derive the gradient estimator for $\mathbb{E}_{q(\mathbf{z}^{(1:K)}|\mathbf{x})}\Big[\log \frac{1}{K}\sum_{k=1}^{K} f\big(\mathbf{x},\mathbf{z}^{(k)}\big)\Big]$, by VIMCO (Minh and Rezende, 2016) algorithm.

&nabla,mathbb{E}_{q(mathbf{z}^{(1:K)}|mathbf{x})}Big[log frac{1}{K} sum_{k=1}^K fbig(mathbf{x},mathbf{z}^{

&quad = mathbb{E}_{q(mathbf{z}^{(1:K)}|mathbf{x})}bigg[{sum_{k=1}^K hat{L}(mathbf{z}^{(k)}|mathbf{z} mathbb{E}_{q(mathbf{z}^{(1:K)}|mathbf{x})}bigg[{sum_{k=1}^K wide-tilde{w}_k,nablalog f(mathbf{x},mathbf{z}^{(k)})}bigg]

end{aligned}

where $w_k = f\big(\mathbf{x},\mathbf{z}^{(k)}\big)$, $\widetilde{w}_k = w_k / \sum_{i=1}^{K} w_i$, and:

$$\hat{L}(\mathbf{z}^{(k)}|\mathbf{z}^{(-k)}) = \hat{L}(\mathbf{z}^{(1:K)}) - \log\frac{1}{K}\left(\hat{f}(\mathbf{x},\mathbf{z}^{(-k)}) + \sum_{i\neq k} f(\mathbf{x},\mathbf{z}^{(i)})\right)$$

$$\hat{L}(\mathbf{z}^{(1:K)}) = \log\frac{1}{K}\sum_{k=1}^{K} f(\mathbf{x},\mathbf{z}^{(k)})$$

$$\hat{f}(\mathbf{x},\mathbf{z}^{(-k)}) = \exp\big(\frac{1}{K-1}\sum_{i\neq k}\log f(\mathbf{x},\mathbf{z}^{(i)})\big)$$

Args:

log_values: Log values of the target function given z and x, i.e., $\log f(\mathbf{z},\mathbf{x})$.

latent_log_joint: Values of $\log q(\mathbf{z}|\mathbf{x})$. axis: The sampling axes to be reduced in outputs. keepdims (bool): When *axis* is specified, whether or not to keep

the reduced axes? (default `False`)

name (str): Default name of the name scope. If not specified, generate one according to the method name.

Returns:

tf.Tensor: The surrogate for optimizing the original target. Maximizing/minimizing this surrogate via gradient descent will effectively maximize/minimize the original target.

get_config_defaults

tfsnippet.**get_config_defaults**(*config*)

Get the default config values of *config*.

Parameters **config** – An instance of *Config*, or a class which is a subclass of *Config*.

Returns The default config values of *config*.

Return type dict[str, any]

register_config_arguments

tfsnippet.**register_config_arguments**(*config*, *parser*, *prefix=None*, *title=None*, *description=None*, *sort_keys=False*)

Register config to the specified argument parser.

Usage:

```
class YourConfig(Config):
    max_epoch = 1000
    learning_rate = 0.01
    activation = ConfigField(
        str, default='leaky_relu', choices=['relu', 'leaky_relu'])

# First, you should obtain an instance of your config object
config = YourConfig()

# You can then parse config values from CLI arguments.
# For example, if sys.argv[1:] == ['--max_epoch=2000']:
from argparse import ArgumentParser
parser = ArgumentParser()
spt.register_config_arguments(config, parser)
parser.parse_args(sys.argv[1:])

# Now you can access the config value `config.max_epoch == 2000`
print(config.max_epoch)
```

Parameters

- **config** (*Config*) – The config object.
- **parser** (*ArgumentParser*) – The argument parser.

- **prefix** (*str*) – Optional prefix of the config keys. *new_config_key* = *prefix* + '.' + *old_config_key*
- **title** (*str*) – If specified, will create an argument group to collect all the config arguments.
- **description** (*str*) – The description of the argument group.
- **sort_keys** (*bool*) – Whether or not to sort the config keys before registering to the parser? (default `False`)

model_variable

tfsnippet.**model_variable**(*name*, *shape=None*, *dtype=None*, *initializer=None*, *regularizer=None*, *constraint=None*, *trainable=True*, *collections=None*, ***kwargs*)

Get or create a model variable.

When the variable is created, it will be added to both *GLOBAL_VARIABLES* and *MODEL_VARIABLES* collection.

Parameters

- **name** – Name of the variable.
- **shape** – Shape of the variable.
- **dtype** – Data type of the variable.
- **initializer** – Initializer of the variable.
- **regularizer** – Regularizer of the variable.
- **constraint** – Constraint of the variable.
- **trainable** (*bool*) – Whether or not the variable is trainable?
- **collections** – In addition to *GLOBAL_VARIABLES* and *MODEL_VARIABLES*, also add the variable to these collections.
- ****kwargs** – Other named arguments passed to `tf.get_variable()`.

Returns The variable.

Return type tf.Variable

get_model_variables

tfsnippet.**get_model_variables**(*scope=None*)

Get all model variables (i.e., variables in *MODEL_VARIABLES* collection).

Parameters **scope** – If specified, will obtain variables only within this scope.

Returns The model variables.

Return type list[tf.Variable]

instance_reuse

tfsnippet.**instance_reuse**(*method_or_scope=None*, *_sentinel=None*, *scope=None*)

Decorate an instance method to reuse a variable scope automatically.

This decorator should be applied to unbound instance methods, and the instance that owns the methods should have `variable_scope` attribute. The first time to enter a decorated method will open a new variable scope under the *variable_scope* of the instance. This variable scope will be reused the next time to enter this method. For example:

```
class Foo(object):

    def __init__(self, name):
        with tf.variable_scope(name) as vs:
            self.variable_scope = vs

    @instance_reuse
    def bar(self):
        return tf.get_variable('bar', ...)

foo = Foo()
bar = foo.bar()
bar_2 = foo.bar()
assert(bar is bar_2)  # should be True
```

By default the name of the variable scope should be chosen according to the name of the decorated method. You can change this behavior by specifying an alternative name, for example:

```
class Foo(object):

    @instance_reuse('scope_name')
    def foo(self):
        # name will be self.variable_scope.name + '/foo/bar'
        return tf.get_variable('bar', ...)
```

Unlike the behavior of *global_reuse()*, if you have two methods sharing the same scope name, they will indeed use the same variable scope. For example:

```
class Foo(object):

    @instance_reuse('foo')
    def foo_1(self):
        return tf.get_variable('bar', ...)

    @instance_reuse('foo')
    def foo_2(self):
        return tf.get_variable('bar', ...)


    @instance_reuse('foo')
    def foo_2(self):
        return tf.get_variable('bar2', ...)

foo = Foo()
foo.foo_1()  # its name should be variable_scope.name + '/foo/bar'
foo.foo_2()  # should raise an error, because 'bar' variable has
             # been created, but the decorator of `foo_2` does not
             # aware of this, so has not set ``reused = True``.
foo.foo_3()  # its name should be variable_scope.name + '/foo/bar2'
```

The reason to take this behavior is because the TensorFlow seems to have some absurd behavior when using `tf.variable_scope(..., default_name=?)` to uniquify the variable scope name. In some cases we the following absurd behavior would appear:

```
@global_reuse
def foo():
    with tf.variable_scope(None, default_name='bar') as vs:
        return vs

vs1 = foo()  # vs.name == 'foo/bar'
vs2 = foo()  # still expected to be 'foo/bar', but sometimes would be
             # 'foo/bar_1'. this absurd behavior is related to the
             # entering and exiting of variable scopes, which is very
             # hard to diagnose.
```

In order to compensate such behavior, if you have specified the `scope` argument of a *VarScopeObject*, then it will always take the desired variable scope. Also, constructing a *VarScopeObject* within a method or a function decorated by *global_reuse* or *instance_reuse* has been totally disallowed.

See also:

tfsnippet.utils.VarScopeObject, *tfsnippet.utils.global_reuse()*

global_reuse

tfsnippet.**global_reuse**(*method_or_scope=None*, *_sentinel=None*, *scope=None*)

Decorate a function to reuse a variable scope automatically.

The first time to enter a function decorated by this utility will open a new variable scope under the root variable scope. This variable scope will be reused the next time to enter this function. For example:

```
@global_reuse
def foo():
    return tf.get_variable('bar', ...)

bar = foo()
bar_2 = foo()
assert(bar is bar_2)  # should be True
```

By default the name of the variable scope should be chosen according to the name of the decorated method. You can change this behavior by specifying an alternative name, for example:

```
@global_reuse('dense')
def dense_layer(inputs):
    w = tf.get_variable('w', ...)  # name will be 'dense/w'
    b = tf.get_variable('b', ...)  # name will be 'dense/b'
    return tf.matmul(w, inputs) + b
```

If you have two functions sharing the same scope name, they will not use the same variable scope. Instead, one of these two functions will have its scope name added with a suffix '_?', for example:

```
@global_reuse('foo')
def foo_1():
    return tf.get_variable('bar', ...)

@global_reuse('foo')
def foo_2():
    return tf.get_variable('bar', ...)

assert(foo_1().name == 'foo/bar')
assert(foo_2().name == 'foo_1/bar')
```

The variable scope name will depend on the calling order of these two functions, so you should better not guess the scope name by yourself.

Note: If you use Keras, you SHOULD NOT create a Keras layer inside a *global_reuse* decorated function. Instead, you should create it outside the function, and pass it into the function.

See also:

tfsnippet.utils.instance_reuse()

add_histogram

tfsnippet.**add_histogram**(*tensor*, *summary_name=None*, *strip_scope=False*, *collections=None*, *name=None*)

Add the histogram of *tensor* to the default summary collector, and to *collections*.

Parameters

- **tensor** – Take histogram of this tensor.
- **summary_name** – Specify the summary name for *tensor*.
- **strip_scope** – If `True`, strip the name scope from *tensor.name* when adding the histogram.
- **collections** – Also add the histogram to these collections. Defaults to *self.collections*.

Returns The serialized histogram tensor of *tensor*.

add_summary

tfsnippet.**add_summary**(*summary*, *collections=None*)

Add the summary to the default summary collector, and to *collections*.

Parameters

- **summary** – TensorFlow summary tensor.
- **collections** – Also add the summary to these collections. Defaults to *self.collections*.

Returns The *summary* tensor.

default_summary_collector

tfsnippet.**default_summary_collector**()

Get the *SummaryCollector* object at the top of context stack.

Returns The summary collector.

Return type *SummaryCollector*

Classes

BatchToValueDistribution(distribution, ndims)	Distribution that converts the last few *batch_ndims* into *values_ndims*.
Bernoulli(logits[, dtype])	Univariate Bernoulli distribution.
Categorical(logits[, dtype])	Univariate Categorical distribution.
Concrete(temperature, logits[, ...])	The class of Concrete (or Gumbel-Softmax) distribution from (Maddison, 2016; Jang, 2016), served as the continuous relaxation of the *OnehotCategorical*.
Discrete	alias of `tfsnippet.distributions.univariate.Categorical`
DiscretizedLogistic(mean, log_scale, bin_size)	Discretized logistic distribution (Kingma et.
Distribution(dtype, is_continuous, ...)	Base class for probability distributions.
ExpConcrete(temperature, logits[, ...])	The class of ExpConcrete distribution from (Maddison, 2016), transformed from *Concrete* by taking logarithm.
FlowDistribution(distribution, flow)	Transform a *Distribution* by a `BaseFlow`, as a new distribution.
FlowDistributionDerivedTensor(tensor, ...)	A combination of a *FlowDistribution* derived tensor, and its original stochastic tensor from the base distribution.
Mixture(categorical, components[, ...])	Mixture distribution.
Normal(mean[, std, logstd, ...])	Univariate Normal distribution.
OnehotCategorical(logits[, dtype])	One-hot multivariate Categorical distribution.
Uniform([minval, maxval, ...])	Univariate Uniform distribution.
AnnealingVariable(name, initial_value, ratio)	A non-trainable `tf.Variable`, whose value will be annealed as training goes by.
CheckpointSavableObject	Base class for all objects that can be saved via *CheckpointSaver*.
CheckpointSaver(variables, save_dir[, ...])	Save and restore `tf.Variable`, *ScheduledVariable* and *CheckpointSavableObject* with `tf.train.Saver`.
DefaultMetricFormatter	Default training metric formatter.
EventKeys	Defines event keys for TFSnippet.
MetricFormatter	Base class for a training metrics formatter.
MetricLogger([summary_writer, ...])	Logger for the training metrics.
ScheduledVariable(name, initial_value[, ...])	A non-trainable `tf.Variable`, whose value might need to be changed as training goes by.
TrainLoop(param_vars[, var_groups, ...])	Training loop object.
AnnealingScalar(loop, initial_value, ratio)	A *DynamicValue* scalar, which anneals every few epochs or steps.
BaseTrainer(loop[, ensure_variables_initialized])	Base class for all trainers.
DynamicValue	Dynamic values to be fed into trainers and evaluators.
Evaluator(loop, metrics, inputs, data_flow)	Class to compute evaluation metrics.
LossTrainer(**kwargs)	A subclass of *BaseTrainer*, which optimizes a single loss.
Trainer(loop, train_op, inputs, data_flow[, ...])	A subclass of *BaseTrainer*, executing a training operation per step.
Validator(**kwargs)	Class to compute validation loss and other metrics.
VariationalChain(variational, model[, ...])	Chain of the variational and model nets for variational inference.

Continued on next page

Table 2 – continued from previous page

VariationalEvaluation(vi)	Factory for variational evaluation outputs.
VariationalInference(log_joint, latent_log_probs)	Class for variational inference.
VariationalLowerBounds(vi)	Factory for variational lower-bounds.
VariationalTrainingObjectives(vi)	Factory for variational training objectives.
BayesianNet([observed])	Bayesian networks.
DataFlow	Data flows are objects for constructing mini-batch iterators.
DataMapper	Base class for all data mappers.
SlidingWindow(data_array, window_size)	*DataMapper* for producing sliding windows according to indices.
Config	Base class for defining config values.
ConfigField(type[, default, description, ...])	A config field.
GraphKeys	Defines TensorFlow graph collection keys for TFSnippet.
InvertibleMatrix(size[, strict, dtype, ...])	A matrix initialized to be an invertible, orthogonal matrix.
VarScopeObject([name, scope])	Base class for objects that own a variable scope.
SummaryCollector([collections, ...])	Collecting summaries and histograms added by *tfsnippet.add_summary()* and *tfsnippet.add_histogram()*.
StochasticTensor(distribution, tensor[, ...])	Samples or observations of a stochastic variable.

BatchToValueDistribution

class tfsnippet.**BatchToValueDistribution**(*distribution*, *ndims*)

Bases: tfsnippet.distributions.base.Distribution

Distribution that converts the last few *batch_ndims* into *values_ndims*. See *Distribution.batch_ndims_to_value()* for more details.

Attributes Summary

base_distribution	Get the base distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, group_ndims, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *`sample()`*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *`Distribution`*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *group_ndims=0*, *is_reparameterized=None*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

Bernoulli

class `tfsnippet.`**Bernoulli**(*logits*, *dtype=tf.int32*)

Bases: `tfsnippet.distributions.wrapper.ZhuSuanDistribution`

Univariate Bernoulli distribution.

See also:

`tfsnippet.distributions.Distribution`, `zhusuan.distributions.Distribution`, `zhusuan.distributions.Bernoulli`

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
logits	The log-odds of probabilities of being 1.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.

Continued on next page

Table 6 – continued from previous page

prob(given[, group_ndims, name])	Compute the densities of x against the distribution.
sample([n_samples, is_reparameterized, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

logits
The log-odds of probabilities of being 1.

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)

- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *is_reparameterized=None*, *group_ndims=0*, *compute_density=None*, *name=None*)

Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

Categorical

class tfsnippet.**Categorical**(*logits*, *dtype=None*)

Bases: `tfsnippet.distributions.wrapper.ZhuSuanDistribution`

Univariate Categorical distribution.

A batch of samples is an (N-1)-D Tensor with *dtype* values in range `[0, n_categories)`.

See also:

`tfsnippet.distributions.Distribution`, `zhusuan.distributions.Distribution`, `zhusuan.distributions.Categorical`

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
logits	The un-normalized log probabilities.

Continued on next page

Table 7 – continued from previous page

n_categories	The number of categories in the distribution.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, is_reparameterized, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.

- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *is_reparameterized=None*, *group_ndims=0*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

Concrete

class tfsnippet.**Concrete**(*temperature*, *logits*, *is_reparameterized=True*, *check_numerics=None*)
Bases: tfsnippet.distributions.wrapper.ZhuSuanDistribution

The class of Concrete (or Gumbel-Softmax) distribution from (Maddison, 2016; Jang, 2016), served as the continuous relaxation of the *OnehotCategorical*.

See also:

tfsnippet.distributions.Distribution, zhusuan.distributions.Distribution, zhusuan.distributions.Concrete

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
logits	The un-normalized log probabilities.
n_categories	The number of categories in the distribution.
temperature	The temperature of this concrete distribution.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, is_reparameterized, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution

Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape

Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype

Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous

Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

temperature
The temperature of this concrete distribution.

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *is_reparameterized=None*, *group_ndims=0*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)

- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

Discrete

`tfsnippet.`**`Discrete`**

alias of `tfsnippet.distributions.univariate.Categorical`

DiscretizedLogistic

class `tfsnippet.`**`DiscretizedLogistic`**(*mean*, *log_scale*, *bin_size*, *min_val=None*, *max_val=None*, *dtype=tf.float32*, *biased_edges=True*, *discretize_given=True*, *discretize_sample=True*, *epsilon=1e-07*)

Bases: `tfsnippet.distributions.base.Distribution`

Discretized logistic distribution (Kingma et. al, 2016).

For discrete value *x* with equal intervals:

```
p(x) = sigmoid((x - mean + bin_size * 0.5) / scale) -
    sigmoid((x - mean - bin_size * 0.5) / scale)
```

where *delta* is the interval between two possible values of *x*.

The *min_val* and *max_val* specifies the minimum and maximum possible value of *x*. It should constraint the generated samples, and if *biased_edges* is True, then:

```
p(x_min) = sigmoid((x_min - mean + bin_size * 0.5) / scale)
p(x_max) = 1 - sigmoid((x_max - mean - bin_size * 0.5) / scale)
```

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
biased_edges	Whether or not to use biased density for edge values?
bin_size	Get the bin size.
discretize_given	Whether or not to discretize *given* in *log_prob()* and *prob()*?
discretize_sample	Whether or not to discretize the generated samples in *sample()*?
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
log_scale	Get the log-scale.
max_val	Get the maximum value.
mean	Get the mean.

Continued on next page

Table 11 – continued from previous page

min_val	Get the minimum value.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, group_ndims, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

biased_edges
Whether or not to use biased density for edge values?

bin_size
Get the bin size.

discretize_given
Whether or not to discretize *given* in *log_prob()* and *prob()*?

discretize_sample
Whether or not to discretize the generated samples in *sample()*?

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *`sample()`*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

log_scale
Get the log-scale.

max_val
Get the maximum value.

mean
Get the mean.

min_val
Get the minimum value.

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *`Distribution`*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *`Distribution`*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *group_ndims=0*, *is_reparameterized=None*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

Distribution

class tfsnippet.**Distribution**(*dtype*, *is_continuous*, *is_reparameterized*, *batch_shape*, *batch_static_shape*, *value_ndims*)

Bases: `object`

Base class for probability distributions.

A *Distribution* object receives inputs as distribution parameters, generating samples and computing densities according to these inputs. The shape of the inputs can have more dimensions than the nature shape of the distribution parameters, since *Distribution* is designed to work with batch parameters, samples and densities.

The shape of the parameters of a *Distribution* object would be decomposed into `batch_shape + param_shape`, with *param_shape* being the nature shape of the parameter. For example, a 5-class *Categorical* distribution with class probabilities of shape `(3, 4, 5)` would have `(3, 4)` as the *batch_shape*, with `(5,)` as the *param_shape*, corresponding to the probabilities of 5 classes.

Generating *n* samples from a *Distribution* object would result in tensors with shape `[n] (sample_shape) + batch_shape + value_shape`, with `value_shape` being the nature shape of an individual sample from the distribution. For example, the *value_shape* of a *Categorical* is `()`, such that the sample shape would be `(3, 4)`, provided the shape of class probabilities is `(3, 4, 5)`.

Computing the densities (i.e., *prob(x)* or *log_prob(x)*) of samples involves broadcasting these samples against the distribution parameters. These samples should be broadcastable against `batch_shape + value_shape`. Suppose the shape of the samples can be decomposed into `sample_shape + batch_shape + value_shape`, then by default, the shape of the densities should be `sample_shape + batch_shape`, i.e., each individual sample resulting in an individual density value.

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, group_ndims, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)

Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()

Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)

Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)

Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *group_ndims=0*, *is_reparameterized=None*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

ExpConcrete

class tfsnippet.**ExpConcrete**(*temperature*, *logits*, *is_reparameterized=True*, *check_numerics=None*)
Bases: `tfsnippet.distributions.wrapper.ZhuSuanDistribution`

The class of ExpConcrete distribution from (Maddison, 2016), transformed from *Concrete* by taking logarithm.

See also:

`tfsnippet.distributions.Distribution`, `zhusuan.distributions.Distribution`, `zhusuan.distributions.ExpConcrete`

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
logits	The un-normalized log probabilities.
n_categories	The number of categories in the distribution.
temperature	The temperature of this concrete distribution.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, is_reparameterized, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

temperature
The temperature of this concrete distribution.

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.

- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *is_reparameterized=None*, *group_ndims=0*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

FlowDistribution

class tfsnippet.**FlowDistribution**(*distribution*, *flow*)
Bases: tfsnippet.distributions.base.Distribution

Transform a *Distribution* by a `BaseFlow`, as a new distribution.

Attributes Summary

base_distribution	Get the base distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
flow	Get the transformation flow.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, group_ndims, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution.

Returns The base distribution to transform from.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

flow
Get the transformation flow.

Returns The transformation flow.

Return type *BaseFlow*

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *`sample()`*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *`Distribution`*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *`Distribution`*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *group_ndims=0*, *is_reparameterized=None*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

FlowDistributionDerivedTensor

class tfsnippet.**FlowDistributionDerivedTensor**(*tensor*, *flow_origin*)
Bases: tfsnippet.utils.tensor_wrapper.TensorWrapper

A combination of a *FlowDistribution* derived tensor, and its original stochastic tensor from the base distribution.

Attributes Summary

flow_origin	Get the original stochastic tensor from the base distribution.
tensor	

Attributes Documentation

flow_origin

Get the original stochastic tensor from the base distribution.

Returns The original stochastic tensor.

Return type *StochasticTensor*

tensor

Mixture

class tfsnippet.**Mixture**(*categorical*, *components*, *is_reparameterized=False*)

Bases: tfsnippet.distributions.base.Distribution

Mixture distribution.

Given a categorical distribution, and corresponding component distributions, this class derives a mixture distribution, formulated as follows:

$$p(x) = \sum_{k=1}^{K} \pi(k) p_k(x)$$

where $\pi(k)$ is the probability of taking the k-th component, derived by the categorical distribution, and $p_k(x)$ is the density of the k-th component distribution.

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
categorical	Get the categorical distribution of this mixture.
components	Get the mixture components of this distribution.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
n_components	Get the number of mixture components.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, group_ndims, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

categorical
Get the categorical distribution of this mixture.

Returns The categorical distribution.

Return type *Categorical*

components
Get the mixture components of this distribution.

Returns The mixture components.

Return type tuple[*Distribution*]

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

n_components
Get the number of mixture components.

Returns The number of mixture components.

Return type int

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *group_ndims=0*, *is_reparameterized=None*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

Normal

class tfsnippet.**Normal**(*mean*, *std=None*, *logstd=None*, *is_reparameterized=True*, *check_numerics=None*)

Bases: tfsnippet.distributions.wrapper.ZhuSuanDistribution

Univariate Normal distribution.

See also:

tfsnippet.distributions.Distribution, zhusuan.distributions.Distribution, zhusuan.distributions.Normal

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
logstd	Get the log standard deviation of the Normal distribution.
mean	Get the mean of the Normal distribution.
std	Get the standard deviation of the Normal distribution.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, is_reparameterized, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution

Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape

Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

logstd
Get the log standard deviation of the Normal distribution.

mean
Get the mean of the Normal distribution.

std
Get the standard deviation of the Normal distribution.

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *is_reparameterized=None*, *group_ndims=0*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have

shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.

- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

OnehotCategorical

class `tfsnippet.`**OnehotCategorical**(*logits*, *dtype=None*)

Bases: `tfsnippet.distributions.wrapper.ZhuSuanDistribution`

One-hot multivariate Categorical distribution.

A batch of samples is an N-D Tensor with *dtype* values in range `[0, n_categories)`.

See also:

`tfsnippet.distributions.Distribution`, `zhusuan.distributions.Distribution`, `zhusuan.distributions.OnehotCategorical`

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.
is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
logits	The un-normalized log probabilities.
n_categories	The number of categories in the distribution.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.

Continued on next page

Table 25 – continued from previous page

sample([n_samples, is_reparameterized, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *is_reparameterized=None*, *group_ndims=0*, *compute_density=None*, *name=None*)

Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

Uniform

class tfsnippet.**Uniform**(*minval=0.0*, *maxval=1.0*, *is_reparameterized=True*, *check_numerics=None*)

Bases: `tfsnippet.distributions.wrapper.ZhuSuanDistribution`

Univariate Uniform distribution.

See also:

`tfsnippet.distributions.Distribution`, `zhusuan.distributions.Distribution`, `zhusuan.distributions.Uniform`

Attributes Summary

base_distribution	Get the base distribution of this distribution.
batch_shape	Get the batch shape of the samples.
dtype	Get the data type of samples.

Continued on next page

Table 26 – continued from previous page

is_continuous	Whether or not the distribution is continuous?
is_reparameterized	Whether or not the distribution is re-parameterized?
maxval	The upper bound on the range of the uniform distribution.
minval	The lower bound on the range of the uniform distribution.
value_ndims	Get the number of value dimensions in samples.

Methods Summary

batch_ndims_to_value(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
expand_value_ndims(ndims)	Convert the last few *batch_ndims* into *value_ndims*.
get_batch_shape()	Get the static batch shape of the samples.
log_prob(given[, group_ndims, name])	Compute the log-densities of *x* against the distribution.
prob(given[, group_ndims, name])	Compute the densities of *x* against the distribution.
sample([n_samples, is_reparameterized, ...])	Generate samples from the distribution.

Attributes Documentation

base_distribution
Get the base distribution of this distribution.

For distribution other than *tfsnippet.BatchToValueDistribution*, this property should return this distribution itself.

Returns The base distribution.

Return type *Distribution*

batch_shape
Get the batch shape of the samples.

Returns The batch shape as tensor.

Return type tf.Tensor

dtype
Get the data type of samples.

Returns Data type of the samples.

Return type tf.DType

is_continuous
Whether or not the distribution is continuous?

Returns A boolean indicating whether it is continuous.

Return type bool

is_reparameterized
Whether or not the distribution is re-parameterized?

The re-parameterization trick is proposed in "Auto-Encoding Variational Bayes" (Kingma, D.P. and Welling), allowing the gradients to be propagated back along the samples. Note that the re-parameterization can be disabled by specifying `is_reparameterized = False` as an argument of *sample()*.

Returns A boolean indicating whether it is re-parameterized.

Return type bool

maxval
The upper bound on the range of the uniform distribution.

minval
The lower bound on the range of the uniform distribution.

value_ndims
Get the number of value dimensions in samples.

Returns The number of value dimensions in samples.

Return type int

Methods Documentation

batch_ndims_to_value(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

expand_value_ndims(*ndims*)
Convert the last few *batch_ndims* into *value_ndims*.

For a particular *Distribution*, the number of dimensions between the samples and the log-probability of the samples should satisfy:

```
samples.ndims - distribution.value_ndims == log_det.ndims
```

We denote *samples.ndims - distribution.value_ndims* by *batch_ndims*. This method thus wraps the current distribution, converts the last few *batch_ndims* into *value_ndims*.

Parameters **ndims** (*int*) – The last few *batch_ndims* to be converted into *value_ndims*. Must be non-negative.

Returns The converted distribution.

Return type *Distribution*

get_batch_shape()
Get the static batch shape of the samples.

Returns The batch shape.

Return type tf.TensorShape

log_prob(*given*, *group_ndims=0*, *name=None*)
Compute the log-densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "log_prob").

Returns The log-densities of *given*.

Return type tf.Tensor

prob(*given*, *group_ndims=0*, *name=None*)
Compute the densities of *x* against the distribution.

Parameters

- **given** (*Tensor*) – The samples to be tested.
- **group_ndims** (*int or tf.Tensor*) – If specified, the last *group_ndims* dimensions of the log-densities will be summed up. (default 0)
- **name** – TensorFlow name scope of the graph nodes. (default "prob").

Returns The densities of *given*.

Return type tf.Tensor

sample(*n_samples=None*, *is_reparameterized=None*, *group_ndims=0*, *compute_density=None*, *name=None*)
Generate samples from the distribution.

Parameters

- **n_samples** (*int or tf.Tensor or None*) – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution. The samples will have shape `[n_samples] + batch_shape + value_shape`, or `batch_shape + value_shape` if *n_samples* is `None`.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. This will effect the behavior of *log_prob()* and *prob()*. (default 0)
- **is_reparameterized** (*bool*) – If `True`, raises `RuntimeError` if the distribution is not re-parameterized. If `False`, disable re-parameterization even if the distribution is re-parameterized. (default `None`, following the setting of distribution)
- **compute_density** (*bool*) – Whether or not to immediately compute the log-density for the samples? (default `None`, determine by the distribution class itself)
- **name** – TensorFlow name scope of the graph nodes. (default "sample").

Returns

The samples as `StochasticTensor`.

Return type tfsnippet.stochastic.StochasticTensor

AnnealingVariable

class tfsnippet.**AnnealingVariable**(*name*, *initial_value*, *ratio*, *min_value=None*, *dtype=tf.float32*, *model_var=False*, *collections=None*)

Bases: tfsnippet.scaffold.scheduled_var.ScheduledVariable

A non-trainable `tf.Variable`, whose value will be annealed as training goes by.

Attributes Summary

assign_op	Get the assignment operation.
assign_ph	Get the assignment placeholder.
tensor	Get the wrapped `tf.Tensor`.
variable	Get the TensorFlow variable object.

Methods Summary

anneal()	Anneal the value.
get()	Get the current value of the variable.
set(value)	Set the value of the variable.

Attributes Documentation

assign_op

Get the assignment operation.

Returns The assignment operation.

Return type tf.Operation

assign_ph

Get the assignment placeholder.

Returns The assignment placeholder.

Return type tf.Tensor

tensor

Get the wrapped `tf.Tensor`. Derived classes must override this to return the actual wrapped tensor.

Returns The wrapped tensor.

Return type tf.Tensor

variable

Get the TensorFlow variable object.

Returns The TensorFlow variable object.

Return type tf.Variable

Methods Documentation

anneal()

Anneal the value.

Returns The new value of the variable.

get()
Get the current value of the variable.

set(*value*)
Set the value of the variable.

Parameters **value** – The value to be assigned to the variable.

Returns The new value assigned to the variable.

CheckpointSavableObject

class tfsnippet.**CheckpointSavableObject**
Bases: object

Base class for all objects that can be saved via *CheckpointSaver*.

Methods Summary

get_state()	Get the internal states of the object.
set_state(state)	Set the internal states of the object.

Methods Documentation

get_state()
Get the internal states of the object.

The returned state dict must be pickle-able.

Returns The internal states dict.

Return type dict

set_state(*state*)
Set the internal states of the object.

Parameters **state** – The internal states dict.

CheckpointSaver

class tfsnippet.**CheckpointSaver**(*variables*, *save_dir*, *objects=None*, *filename='checkpoint.dat'*, *max_to_keep=None*, *save_meta=True*, *name=None*, *scope=None*)
Bases: tfsnippet.utils.reuse.VarScopeObject

Save and restore `tf.Variable`, *ScheduledVariable* and *CheckpointSavableObject* with `tf.train.Saver`.

Attributes Summary

filename	Get the filename of checkpoint files.
name	Get the name of this object.
save_dir	Get the checkpoint directory.

Continued on next page

Table 31 – continued from previous page

save_meta	Whether or not to save graph meta?
saver	Get the TensorFlow saver object.
variable_scope	Get the variable scope of this object.

Methods Summary

latest_checkpoint()	Get the path of the latest checkpoint file.
recover_internal_states()	Restore the internal states of this saver.
restore(save_path[, session])	Restore from a checkpoint file.
restore_latest([ignore_non_exist, session])	Restore the latest checkpoint file.
save([global_step, session])	Save the session to a checkpoint file.

Attributes Documentation

filename
Get the filename of checkpoint files.

name
Get the name of this object.

save_dir
Get the checkpoint directory.

save_meta
Whether or not to save graph meta?

saver
Get the TensorFlow saver object.

Returns The TensorFlow saver object.

Return type tf.train.Saver

variable_scope
Get the variable scope of this object.

Methods Documentation

latest_checkpoint()
Get the path of the latest checkpoint file.

Returns

The path of the latest checkpoint file, or `None` if no checkpoint file is found.

Return type str or None

recover_internal_states()
Restore the internal states of this saver.

restore(*save_path*, *session=None*)
Restore from a checkpoint file.

Parameters

- **save_path** (*str*) – Restore from this checkpoint file.

- **session** (*tf.Session*) – Restore the variables into this session. If not specified, restore into the default session.

restore_latest(*ignore_non_exist=False*, *session=None*)
Restore the latest checkpoint file. :param ignore_non_exist: Whether or not to ignore error if the

latest checkpoint file does not exist?

Parameters **session** (*tf.Session*) – Restore the variables into this session. If not specified, restore into the default session.

Raises IOError – If no checkpoint file is found.

save(*global_step=None*, *session=None*)
Save the session to a checkpoint file.

Parameters

- **global_step** (*int or tf.Tensor*) – The global step counter.
- **session** (*tf.Session*) – The session to save. If not specified, select the default session.

Returns The path of the saved checkpoint file.

Return type str

DefaultMetricFormatter

class tfsnippet.**DefaultMetricFormatter**
Bases: tfsnippet.scaffold.logging_.MetricFormatter

Default training metric formatter.

The time metrics (names ending with "time" or "timer") should be placed before all other metrics. The values of the metrics would be formatted into 6-digit real numbers, except for metrics with "time" or "timer" as suffices in their names, which would be formatted using *humanize_duration()*.

Attributes Summary

METRIC_ORDERS

Methods Summary

format_metric(name, value)	Format the value of specified metric.
sort_metrics(names)	Sort the names of metrics.

Attributes Documentation

METRIC_ORDERS = [(-1, <_sre.SRE_Pattern object>)]

Methods Documentation

format_metric(*name*, *value*)
Format the value of specified metric.

Parameters

- **name** – Name of the metric.
- **value** – Value of the metric.

Returns Human readable string representation of the metric value.

Return type str

sort_metrics(*names*)
Sort the names of metrics.

Parameters **names** – Iterable metric names.

Returns Sorted metric names.

Return type list[str]

EventKeys

class tfsnippet.**EventKeys**
Bases: object

Defines event keys for TFSnippet.

Attributes Summary

AFTER_EPOCH
AFTER_EXECUTION
AFTER_STEP
BEFORE_EPOCH
BEFORE_EXECUTION
BEFORE_STEP
ENTER_LOOP
EPOCH_ANNEALING
EPOCH_EVALUATION
EPOCH_LOGGING
EXIT_LOOP
METRICS_COLLECTED
METRIC_STATS_PRINTED
STEP_ANNEALING
STEP_EVALUATION
STEP_LOGGING
SUMMARY_ADDED
TIME_METRICS_COLLECTED
TIME_METRIC_STATS_PRINTED

Attributes Documentation

AFTER_EPOCH = 'after_epoch'

AFTER_EXECUTION = 'after_execution'

AFTER_STEP = 'after_step'

BEFORE_EPOCH = 'before_epoch'

BEFORE_EXECUTION = 'before_execution'

BEFORE_STEP = 'before_step'

ENTER_LOOP = 'enter_loop'

EPOCH_ANNEALING = 'epoch_annealing'

EPOCH_EVALUATION = 'epoch_evaluation'

EPOCH_LOGGING = 'epoch_logging'

EXIT_LOOP = 'exit_loop'

METRICS_COLLECTED = 'metrics_collected'

METRIC_STATS_PRINTED = 'metric_stats_printed'

STEP_ANNEALING = 'step_annealing'

STEP_EVALUATION = 'step_evaluation'

STEP_LOGGING = 'step_logging'

SUMMARY_ADDED = 'summary_added'

TIME_METRICS_COLLECTED = 'time_metrics_collected'

TIME_METRIC_STATS_PRINTED = 'time_metric_stats_printed'

MetricFormatter

class tfsnippet.**MetricFormatter**

Bases: object

Base class for a training metrics formatter.

A training metric formatter determines the order of metrics, and the way to display the values of these metrics, in *MetricLogger*.

Methods Summary

format_metric(name, value)	Format the value of specified metric.
sort_metrics(names)	Sort the names of metrics.

Methods Documentation

format_metric(*name*, *value*)

Format the value of specified metric.

Parameters

- **name** – Name of the metric.
- **value** – Value of the metric.

Returns Human readable string representation of the metric value.

Return type str

sort_metrics(*names*)
Sort the names of metrics.

Parameters **names** – Iterable metric names.

Returns Sorted metric names.

Return type list[str]

MetricLogger

class tfsnippet.**MetricLogger**(*summary_writer=None*, *summary_metric_prefix=''*, *summary_skip_pattern=None*, *summary_commit_freqs=None*, *formatter=None*)

Bases: object

Logger for the training metrics.

This class provides convenient methods for logging training metrics, and for writing metrics onto disk via TensorFlow summary writer. The statistics of the metrics could be formatted into human readable strings via *format_logs()*.

An example of using this logger is:

```
logger = MetricLogger(tf.summary.FileWriter(log_dir))
global_step = 1

for epoch in range(1, max_epoch+1):
    for batch in DataFlow.arrays(...):
        loss, _ = session.run([loss, train_op], ...)
        logger.collect_metrics({'loss': loss}, global_step)
        global_step += 1

    valid_loss = session.run([loss], ...)
    logger.collect_metrics({'valid_loss': valid_loss}, global_step)
    print('Epoch {}, step {}: {}'.format(
        epoch, global_step, logger.format_logs()))
    logger.clear()
```

Attributes Summary

metrics	Get the dict of metric collectors.

Methods Summary

clear()	Clear all the metric statistics.
collect_metrics(metrics[, global_step])	Collect the statistics of metrics.

Continued on next page

Table 38 – continued from previous page

format_logs()	Format the metric statistics as human readable strings.

Attributes Documentation

metrics
Get the dict of metric collectors.

Returns The metric collectors.

Return type dict[str, *StatisticsCollector*]

Methods Documentation

clear()
Clear all the metric statistics.

collect_metrics(*metrics*, *global_step=None*)
Collect the statistics of metrics.

Parameters

- **metrics** (*dict[str, float or np.ndarray or ScheduledVariable]*) – Dict from metrics names to their values. For *format_logs()*, there is no difference between calling *collect_metrics()* only once, with an array of metric values; or calling *collect_metrics()* multiple times, with one value at each time. However, for the TensorFlow summary writer, only the mean of the metric values would be recorded, if calling *collect_metrics()* with an array.
- **global_step** (*int or tf.Variable or tf.Tensor*) – The global step counter. (optional)

format_logs()
Format the metric statistics as human readable strings.

Returns The formatted metric statistics.

Return type str

ScheduledVariable

class tfsnippet.**ScheduledVariable**(*name*, *initial_value*, *dtype=tf.float32*, *model_var=False*, *collections=None*)
Bases: tfsnippet.utils.tensor_wrapper.TensorWrapper

A non-trainable tf.Variable, whose value might need to be changed as training goes by.

Attributes Summary

assign_op	Get the assignment operation.
assign_ph	Get the assignment placeholder.
tensor	Get the wrapped tf.Tensor.
variable	Get the TensorFlow variable object.

Methods Summary

get()	Get the current value of the variable.
set(value)	Set the value of the variable.

Attributes Documentation

assign_op
Get the assignment operation.

Returns The assignment operation.

Return type tf.Operation

assign_ph
Get the assignment placeholder.

Returns The assignment placeholder.

Return type tf.Tensor

tensor
Get the wrapped `tf.Tensor`. Derived classes must override this to return the actual wrapped tensor.

Returns The wrapped tensor.

Return type tf.Tensor

variable
Get the TensorFlow variable object.

Returns The TensorFlow variable object.

Return type tf.Variable

Methods Documentation

get()
Get the current value of the variable.

set(*value*)
Set the value of the variable.

Parameters **value** – The value to be assigned to the variable.

Returns The new value assigned to the variable.

TrainLoop

class tfsnippet.**TrainLoop**(*param_vars*, *var_groups=None*, *show_eta=True*, *print_func=<built-in function print>*, *max_epoch=None*, *max_step=None*, *metric_formatter=<tfsnippet.scaffold.logging_.DefaultMetricFormatter object>*, *checkpoint_dir=None*, *checkpoint_epoch_freq=None*, *checkpoint_max_to_keep=None*, *checkpoint_save_objects=None*, *restore_checkpoint=True*, *summary_dir=None*, *summary_writer=None*, *summary_graph=None*, *summary_metric_prefix='metrics/'*, *summary_skip_pattern=<_sre.SRE_Pattern object>*, *summary_commit_freqs=None*, *valid_metric_name='valid_loss'*, *valid_metric_smaller_is_better=None*, *early_stopping=False*)

Bases: tfsnippet.utils.concepts.DisposableContext

Training loop object.

This class provides a set of convenient methods for writing training loop. It is useful for maintaining epoch and step counters, logging training metrics, memorizing best parameters for early-stopping, etc. An example of using the *TrainLoop*:

```
import tfsnippet as spt

with spt.TrainLoop(param_vars,
                   max_epoch=10,
                   early_stopping=True) as loop:
    loop.print_training_summary()
    train_flow = spt.DataFlow.arrays([x, y], batch_size, shuffle=True)

    for epoch in loop.iter_epochs():
        for step, (x, y) in loop.iter_steps(train_flow):
            step_loss = session.run(
                [loss, train_op],
                feed_dict={input_x: x, input_y: y}
            )
            loop.collect_metrics(loss=step_loss)
        with loop.timeit('valid_time'):
            valid_loss = session.run(
                loss, feed_dict={input_x: test_x, input_y: test_y})
            loop.collect_metrics(valid_loss=valid_loss)
        loop.print_logs()
```

The event schedule of a *TrainLoop* can be briefly described as:

```
# the main training loop
events.fire(EventKeys.ENTER_LOOP, self)

for epoch in self.iter_epochs():
    events.fire(EventKeys.BEFORE_EPOCH, self)

    for step in self.iter_steps(...):
        events.fire(EventKeys.BEFORE_STEP, self)

        ...  # execute the step

        events.reverse_fire(EventKeys.AFTER_STEP, self)

    events.reverse_fire(EventKeys.AFTER_EPOCH, self)
```

(continues on next page)

(continued from previous page)

```
events.fire(EventKeys.EXIT_LOOP, self)

# when metrics are fed into the loop by :meth:`collect_metrics`
def collect_metrics(self, metrics_dict=None, **kwargs):
    metrics_dict = merge(metrics_dict, kwargs)
    events.fire(EventKeys.METRICS_COLLECTED, self, metrics_dict)

# when summaries are fed into the loop by :meth:`add_summary`
def add_summary(self, summary):
    events.fire(EventKeys.SUMMARY_ADDED, self, summary)

# when metric statistics have been printed as log
def print_logs(self):
    ...
    events.fire(EventKeys.METRIC_STATS_PRINTED, self, metric_stats)
    events.fire(EventKeys.TIME_METRIC_STATS_PRINTED, self,
                time_metric_stats)
```

Warning: If you use early-stopping along with checkpoint, there is one case which is very dangerous: you've already successfully done a training loop, and the early-stopping variables have been restored. But you then recover from the latest checkpoint and continue to train. In this case, the *param_vars* (which is covered by early-stopping) are restored to the best validation step, but the other variables and the internal states of *TrainLoop* are recovered to the last step. Then you obtain a state mismatch, and the behaviour will be un-predictable after this recovery.

Attributes Summary

best_valid_metric	Get the best valid metric.
epoch	Get the epoch counter (starting from 1).
events	Get the event source.
max_epoch	Get or set the max value for epoch counter.
max_step	Get or set the max value for global step counter.
param_vars	Get the trainable parameter variables.
step	Get the global step counter (starting from 1).
step_data	Get the data of current step.
summary_writer	Get the summary writer instance.
use_early_stopping	Whether or not to adopt early-stopping?
valid_metric_name	Get the name of the validation metric.
valid_metric_smaller_is_better	Whether or not the smaller value is better for validation metric?
var_groups	Get the variable groups.
within_epoch	Whether or not an epoch is open?
within_step	Whether or not a step is open?

Methods Summary

add_summary(summary)	Add a summary object, with `self.step` as *global_step*.
collect_metrics([metrics])	Add metric values.
get_eta()	Get the estimated time ahead (ETA).
get_progress()	Get the progress of training.
iter_epochs()	Iterate through the epochs.
iter_steps([data_generator])	Iterate through the steps.
make_checkpoint()	Make a checkpoint.
metric_collector(**kwds)	Get a *StatisticsCollector* for metric.
print_logs()	Print the training logs.
print_training_summary()	Print the training summary.
println(message[, with_tag])	Print *message* via *print_function*.
timeit(**kwds)	Open a context for timing.

Attributes Documentation

best_valid_metric
Get the best valid metric.

epoch
Get the epoch counter (starting from 1).

events
Get the event source.

Returns The event source.

Return type *EventSource*

max_epoch
Get or set the max value for epoch counter.

max_step
Get or set the max value for global step counter.

param_vars
Get the trainable parameter variables.

step
Get the global step counter (starting from 1).

step_data
Get the data of current step.

summary_writer
Get the summary writer instance.

use_early_stopping
Whether or not to adopt early-stopping?

valid_metric_name
Get the name of the validation metric.

valid_metric_smaller_is_better
Whether or not the smaller value is better for validation metric?

var_groups
Get the variable groups.

within_epoch
Whether or not an epoch is open?

within_step
Whether or not a step is open?

Methods Documentation

add_summary(*summary*)
Add a summary object, with `self.step` as *global_step*.

Parameters **summary** (*tf.summary.Summary or bytes*) – TensorFlow summary object, or serialized summary.

collect_metrics(*metrics=None*, ***kwargs*)
Add metric values.

This method must be called when there's at least an active epoch loop. It will add metrics to the epoch metrics collector, and if there's an active step loop, it will also add metrics to the step metrics collector.

If *summary_writer* is configured, it will also write the metrics as summaries onto disk. Furthermore, if *valid_metric_name* is configured, it will also perform early-stopping.

Parameters

- **metrics** (*dict[str, float or np.ndarray]*) – Metric values as dict.
- ****kwargs** – Metric values, specified as named arguments.

get_eta()
Get the estimated time ahead (ETA).

Returns

The estimated time ahead in seconds, or None if not available.

Return type float or None

get_progress()
Get the progress of training.

Returns

The progress in range [0, 1], or None if the progress cannot be estimated.

Return type float or None

iter_epochs()
Iterate through the epochs.

This method can only be called when there's no other epoch loop is being iterated. Furthermore, after exiting this loop, both the epoch metrics as well as the step metrics will be cleared.

If *max_epoch* is configured, it will stop at it.

Yields *int* – The epoch counter (starting from 1).

iter_steps(*data_generator=None*)
Iterate through the steps.

This method can only be called when there's no other step loop is being iterated, and an epoch loop is active.

Parameters **data_generator** – Optional iterable data to be yielded at every step. This is required if *max_step* is not configured, so as to prevent an infinite step loop.

Yields *int or (int, any)* –

The global step counter (starting from 1), or the tuple of `(step counter, batch data)` if *data_generator* is specified.

make_checkpoint()
Make a checkpoint.

This method must be called within an eopch or a step context. For example:

```
for epoch in loop.iter_epochs():
    for [x] in loop.iter_steps(train_data):
        ...

    if epoch % 100 == 0:
        loop.make_checkpoint()
```

metric_collector(**kwds*)
Get a *StatisticsCollector* for metric.

The mean value of the collected metrics will be added to summary after exiting the context. Other statistics will be discarded.

Parameters **metric_name** (*str*) – The name of this metric.

Yields *StatisticsCollector* – The collector for metric values.

print_logs()
Print the training logs.

This method will print the collected metrics. If there's an active step loop, it will print metrics from the step metrics collector. Otherwise if there's only an epoch loop, it will print metrics from the epoch metrics accumulator.

Note it must be called at the end of an epoch or a step. This is because the metrics of corresponding loop context will be cleared after the logs are printed. Moreover, the epoch or step timer will be committed as metric immediately when this method is called, before printing the logs.

print_training_summary()
Print the training summary.

The training summary include the following content:

1. Execution environment.
2. Parameters to be optimized during training.

println(*message*, *with_tag=False*)
Print *message* via *print_function*.

Parameters

- **message** (*str*) – Message to be printed.
- **with_tag** (*bool*) – Whether or not to add the epoch & step tag? (default `False`)

timeit(**kwds*)
Open a context for timing.

Parameters **metric_name** (*str*) – Store the timing result in metric of this name. Note that *metric_name* must end with `time` or `timer`, otherwise by default the time values will not be formatted as human readable strings.

AnnealingScalar

class tfsnippet.**AnnealingScalar**(*loop*, *initial_value*, *ratio*, *epochs=None*, *steps=None*, *min_value=None*, *max_value=None*)

Bases: tfsnippet.trainer.dynamic_values.DynamicValue

A *DynamicValue* scalar, which anneals every few epochs or steps.

For example, to anneal the learning rate every 100 epochs:

```
learning_rate = tf.placeholder(dtype=tf.float32, shape=())
...

with spt.TrainLoop(...) as loop:
    trainer = spt.Trainer(
        ...,
        feed_dict={learning_rate: spt.AnnealingScalar(
            loop, initial=0.001, ratio=0.5, epochs=100)}
    )
```

Methods Summary

get()	Get the current value of this *DynamicValue* object.

Methods Documentation

get()

Get the current value of this *DynamicValue* object.

BaseTrainer

class tfsnippet.**BaseTrainer**(*loop*, *ensure_variables_initialized=True*)

Bases: object

Base class for all trainers.

All the trainers provided in tfsnippet.trainer are not designed to take control of the training totally, which is often assumed in other libraries such as Keras. Instead, it just takes responsibility of assembling different steps of a training process together, and run the main training loop. So it is usually the caller's responsibility to derive his training operation from a certain TensorFlow optimizer, and pass it to a proper trainer.

The event schedule of a *BaseTrainer* can be briefly described as:

```
events.fire(EventKeys.BEFORE_EXECUTION, self)

for epoch in epochs:
    events.fire(EventKeys.BEFORE_EPOCH, self)

    for step in steps:
        events.fire(EventKeys.BEFORE_STEP, self)

        ...  # actually train for a step
```

(continues on next page)

(continued from previous page)

```
        events.fire(EventKeys.STEP_EVALUATION, self)
        events.fire(EventKeys.STEP_ANNEALING, self)
        events.fire(EventKeys.STEP_LOGGING, self)
        events.reverse_fire(EventKeys.AFTER_STEP, self)

    events.fire(EventKeys.EPOCH_EVALUATION, self)
    events.fire(EventKeys.EPOCH_ANNEALING, self)
    events.fire(EventKeys.EPOCH_LOGGING, self)
    events.reverse_fire(EventKeys.AFTER_EPOCH, self)

events.reverse_fire(EventKeys.AFTER_EXECUTION, self)
```

Using *trainer.events.on(EventKeys.AFTER_EPOCH, lambda trainer: ...)* can register an after-epoch event handler. Handlers for other events can be registered in a similar way.

To make things even simpler, we provide several methods to register callbacks that will run every few epochs/steps, e.g.:

```
trainer.evaluate_after_epochs(
    lambda: print('after epoch callback'), 10)  # run every 10 epochs
trainer.log_after_steps(1000)  # call `loop.print_logs` every 1000 steps
```

Attributes Summary

events	Get the event source object.
loop	Get the training loop object.

Methods Summary

anneal_after(value[, epochs, steps])	Add an annealing hook to run after every few epochs or steps.
anneal_after_epochs(value, freq)	Add an annealing hook to run after every few epochs.
anneal_after_steps(value, freq)	Add an annealing hook to run after every few steps.
evaluate_after(evaluator[, epochs, steps])	Add an evaluation hook to run after every few epochs or steps.
evaluate_after_epochs(evaluator, freq)	Add an evaluation hook to run after every few epochs.
evaluate_after_steps(evaluator, freq)	Add an evaluation hook to run after every few steps.
log_after([epochs, steps])	Add a logging hook to run after every few epochs or steps.
log_after_epochs(freq)	Add a logging hook to run after every few epochs.
log_after_steps(freq)	Add a logging hook to run after every few steps.
remove_annealing_hooks()	Remove annealing hooks from all lists.
remove_evaluation_hooks()	Remove evaluation hooks from all lists.
remove_log_hooks()	Remove logging hooks from all lists.
remove_validation_hooks()	Remove evaluation hooks from all lists.
run()	Run training loop.
validate_after(evaluator[, epochs, steps])	Add an evaluation hook to run after every few epochs or steps.

Continued on next page

Table 45 – continued from previous page

validate_after_epochs(evaluator, freq)	Add an evaluation hook to run after every few epochs.
validate_after_steps(evaluator, freq)	Add an evaluation hook to run after every few steps.

Attributes Documentation

events
Get the event source object.

Returns The event source object.

Return type *EventSource*

loop
Get the training loop object.

Returns The training loop object.

Return type *TrainLoop*

Methods Documentation

anneal_after(*value*, *epochs=None*, *steps=None*)
Add an annealing hook to run after every few epochs or steps.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises `ValueError` – If both *epochs* and *steps* are specified, or neither is specified.

anneal_after_epochs(*value*, *freq*)
Add an annealing hook to run after every few epochs.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **freq** (*int*) – The frequency for this annealing hook to run.

anneal_after_steps(*value*, *freq*)
Add an annealing hook to run after every few steps.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **freq** (*int*) – The frequency for this annealing hook to run.

evaluate_after(*evaluator*, *epochs=None*, *steps=None*)
Add an evaluation hook to run after every few epochs or steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises ValueError – If both *epochs* and *steps* are specified, or neither is specified.

evaluate_after_epochs(*evaluator*, *freq*)
Add an evaluation hook to run after every few epochs.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

evaluate_after_steps(*evaluator*, *freq*)
Add an evaluation hook to run after every few steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

log_after(*epochs=None*, *steps=None*)
Add a logging hook to run after every few epochs or steps.

Parameters

- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises ValueError – If both *epochs* and *steps* are specified, or neither is specified.

log_after_epochs(*freq*)
Add a logging hook to run after every few epochs.

Parameters **freq** (*int*) – The frequency for this logging hook to run.

log_after_steps(*freq*)
Add a logging hook to run after every few steps.

Parameters **freq** (*int*) – The frequency for this logging hook to run.

remove_annealing_hooks()
Remove annealing hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_evaluation_hooks()
Remove evaluation hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_log_hooks()
Remove logging hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_validation_hooks()
Remove evaluation hooks from all lists.

Returns The number of removed hooks.

Return type int

run()
Run training loop.

validate_after(*evaluator*, *epochs=None*, *steps=None*)
Add an evaluation hook to run after every few epochs or steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises ValueError – If both *epochs* and *steps* are specified, or neither is specified.

validate_after_epochs(*evaluator*, *freq*)
Add an evaluation hook to run after every few epochs.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

validate_after_steps(*evaluator*, *freq*)
Add an evaluation hook to run after every few steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

DynamicValue

class tfsnippet.**DynamicValue**
Bases: object

Dynamic values to be fed into trainers and evaluators.

For example, if you want to feed a learning rate into trainer, which shrinks into half every 100 epochs, you may use the following code:

```
class MyLearningRate(spt.DynamicValue):

    def __init__(self, loop):
        self.loop = loop

    def get(self):
        return 0.001 * int(self.loop.epoch // 100) * 0.5
```

(continues on next page)

(continued from previous page)

```
learning_rate = tf.placeholder(dtype=tf.float32, shape=())
...

with spt.TrainLoop(...) as loop:
    trainer = spt.Trainer(
        ...,
        feed_dict={learning_rate: MyLearningRate(loop)}
    )
    trainer.run()
```

Or you may also use *AnnealingScalar*, a class that has already implemented such behaviour.

Methods Summary

get()	Get the current value of this *DynamicValue* object.

Methods Documentation

get()
Get the current value of this *DynamicValue* object.

Evaluator

class tfsnippet.**Evaluator**(*loop*, *metrics*, *inputs*, *data_flow*, *feed_dict=None*, *time_metric_name='eval_time'*, *batch_weight_func=<function auto_batch_weight>*)

Bases: object

Class to compute evaluation metrics.

It is a common practice to compute one or more metrics for evaluation and validation during the training process. This class provides a convenient interface for computing metrics by mini-batches.

The event schedule of an *Evaluator* can be briefly described as follows:

```
events.fire(EventKeys.BEFORE_EXECUTION, self)

...  # actually run the evaluation

events.reverse_fire(EventKeys.AFTER_EXECUTION, self)
```

Attributes Summary

batch_weight_func	Get the function to compute the metric weight for each mini-batch.
data_flow	Get the validation data flow.
events	Get the event source object.
feed_dict	Get the fixed feed dict.

Continued on next page

Table 47 – continued from previous page

inputs	Get the input placeholders.
last_metrics_dict	Get the metric values from last evaluation.
loop	Get the training loop object.
metrics	Get the metrics to compute.
time_metric_name	Get the metric name for collecting evaluation time usage.

Methods Summary

run([feed_dict])	Run evaluation.

Attributes Documentation

batch_weight_func
Get the function to compute the metric weight for each mini-batch.

data_flow
Get the validation data flow.

Returns The validation data flow.

Return type *DataFlow*

events
Get the event source object.

Returns The event source object.

Return type *EventSource*

feed_dict
Get the fixed feed dict.

Returns The fixed feed dict.

Return type dict[tf.Tensor, any]

inputs
Get the input placeholders.

Returns The input placeholders.

Return type list[tf.Tensor]

last_metrics_dict
Get the metric values from last evaluation.

Returns The metric values dict.

Return type dict[str, any]

loop
Get the training loop object.

Returns The training loop object.

Return type *TrainLoop*

metrics
Get the metrics to compute.

Returns The metrics to compute.

Return type OrderedDict[str, tf.Tensor]

time_metric_name
Get the metric name for collecting evaluation time usage.

Methods Documentation

run(*feed_dict=None*)
Run evaluation.

Parameters **feed_dict** – The extra feed dict to be merged with the already configured dict. (default None)

LossTrainer

class tfsnippet.**LossTrainer**(***kwargs*)
Bases: tfsnippet.trainer.trainer.Trainer

A subclass of *BaseTrainer*, which optimizes a single loss.

Attributes Summary

data_flow	Get the training data flow.
events	Get the event source object.
feed_dict	Get the feed dict for training.
inputs	Get the input placeholders.
loop	Get the training loop object.
loss	Get the training loss.
metric_name	Get the metric name for collecting training loss.
metrics	Get the metrics to be computed along with *train_op*.
summaries	Get the summaries to be computed along with *train_op*.
train_op	Get the training operation.

Methods Summary

anneal_after(value[, epochs, steps])	Add an annealing hook to run after every few epochs or steps.
anneal_after_epochs(value, freq)	Add an annealing hook to run after every few epochs.
anneal_after_steps(value, freq)	Add an annealing hook to run after every few steps.
evaluate_after(evaluator[, epochs, steps])	Add an evaluation hook to run after every few epochs or steps.
evaluate_after_epochs(evaluator, freq)	Add an evaluation hook to run after every few epochs.
evaluate_after_steps(evaluator, freq)	Add an evaluation hook to run after every few steps.
log_after([epochs, steps])	Add a logging hook to run after every few epochs or steps.
log_after_epochs(freq)	Add a logging hook to run after every few epochs.

Continued on next page

Table 50 – continued from previous page

log_after_steps(freq)	Add a logging hook to run after every few steps.
remove_annealing_hooks()	Remove annealing hooks from all lists.
remove_evaluation_hooks()	Remove evaluation hooks from all lists.
remove_log_hooks()	Remove logging hooks from all lists.
remove_validation_hooks()	Remove evaluation hooks from all lists.
run(**kwargs)	Run training loop.
validate_after(evaluator[, epochs, steps])	Add an evaluation hook to run after every few epochs or steps.
validate_after_epochs(evaluator, freq)	Add an evaluation hook to run after every few epochs.
validate_after_steps(evaluator, freq)	Add an evaluation hook to run after every few steps.

Attributes Documentation

data_flow
Get the training data flow.

Returns The training data flow.

Return type *DataFlow*

events
Get the event source object.

Returns The event source object.

Return type *EventSource*

feed_dict
Get the feed dict for training.

Returns The feed dict for training.

Return type dict[tf.Tensor, any]

inputs
Get the input placeholders.

Returns The input placeholders.

Return type list[tf.Tensor]

loop
Get the training loop object.

Returns The training loop object.

Return type *TrainLoop*

loss
Get the training loss.

metric_name
Get the metric name for collecting training loss.

metrics
Get the metrics to be computed along with *train_op*.

summaries
Get the summaries to be computed along with *train_op*.

train_op
Get the training operation.

Methods Documentation

anneal_after(*value*, *epochs=None*, *steps=None*)
Add an annealing hook to run after every few epochs or steps.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises `ValueError` – If both *epochs* and *steps* are specified, or neither is specified.

anneal_after_epochs(*value*, *freq*)
Add an annealing hook to run after every few epochs.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **freq** (*int*) – The frequency for this annealing hook to run.

anneal_after_steps(*value*, *freq*)
Add an annealing hook to run after every few steps.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **freq** (*int*) – The frequency for this annealing hook to run.

evaluate_after(*evaluator*, *epochs=None*, *steps=None*)
Add an evaluation hook to run after every few epochs or steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has `.run()`), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises `ValueError` – If both *epochs* and *steps* are specified, or neither is specified.

evaluate_after_epochs(*evaluator*, *freq*)
Add an evaluation hook to run after every few epochs.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has `.run()`), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

evaluate_after_steps(*evaluator*, *freq*)
Add an evaluation hook to run after every few steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has `.run()`), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

log_after(*epochs=None*, *steps=None*)
Add a logging hook to run after every few epochs or steps.

Parameters

- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises `ValueError` – If both *epochs* and *steps* are specified, or neither is specified.

log_after_epochs(*freq*)
Add a logging hook to run after every few epochs.

Parameters **freq** (*int*) – The frequency for this logging hook to run.

log_after_steps(*freq*)
Add a logging hook to run after every few steps.

Parameters **freq** (*int*) – The frequency for this logging hook to run.

remove_annealing_hooks()
Remove annealing hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_evaluation_hooks()
Remove evaluation hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_log_hooks()
Remove logging hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_validation_hooks()
Remove evaluation hooks from all lists.

Returns The number of removed hooks.

Return type int

run(***kwargs*)
Run training loop.

Parameters **feed_dict** – DEPRECATED. The extra feed dict to be merged with the already configured dict. (default `None`)

validate_after(*evaluator*, *epochs=None*, *steps=None*)
Add an evaluation hook to run after every few epochs or steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises ValueError – If both *epochs* and *steps* are specified, or neither is specified.

validate_after_epochs(*evaluator*, *freq*)
Add an evaluation hook to run after every few epochs.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

validate_after_steps(*evaluator*, *freq*)
Add an evaluation hook to run after every few steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

Trainer

class tfsnippet.**Trainer**(*loop*, *train_op*, *inputs*, *data_flow*, *feed_dict=None*, *metrics=None*, *summaries=None*, *ensure_variables_initialized=True*)
Bases: tfsnippet.trainer.base_trainer.BaseTrainer

A subclass of *BaseTrainer*, executing a training operation per step. This might be the most commonly used *Trainer*. Code example:

```
import tfsnippet as spt

# build the model
input_x = tf.placeholder(...)
input_y = tf.placeholder(...)
learning_rate = tf.placeholder(...)  # learning rate annealing

# prepare for the data and
train_data = spt.DataFlow.arrays(
    [train_x, train_y], batch_size=128, shuffle=True,
    skip_incomplete=True
)
valid_data = spt.DataFlow.arrays(
    [valid_x, valid_y], batch_size=512)
...

# derive the training operation
optimizer = tf.train.AdamOptimizer(learning_rate)
train_op = optimizer.minimize(loss)

# run the trainer
learning_rate = spt.AnnealingVariable('learning_rate', 0.001, 0.75)
```

(continues on next page)

(continued from previous page)

```
with spt.TrainLoop(param_vars,
                   max_epoch=10,
                   early_stopping=True) as loop:
    trainer = spt.Trainer(
        loop, train_op, [input_x, input_y], train_data,
        metrics={'loss': loss}
    )
    evaluator = spt.Evaluator(
        loop, {'loss': loss}, [input_x, input_y], valid_data)

    # validate after every epoch
    trainer.evaluate_after_epochs(evaluator, freq=1)

    # log after every epoch (and after validation, since
    # ``HookPriority.VALIDATION < HookPriority.LOGGING``)
    trainer.log_after_epochs(freq=1)

    # anneal the learning rate after every 10 epochs
    trainer.anneal_after_epochs(learning_rate, freq=10)

    # run the main training loop
    trainer.run()
```

See also:

`tfsnippet.trainer.BaseTrainer`

Attributes Summary

data_flow	Get the training data flow.
events	Get the event source object.
feed_dict	Get the feed dict for training.
inputs	Get the input placeholders.
loop	Get the training loop object.
metrics	Get the metrics to be computed along with *train_op*.
summaries	Get the summaries to be computed along with *train_op*.
train_op	Get the training operation.

Methods Summary

anneal_after(value[, epochs, steps])	Add an annealing hook to run after every few epochs or steps.
anneal_after_epochs(value, freq)	Add an annealing hook to run after every few epochs.
anneal_after_steps(value, freq)	Add an annealing hook to run after every few steps.
evaluate_after(evaluator[, epochs, steps])	Add an evaluation hook to run after every few epochs or steps.
evaluate_after_epochs(evaluator, freq)	Add an evaluation hook to run after every few epochs.
evaluate_after_steps(evaluator, freq)	Add an evaluation hook to run after every few steps.

Continued on next page

Table 52 – continued from previous page

log_after([epochs, steps])	Add a logging hook to run after every few epochs or steps.
log_after_epochs(freq)	Add a logging hook to run after every few epochs.
log_after_steps(freq)	Add a logging hook to run after every few steps.
remove_annealing_hooks()	Remove annealing hooks from all lists.
remove_evaluation_hooks()	Remove evaluation hooks from all lists.
remove_log_hooks()	Remove logging hooks from all lists.
remove_validation_hooks()	Remove evaluation hooks from all lists.
run()	Run training loop.
validate_after(evaluator[, epochs, steps])	Add an evaluation hook to run after every few epochs or steps.
validate_after_epochs(evaluator, freq)	Add an evaluation hook to run after every few epochs.
validate_after_steps(evaluator, freq)	Add an evaluation hook to run after every few steps.

Attributes Documentation

data_flow
Get the training data flow.

Returns The training data flow.

Return type *DataFlow*

events
Get the event source object.

Returns The event source object.

Return type *EventSource*

feed_dict
Get the feed dict for training.

Returns The feed dict for training.

Return type dict[tf.Tensor, any]

inputs
Get the input placeholders.

Returns The input placeholders.

Return type list[tf.Tensor]

loop
Get the training loop object.

Returns The training loop object.

Return type *TrainLoop*

metrics
Get the metrics to be computed along with *train_op*.

summaries
Get the summaries to be computed along with *train_op*.

train_op
Get the training operation.

Methods Documentation

anneal_after(*value*, *epochs=None*, *steps=None*)
Add an annealing hook to run after every few epochs or steps.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises `ValueError` – If both *epochs* and *steps* are specified, or neither is specified.

anneal_after_epochs(*value*, *freq*)
Add an annealing hook to run after every few epochs.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **freq** (*int*) – The frequency for this annealing hook to run.

anneal_after_steps(*value*, *freq*)
Add an annealing hook to run after every few steps.

Parameters

- **value** (*AnnealingVariable or () -> any*) – An annealing variable (which has `.anneal()`), or any callable object.
- **freq** (*int*) – The frequency for this annealing hook to run.

evaluate_after(*evaluator*, *epochs=None*, *steps=None*)
Add an evaluation hook to run after every few epochs or steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has `.run()`), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises `ValueError` – If both *epochs* and *steps* are specified, or neither is specified.

evaluate_after_epochs(*evaluator*, *freq*)
Add an evaluation hook to run after every few epochs.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has `.run()`), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

evaluate_after_steps(*evaluator*, *freq*)
Add an evaluation hook to run after every few steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has `.run()`), or any callable object.

- **freq** (*int*) – The frequency for this evaluation hook to run.

log_after(*epochs=None*, *steps=None*)
Add a logging hook to run after every few epochs or steps.

Parameters

- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises ValueError – If both *epochs* and *steps* are specified, or neither is specified.

log_after_epochs(*freq*)
Add a logging hook to run after every few epochs.

Parameters **freq** (*int*) – The frequency for this logging hook to run.

log_after_steps(*freq*)
Add a logging hook to run after every few steps.

Parameters **freq** (*int*) – The frequency for this logging hook to run.

remove_annealing_hooks()
Remove annealing hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_evaluation_hooks()
Remove evaluation hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_log_hooks()
Remove logging hooks from all lists.

Returns The number of removed hooks.

Return type int

remove_validation_hooks()
Remove evaluation hooks from all lists.

Returns The number of removed hooks.

Return type int

run()
Run training loop.

validate_after(*evaluator*, *epochs=None*, *steps=None*)
Add an evaluation hook to run after every few epochs or steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has .run()), or any callable object.
- **epochs** (*None or int*) – Run validation after every this few *epochs*.
- **steps** (*None or int*) – Run validation after every this few *steps*.

Raises ValueError – If both *epochs* and *steps* are specified, or neither is specified.

validate_after_epochs(*evaluator*, *freq*)
Add an evaluation hook to run after every few epochs.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has `.run()`), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

validate_after_steps(*evaluator*, *freq*)
Add an evaluation hook to run after every few steps.

Parameters

- **evaluator** (*Evaluator or () -> any*) – A evaluator object (which has `.run()`), or any callable object.
- **freq** (*int*) – The frequency for this evaluation hook to run.

Validator

class tfsnippet.**Validator**(***kwargs*)
Bases: tfsnippet.trainer.evaluator.Evaluator

Class to compute validation loss and other metrics.

Attributes Summary

batch_weight_func	Get the function to compute the metric weight for each mini-batch.
data_flow	Get the validation data flow.
events	Get the event source object.
feed_dict	Get the fixed feed dict.
inputs	Get the input placeholders.
last_metrics_dict	Get the metric values from last evaluation.
loop	Get the training loop object.
metrics	Get the metrics to compute.
time_metric_name	Get the metric name for collecting evaluation time usage.

Methods Summary

run([feed_dict])	Run evaluation.

Attributes Documentation

batch_weight_func
Get the function to compute the metric weight for each mini-batch.

data_flow
Get the validation data flow.

Returns The validation data flow.

Return type *DataFlow*

events
Get the event source object.

Returns The event source object.

Return type *EventSource*

feed_dict
Get the fixed feed dict.

Returns The fixed feed dict.

Return type dict[tf.Tensor, any]

inputs
Get the input placeholders.

Returns The input placeholders.

Return type list[tf.Tensor]

last_metrics_dict
Get the metric values from last evaluation.

Returns The metric values dict.

Return type dict[str, any]

loop
Get the training loop object.

Returns The training loop object.

Return type *TrainLoop*

metrics
Get the metrics to compute.

Returns The metrics to compute.

Return type OrderedDict[str, tf.Tensor]

time_metric_name
Get the metric name for collecting evaluation time usage.

Methods Documentation

run(*feed_dict=None*)
Run evaluation.

Parameters **feed_dict** – The extra feed dict to be merged with the already configured dict. (default `None`)

VariationalChain

class tfsnippet.**VariationalChain**(*variational*, *model*, *log_joint=None*, *latent_names=None*, *latent_axis=None*)

Bases: `object`

Chain of the variational and model nets for variational inference.

In the context of variational inference, it is a common usage for chaining the variational net and the model net, by feeding the samples of latent variables from the variational net as the observations of the model net. *VariationalChain* holds the *BayesianNet* instances of the variational and the model nets, and the *VariationalInference* object for this chain.

See also:

`tfsnippet.bayes.BayesianNet.variational_chain()`

Attributes Summary

latent_axis	Get the axes of sampling dimensions of latent variables.
latent_names	Get the names of the latent variables for variational inference.
log_joint	Get the log-joint of the model.
model	Get the model net.
variational	Get the variational net.
vi	Get the variational inference object.

Attributes Documentation

latent_axis
Get the axes of sampling dimensions of latent variables.

latent_names
Get the names of the latent variables for variational inference.

Returns The names of the latent variables.

Return type tuple[str]

log_joint
Get the log-joint of the model.

Returns The log-joint of the model.

Return type tf.Tensor

model
Get the model net.

Returns The model net.

Return type *BayesianNet*

variational
Get the variational net.

Returns The variational net.

Return type *BayesianNet*

vi
Get the variational inference object.

Returns The variational inference object.

Return type *VariationalInference*

VariationalEvaluation

class tfsnippet.**VariationalEvaluation**(*vi*)

Bases: object

Factory for variational evaluation outputs.

Methods Summary

importance_sampling_log_likelihood([name])	Compute $logp(x)$ by importance sampling.
is_loglikelihood([name])	Short-cut for *importance_sampling_log_likelihood()*.

Methods Documentation

importance_sampling_log_likelihood(*name=None*)

Compute $logp(x)$ by importance sampling.

Returns The per-data $logp(x)$.

Return type tf.Tensor

See also:

`tfsnippet.variational.importance_sampling_log_likelihood()`

Parameters **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

is_loglikelihood(*name=None*)

Short-cut for *importance_sampling_log_likelihood()*.

VariationalInference

class tfsnippet.**VariationalInference**(*log_joint*, *latent_log_probs*, *axis=None*)

Bases: object

Class for variational inference.

Attributes Summary

axis	Get the axis or axes to be considered as the sampling dimensions of latent variables.
evaluation	Get the factory for evaluation outputs.
latent_log_prob	Get the summed log-density of latent variables.
latent_log_probs	Get the log-densities of latent variables.
log_joint	Get the log-joint of the model.
lower_bound	Get the factory for variational lower-bounds.
training	Get the factory for training objectives.

Attributes Documentation

axis
Get the axis or axes to be considered as the sampling dimensions of latent variables.

evaluation
Get the factory for evaluation outputs.

Returns The factory for evaluation outputs.

Return type *VariationalEvaluation*

latent_log_prob
Get the summed log-density of latent variables.

Returns The summed log-density of latent variables.

Return type tf.Tensor

latent_log_probs
Get the log-densities of latent variables.

Returns The log-densities of latent variables.

Return type tuple[tf.Tensor]

log_joint
Get the log-joint of the model.

Returns The log-joint of the model.

Return type tf.Tensor

lower_bound
Get the factory for variational lower-bounds.

Returns The factory for variational lower-bounds.

Return type *VariationalLowerBounds*

training
Get the factory for training objectives.

Returns The factory for training objectives.

Return type *VariationalTrainingObjectives*

VariationalLowerBounds

class tfsnippet.**VariationalLowerBounds**(*vi*)
Bases: object

Factory for variational lower-bounds.

Methods Summary

elbo([name])	Get the evidence lower-bound.
importance_weighted_objective([name])	Get the importance weighted lower-bound.
monte_carlo_objective([name])	Get the importance weighted lower-bound.

Methods Documentation

elbo(*name=None*)
Get the evidence lower-bound.

Returns The evidence lower-bound.

Return type tf.Tensor

See also:

`tfsnippet.variational.elbo_objective()`

Parameters **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

importance_weighted_objective(*name=None*)
Get the importance weighted lower-bound.

Returns The per-data importance weighted lower-bound.

Return type tf.Tensor

See also:

`tfsnippet.variational.monte_carlo_objective()`

Parameters **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

monte_carlo_objective(*name=None*)
Get the importance weighted lower-bound.

Returns The per-data importance weighted lower-bound.

Return type tf.Tensor

See also:

`tfsnippet.variational.monte_carlo_objective()`

Parameters **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

VariationalTrainingObjectives

class tfsnippet.**VariationalTrainingObjectives**(*vi*)
Bases: object

Factory for variational training objectives.

Methods Summary

iwae([name])	Get the SGVB training objective for importance weighted objective.
nvil([baseline, center_by_moving_average, ...])	Get the NVIL training objective.
reinforce([baseline, ...])	Get the NVIL training objective.

Continued on next page

Table 59 – continued from previous page

sgvb([name])	Get the SGVB training objective.
vimco([name])	Get the VIMCO training objective.

Methods Documentation

iwae(*name=None*)

Get the SGVB training objective for importance weighted objective.

Returns

The per-data SGVB training objective for importance weighted objective.

Return type tf.Tensor

See also:

`tfsnippet.variational.iwae_estimator()`

Parameters **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

nvil(*baseline=None*, *center_by_moving_average=True*, *decay=0.8*, *baseline_cost_weight=1.0*, *name=None*)

Get the NVIL training objective.

Parameters

- **baseline** – Values of the baseline function math:*C_{psi}(mathbf{x})* given input *x*. If this is not specified, then this method will degenerate to the REINFORCE algorithm, with only a moving average estimated constant baseline *c*.
- **center_by_moving_average** (*bool*) – Whether or not to use the moving average to maintain an estimation of *c* in above equations?
- **decay** – The decaying factor for moving average.
- **baseline_cost_weight** – Weight of the baseline cost.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The per-data NVIL training objective.

Return type tf.Tensor

reinforce(*baseline=None*, *center_by_moving_average=True*, *decay=0.8*, *baseline_cost_weight=1.0*, *name=None*)

Get the NVIL training objective.

Parameters

- **baseline** – Values of the baseline function math:*C_{psi}(mathbf{x})* given input *x*. If this is not specified, then this method will degenerate to the REINFORCE algorithm, with only a moving average estimated constant baseline *c*.
- **center_by_moving_average** (*bool*) – Whether or not to use the moving average to maintain an estimation of *c* in above equations?
- **decay** – The decaying factor for moving average.
- **baseline_cost_weight** – Weight of the baseline cost.

- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The per-data NVIL training objective.

Return type tf.Tensor

sgvb(*name=None*)

Get the SGVB training objective.

Returns

The per-data SGVB training objective. It is the negative of ELBO, which should directly be minimized.

Return type tf.Tensor

See also:

`tfsnippet.variational.sgvb_estimator()`

Parameters **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

vimco(*name=None*)

Get the VIMCO training objective.

Returns The per-data VIMCO training objective.

Return type tf.Tensor

See also:

`tfsnippet.variational.vimco_estimator()`

Parameters **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

BayesianNet

class tfsnippet.**BayesianNet**(*observed=None*)

Bases: object

Bayesian networks.

BayesianNet is a class which helps to construct Bayesian networks and to derive the variational lower-bounds. It is inspired by `zhusuan.BayesianNet`.

Due to the expressive limitations of TensorFlow, it is hard to build *BayesianNet* with the concept of *random variables*. Instead, we only collect *StochasticTensor* objects, i.e., tensors sampled from the distributions of these random variables. Thus *BayesianNet* is actually a collection of (multiple) ancestral samples from the random variables. Fortunately, we can approximate most interested statistics of the desired random variables with these samples, by using Monte Carlo methods. For example, obtaining the expectation of a random variable by averaging over multiple samples from it. The *StochasticTensor* objects are called *stochastic nodes* within the context of *BayesianNet*.

To build a Bayesian network, first obtain a *BayesianNet*:

```
net = tfsnippet.bayes.BayesianNet()
```

Then add stochastic nodes into the network:

A Bayesian Linear Regression example, as of `zhusuan.BayesianNet`:

$$w \sim N(0, \alpha^2 I)$$
$$y \sim N(w^T x, \beta^2)$$

```
from tfsnippet.bayes import BayesianNet()
from tfsnippet.distributions import Normal

def bayesian_linear_regression(x, alpha, beta, observed=None):
    net = BayesianNet(observed)
    w = net.add('w', Normal(mean=0., logstd=tf.log(alpha)))
    y_mean = tf.reduce_sum(tf.expand_dims(w, 0) * x, axis=1)
    y = net.add('y', Normal(mean=y_mean, logstd=tf.log(beta)))
    return net
```

To observe any stochastic nodes in the network, pass a dictionary mapping of `(name, Tensor)` as *observed* when constructing *BayesianNet*. For example:

```
model = bayesian_linear_regression(..., observed={'w': w_obs})
```

After construction, *BayesianNet* supports queries on the network.

```
# get samples of random variable y following generative process
# in the network
model.output('y')

# because w is observed in this case, its observed value will be
# returned
model.output('w')

# also multiple outputs can be fetched together
model.outputs(['y', 'w'])

# get local log probability values of w and y, which returns
# log p(w) and log p(y|w, x)
model.local_log_probs(['w', 'y'])

# query many quantities at the same time
model.query(['w', 'y'])
```

See also:

`zhusuan.BayesianNet`

Attributes Summary

observed	Get the read-only dict of observations.

Methods Summary

add(name, distribution[, n_samples, ...])	Add a stochastic node to the network.

Continued on next page

Table 61 – continued from previous page

chain(model_builder[, latent_names, ...])	Alias for *variational_chain()*.
get(name)	Get *StochasticTensor* of a stochastic node.
local_log_prob(name)	Get the log-density of a stochastic node.
local_log_probs(names)	Get the log-densities of stochastic nodes.
output(name)	Get the output of a stochastic node.
outputs(names)	Get the outputs of stochastic nodes.
query(names)	Get the outputs and log-densities of stochastic node(s).
variational_chain(model_builder[, ...])	Treat this *BayesianNet* as variational, and build the model net chained after this variational net.

Attributes Documentation

observed
Get the read-only dict of observations.

Returns The read-only observations dict.

Return type collections.Mapping[str, Tensor]

Methods Documentation

add(*name*, *distribution*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=None*)
Add a stochastic node to the network.

A *StochasticTensor* will be created for this node. If *name* exists in *observed* dict, its value will be used as the observation of this node. Otherwise samples will be taken from *distribution*.

Parameters

- **name** (*str*) – Name of the stochastic node.
- **distribution** (*Distribution or zhusuan.distributions.Distribution*) – Distribution where the samples should be taken from.
- **n_samples** (*int or tf.Tensor*) – Number of samples to take. If specified, *n_samples* will be taken, with a dedicated sampling dimension `[n_samples]` at the front. If not specified, just one sample will be taken, without the dedicated dimension.
- **group_ndims** (*int or tf.Tensor*) – Number of dimensions at the end of `[n_samples] + batch_shape` to be considered as events group. (default 0)
- **is_reparameterized** – If observation is not given for *name*, this argument will be used to determine whether or not re-parameterization trick should be applied when taking samples from *distribution* (if not specified, use *distribution.is_reparameterized*).

  If observation is given for *name*, and this argument is set to `True`, it will be used to validate the observation. If this argument is set to `False`, *tf.stop_gradient* will be applied on the observation.

Returns The sampled stochastic tensor.

Return type *StochasticTensor*

Raises

- `TypeError` – If *name* is not a str, or *distribution* is a `TransformedDistribution`.
- `KeyError` – If *StochasticTensor* with *name* already exists.

- `ValueError` – If *transform* cannot be applied, or *is_reparameterized* = *True*, but the observation is not re-parameterized.

See also:

`tfsnippet.distributions.Distribution.sample()`

chain(*model_builder*, *latent_names=None*, *latent_axis=None*, *observed=None*, ***kwargs*)
Alias for *variational_chain()*.

get(*name*)
Get *StochasticTensor* of a stochastic node.

Parameters **name** (*str*) – Name of the queried stochastic node.

Returns

***StochasticTensor* of the queried node,** or `None` if no node exists with *name*.

Return type *StochasticTensor*

local_log_prob(*name*)
Get the log-density of a stochastic node.

Parameters **name** (*str*) – Name of the queried stochastic node.

Returns Log-density of the queried stochastic node.

Return type tf.Tensor

Raises `KeyError` – If non-exist name is queried.

local_log_probs(*names*)
Get the log-densities of stochastic nodes.

Parameters **names** (*Iterable[str]*) – Names of the queried stochastic nodes.

Returns Log-densities of the queried stochastic nodes.

Return type list[tf.Tensor]

Raises `KeyError` – If non-exist name is queried.

output(*name*)
Get the output of a stochastic node. The output of a stochastic node is its *StochasticTensor.tensor*.

Parameters **name** (*str*) – Name of the queried stochastic node.

Returns Output of the queried stochastic node.

Return type tf.Tensor

Raises `KeyError` – If non-exist name is queried.

outputs(*names*)
Get the outputs of stochastic nodes. The output of a stochastic node is its *StochasticTensor.tensor*.

Parameters **names** (*Iterable[str]*) – Names of the queried stochastic nodes.

Returns Outputs of the queried stochastic nodes.

Return type list[tf.Tensor]

Raises `KeyError` – If non-exist name is queried.

query(*names*)
Get the outputs and log-densities of stochastic node(s).

Parameters **names** (*Iterable[str]*) – Names of the queried stochastic nodes.

Returns

Tuples of *(output, log-prob)* of the queried stochastic nodes.

Return type list[(tf.Tensor, tf.Tensor)]

Raises KeyError – If non-exist name is queried.

variational_chain(*model_builder*, *latent_names=None*, *latent_axis=None*, *observed=None*, ***kwargs*)
Treat this *BayesianNet* as variational, and build the model net chained after this variational net.

Parameters

- **model_builder** – Function which receives the *observed* dict, and produce the model *BayesianNet* or a tuple of the model *BayesianNet* and the log-joint of the model.
- **latent_names** (*Iterable[str]*) – Names of the nodes to be considered as latent variables in this *BayesianNet*. All these variables will be fed into *model_builder* as observed variables, overriding the observations in *observed*. (default all the variables in this *BayesianNet*)
- **latent_axis** – The axis or axes to be considered as the sampling dimensions of latent variables. The specified axes will be summed up in the variational lower-bounds or training objectives. (default None)
- **observed** – Dict of `(name, observation)` fed into *model_builder*. (default None)
- ****kwargs** – Additional named arguments passed to *model_builder*.

Returns

The object that holds this *BayesianNet* as the *variational* net, the constructed *model* net, and the VariationalInference object for obtaining the variational lower-bounds and training objectives.

Return type tfsnippet.variational.VariationalChain

See also:

`tfsnippet.variational.VariationalChain`

DataFlow

class tfsnippet.**DataFlow**
Bases: object

Data flows are objects for constructing mini-batch iterators.

There are two major types of *DataFlow* classes: data sources and data transformers. Data sources, like the ArrayFlow, produce mini-batches from underlying data sources. Data transformers, like MapperFlow, produce mini-batches by transforming arrays from the source.

All *DataFlow* subclasses shipped by *tfsnippet.dataflows* can be constructed by factory methods of this base class. For example:

```
# :class:`ArrayFlow` from arrays
array_flow = DataFlow.arrays([x, y], batch_size=256, shuffle=True)

# :class:`MapperFlow` by adding the two arrays from `array_flow`
mapper_flow = array_flow.map(lambda x, y: (x + y,))
```

Attributes Summary

current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.
select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

current_batch

Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)

- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a `ValueError` will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- `ValueError` – If not even one data flow is specified.
- `TypeError` – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises `ValueError` – If this data-flow is empty.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises StopIteration – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)
Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.
- **stop** – The ending number of the sequence.
- **step** – The step of the sequence. (default 1)
- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **dtype** – Data type of the numbers. (default np.int32)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)
Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

DataMapper

class tfsnippet.**DataMapper**
Bases: object

Base class for all data mappers.

A *DataMapper* is a callable object, which maps input arrays into outputs arrays. Instances of *DataMapper* are usually used as the mapper of a *tfsnippet.dataflows.MapperFlow*.

Methods Summary

__call__(*arrays)	Transform the input arrays into outputs.

Methods Documentation

__call__(**arrays*)
Transform the input arrays into outputs.

Parameters ***arrays** – Arrays to be transformed.

Returns The output arrays.

Return type tuple[np.ndarray]

SlidingWindow

class tfsnippet.**SlidingWindow**(*data_array*, *window_size*)
Bases: tfsnippet.dataflows.data_mappers.DataMapper

DataMapper for producing sliding windows according to indices.

Usage:

```
data = np.arange(1000)
sw = SlidingWindow(data, window_size=100)

# construct a DataFlow from this SlidingWindow
sw_flow = sw.as_flow(batch_size=64)
# or equivalently
sw_flow = DataFlow.seq(
    0, len(data) - sw.window_size + 1, batch_size=64).map(sw)
```

Attributes Summary

data_array	Get the data array.
window_size	Get the window size.

Methods Summary

__call__(*arrays)	Transform the input arrays into outputs.
as_flow(batch_size[, shuffle, skip_incomplete])	Get a *DataFlow* which iterates through mini-batches of sliding windows upon `data_array`.

Attributes Documentation

data_array
Get the data array.

window_size
Get the window size.

Methods Documentation

__call__ (**arrays*)
Transform the input arrays into outputs.

Parameters ***arrays** – Arrays to be transformed.

Returns The output arrays.

Return type tuple[np.ndarray]

as_flow (*batch_size*, *shuffle=False*, *skip_incomplete=False*)
Get a *DataFlow* which iterates through mini-batches of sliding windows upon `data_array`.

Parameters

- **batch_size** (*int*) – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)

Returns The data flow for sliding windows.

Return type *DataFlow*

Config

class tfsnippet.**Config**

Bases: object

Base class for defining config values.

Derive sub-classes of *Config*, and define config values as public class attributes. For example:

```
class YourConfig(Config):
    max_epoch = 100
    learning_rate = 0.01
    activation = ConfigField(str, default='leaky_relu',
                             choices=['sigmoid', 'relu', 'leaky_relu'])
    l2_regularization = ConfigField(float, default=None)

config = YourConfig()
```

When an attribute is assigned, it will be validated by:

1. If the attribute is defined as a *ConfigField*, then its `validate()` will be used to validate the value.
2. If the attribute is not `None`, and a validator is registered for *type(value)*, then an instance of this type of validator will be used to validate the value.
3. Otherwise if the attribute is not defined, or is `None`, then no validation will be performed.

For example:

```
config.l2_regularization = 5e-4  # okay
config.l2_regularization = 'xxx'  # raise an error
config.activation = 'sigmoid'  # okay
config.activation = 'tanh'  # raise an error
config.new_attribute = 'yyy'  # okay
```

The config object also implements dict-like interface:

```
# to test whether a key exists
print(key in config)

# to iterate through all config values
for key in config:
    print(key, config[key])

# to set a config value
config[key] = value
```

You may get all the config values of a config object as dict:

```
print(config_to_dict(config))
```

Or you may get the default values of the config class as dict:

```
print(config_defaults(YourConfig))
print(config_defaults(config))  # same as the above line
```

Methods Summary

to_dict()	Get the config values as a dict.
update(key_values)	Update the config values from *key_values*.

Methods Documentation

to_dict()
Get the config values as a dict.

Returns The config values.

Return type dict[str, any]

update(*key_values*)
Update the config values from *key_values*.

Parameters **key_values** – A dict, or a sequence of (key, value) pairs.

ConfigField

class tfsnippet.**ConfigField**(*type*, *default=None*, *description=None*, *nullable=False*, *choices=None*)

Bases: object

A config field.

Attributes Summary

choices	Get the valid choices of the config value.
default_value	Get the default config value.
description	Get the config description.
nullable	Whether or not None is a valid value?
type	Get the value type.

Methods Summary

validate(value[, strict])	Validate the config *value*.

Attributes Documentation

choices
Get the valid choices of the config value.

default_value
Get the default config value.

description
Get the config description.

nullable
Whether or not None is a valid value?

type
Get the value type.

Methods Documentation

validate(*value*, *strict=False*)
Validate the config *value*.

Parameters

- **value** – The value to be validated.
- **strict** (*bool*) – If `True`, disable type conversion. If `False`, the validator will try its best to convert the input *value* into desired type.

Returns The validated value.

GraphKeys

class tfsnippet.**GraphKeys**
Bases: `object`

Defines TensorFlow graph collection keys for TFSnippet.

Attributes Summary

AUTO_HISTOGRAM	

Attributes Documentation

AUTO_HISTOGRAM = 'TFSNIPPET_AUTO_HISTOGRAM'

InvertibleMatrix

class tfsnippet.**InvertibleMatrix**(*size*, *strict=False*, *dtype=tf.float32*, *epsilon=1e-06*, *trainable=True*, *random_state=None*, *name=None*, *scope=None*)
Bases: `tfsnippet.utils.reuse.VarScopeObject`

A matrix initialized to be an invertible, orthogonal matrix.

If *strict* is `False`, then there is no guarantee that the matrix will keep invertible during training. Otherwise, the matrix will be derived through a variant of PLU decomposition as proposed in (Kingma & Dhariwal, 2018):

$$\mathbf{M} = \mathbf{PL}(\mathbf{U} + \mathrm{diag}(\mathbf{sign} \odot \exp(\mathbf{s})))$$

where *P* is a permutation matrix, *L* is a lower triangular matrix with all its diagonal elements equal to one, *U* is an upper triangular matrix with all its diagonal elements equal to zero, *sign* is a vector of *{-1, 1}*, and *s* is a vector. *P* and *sign* are fixed variables, while *L*, *U*, *s* are trainable variables.

A *random_state* can be specified via the constructor. If it is not specified, an instance of `VarScopeRandomState` will be created according to the variable scope name of the object.

Attributes Summary

inv_matrix	Get the inverted matrix.

Continued on next page

Table 71 – continued from previous page

log_det	Get the log-determinant of the matrix.
matrix	Get the matrix tensor.
name	Get the name of this object.
shape	Get the shape of the matrix.
variable_scope	Get the variable scope of this object.

Attributes Documentation

inv_matrix
Get the inverted matrix.

Returns The inverted matrix tensor.

Return type tf.Tensor

log_det
Get the log-determinant of the matrix.

Returns The log-determinant tensor.

Return type tf.Tensor

matrix
Get the matrix tensor.

Returns The matrix tensor.

Return type tf.Tensor or tf.Variable

name
Get the name of this object.

shape
Get the shape of the matrix.

Returns The shape of the matrix.

Return type (int, int)

variable_scope
Get the variable scope of this object.

VarScopeObject

class tfsnippet.**VarScopeObject**(*name=None*, *scope=None*)
Bases: object

Base class for objects that own a variable scope.

The *VarScopeObject* can be used along with *instance_reuse()*, for example:

```
class YourVarScopeObject(VarScopeObject):

    @instance_reuse
    def foo(self):
        return tf.get_variable('bar', ...)

o = YourVarScopeObject('object_name')
o.foo()  # You should get a variable with name "object_name/foo/bar"
```

To build variables in the constructor of derived classes, you may use `reopen_variable_scope(self.variable_scope)` to open the original variable scope and its name scope, right after the constructor of *VarScopeObject* has been called, for example:

```
class YourVarScopeObject(VarScopeObject):

    def __init__(self, name=None, scope=None):
        super(YourVarScopeObject, self).__init__(name=name, scope=scope)
        with reopen_variable_scope(self.variable_scope):
            self.w = tf.get_variable('w', ...)
```

See also:

tfsnippet.utils.instance_reuse().

Attributes Summary

name	Get the name of this object.
variable_scope	Get the variable scope of this object.

Attributes Documentation

name
Get the name of this object.

variable_scope
Get the variable scope of this object.

SummaryCollector

class tfsnippet.**SummaryCollector**(*collections=None*, *no_add_to_collections=False*)
Bases: object

Collecting summaries and histograms added by *tfsnippet.add_summary()* and *tfsnippet.add_histogram()*. For example:

```
collector = SummaryCollector()
with collector.as_default():
    spt.add_summary(...)
summary_op = collector.merge_summary()
```

You may also use this collector to capture the auto histogram generated by layers from *tfsnippet.layers*, for example:

```
collector = SummaryCollector()
with collector.as_default(auto_histogram=True):
    y = spt.layers.dense(x, ...)
summary_op = collector.merge_summary()
```

Attributes Summary

collections	Get the summary collections.
summary_list	Get the list of captured summaries.

Methods Summary

add_histogram(tensor[, summary_name, ...])	Add the histogram of *tensor* to this collector and to *collections*.
add_summary(summary[, collections])	Add the summary to this collector and to *collections*.
as_default(**kwds)	Push this *SummaryCollector* to the top of context stack, and enter a scoped context.
merge_summary()	Merge all the captured summaries into one operation.

Attributes Documentation

collections
Get the summary collections.

summary_list
Get the list of captured summaries.

Methods Documentation

add_histogram(*tensor*, *summary_name=None*, *strip_scope=False*, *collections=None*, *name=None*)
Add the histogram of *tensor* to this collector and to *collections*.

Parameters

- **tensor** – Take histogram of this tensor.
- **summary_name** – Specify the summary name for *tensor*.
- **strip_scope** – If True, strip the name scope from *tensor.name* when adding the histogram.
- **collections** – Also add the histogram to these collections. Defaults to *self.collections*.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The serialized histogram tensor of *tensor*.

add_summary(*summary*, *collections=None*)
Add the summary to this collector and to *collections*.

Parameters

- **summary** – TensorFlow summary tensor.
- **collections** – Also add the summary to these collections. Defaults to *self.collections*.

Returns The *summary* tensor.

as_default(***kwds*)
Push this *SummaryCollector* to the top of context stack, and enter a scoped context.

Parameters **auto_histogram** (*bool*) – If specified, set the config value of *tfsnippet.settings.auto_histogram* within the context.

Yields This summary collector object.

merge_summary()
Merge all the captured summaries into one operation.

Returns

The merged operation, or None if no summary has been captured.

Return type tf.Operation or None

StochasticTensor

class tfsnippet.**StochasticTensor**(*distribution*, *tensor*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=None*, *flow_origin=None*, *log_prob=None*)
Bases: tfsnippet.utils.tensor_wrapper.TensorWrapper

Samples or observations of a stochastic variable.

StochasticTensor is a tensor-like object, carrying samples or observations of a random variable, following some *distribution* of a specific *Distribution* type.

It mimics the interface of zhusuan.model.StochasticTensor, except that it does not carry a *name*, and does not add itself to any *BayesianNet* context automatically.

Attributes Summary

distribution	Get the *Distribution* of this *StochasticTensor*.
flow_origin	Get the original stochastic tensor from the base distribution of a *tfsnippet.FlowDistribution*.
group_ndims	Get the number of dimensions to be considered as events group.
is_continuous	Whether or not this *StochasticTensor* is continuous?
is_reparameterized	Whether or not this *StochasticTensor* is reparameterized?
n_samples	Get the number of samples taken in *Distribution.sample*.
tensor	Get the samples or observations *tensor*.

Methods Summary

log_prob([group_ndims, name])	Compute the log-densities of this *StochasticTensor*.
prob([group_ndims, name])	Compute the densities of this *StochasticTensor*.

Attributes Documentation

distribution
Get the *Distribution* of this *StochasticTensor*.

Returns The distribution instance.

Return type *Distribution*

flow_origin
Get the original stochastic tensor from the base distribution of a *tfsnippet.FlowDistribution*.

Returns

The original stochastic tensor, or None if there is no original stochastic tensor.

Return type *StochasticTensor* or None

group_ndims
Get the number of dimensions to be considered as events group.

Returns The configured *group_ndims*.

Return type int or tf.Tensor

is_continuous
Whether or not this *StochasticTensor* is continuous?

Returns Equivalent to `self.distribution.is_continuous`.

Return type bool

is_reparameterized
Whether or not this *StochasticTensor* is re-parameterized?

Returns A boolean indicating whether it is re-parameterized.

Return type bool

n_samples
Get the number of samples taken in *Distribution.sample*.

Returns The number of samples.

Return type int or tf.Tensor or None

tensor
Get the samples or observations *tensor*.

Returns The *tensor* specified at construction.

Return type tf.Tensor

Methods Documentation

log_prob(*group_ndims=None*, *name=None*)
Compute the log-densities of this *StochasticTensor*.

Parameters

- **group_ndims** (*int or tf.Tensor*) – If specified, overriding the configured *group_ndims*.
- **name** – TensorFlow name scope of the graph nodes.

Returns The log-densities.

Return type tf.Tensor

prob(*group_ndims=None*, *name=None*)
Compute the densities of this *StochasticTensor*.

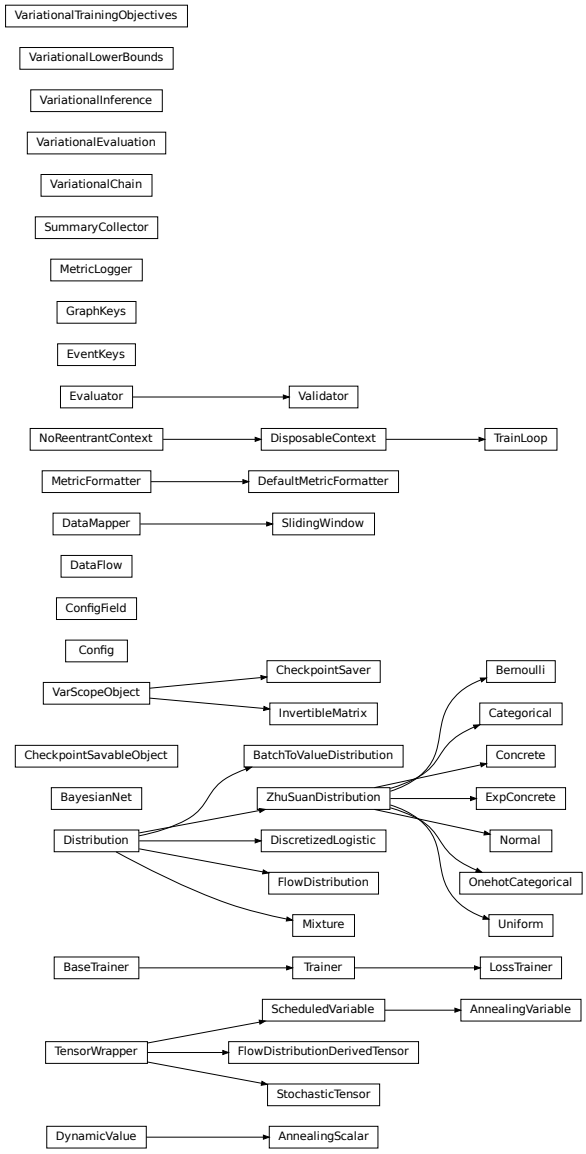
Parameters

- **group_ndims** (*int or tf.Tensor*) – If specified, overriding the configured *group_ndims*.
- **name** – TensorFlow name scope of the graph nodes.

Returns The densities.

Return type tf.Tensor

Class Inheritance Diagram



2.1.2 tfsnippet.dataflows

tfsnippet.dataflows Package

Classes

ArrayFlow(arrays, batch_size[, shuffle, ...])	Using numpy-like arrays as data source flow.
DataFlow	Data flows are objects for constructing mini-batch iterators.
DataMapper	Base class for all data mappers.
ExtraInfoDataFlow(array_count, data_length, ...)	Base class for *DataFlow* subclasses with auxiliary information about the mini-batches.
GatherFlow(flows)	Gathering multiple data flows into a single flow.
IteratorFactoryFlow(factory)	Data flow constructed from an iterator factory.
MapperFlow(source, mapper[, array_indices])	Data flow which transforms the mini-batch arrays from source flow by a specified mapper function.
SeqFlow(start, stop[, step, batch_size, ...])	Using number sequence as data source flow.
SlidingWindow(data_array, window_size)	*DataMapper* for producing sliding windows according to indices.
ThreadingFlow(source, prefetch)	Data flow to prefetch from the source data flow in a background thread.

ArrayFlow

class tfsnippet.dataflows.**ArrayFlow**(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Bases: tfsnippet.dataflows.base.ExtraInfoDataFlow

Using numpy-like arrays as data source flow.

Usage:

```
array_flow = DataFlow.arrays([x, y], batch_size=256, shuffle=True,
                             skip_incomplete=True)
for batch_x, batch_y in array_flow:
    ...
```

Attributes Summary

array_count	Get the count of arrays in each mini-batch.
batch_size	Get the size of each mini-batch.
current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).
data_length	Get the total length of the data.
data_shapes	Get the shapes of the data in each mini-batch.
is_shuffled	Whether or not the data are first shuffled before iterated through mini-batches?

Continued on next page

Table 78 – continued from previous page

skip_incomplete	Whether or not to exclude the last mini-batch if it is incomplete?
the_arrays	Get the tuple of arrays accessed by this *ArrayFlow*.

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.
select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

array_count
Get the count of arrays in each mini-batch.

Returns The count of arrays in each mini-batch.

Return type int

batch_size
Get the size of each mini-batch.

Returns The size of each mini-batch.

Return type int

current_batch
Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

data_length
Get the total length of the data.

Returns The total length of the data.

Return type int

data_shapes
Get the shapes of the data in each mini-batch.

Returns

The shapes of data in a mini-batch. The batch dimension is not included.

Return type tuple[tuple[int]]

is_shuffled
Whether or not the data are first shuffled before iterated through mini-batches?

skip_incomplete
Whether or not to exclude the last mini-batch if it is incomplete?

the_arrays
Get the tuple of arrays accessed by this *ArrayFlow*.

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a ValueError will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- ValueError – If not even one data flow is specified.
- TypeError – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises ValueError – If this data-flow is empty.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises StopIteration – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)
Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.
- **stop** – The ending number of the sequence.
- **step** – The step of the sequence. (default 1)
- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default False)

- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **dtype** – Data type of the numbers. (default `np.int32`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)

Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

DataFlow

class tfsnippet.dataflows.**DataFlow**

Bases: `object`

Data flows are objects for constructing mini-batch iterators.

There are two major types of *DataFlow* classes: data sources and data transformers. Data sources, like the *ArrayFlow*, produce mini-batches from underlying data sources. Data transformers, like *MapperFlow*, produce mini-batches by transforming arrays from the source.

All *DataFlow* subclasses shipped by *tfsnippet.dataflows* can be constructed by factory methods of this base class. For example:

```
# :class:`ArrayFlow` from arrays
array_flow = DataFlow.arrays([x, y], batch_size=256, shuffle=True)
```

(continues on next page)

(continued from previous page)

```
# :class:`MapperFlow` by adding the two arrays from `array_flow`
mapper_flow = array_flow.map(lambda x, y: (x + y,))
```

Attributes Summary

current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.
select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

current_batch

Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)

- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a ValueError will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- ValueError – If not even one data flow is specified.
- TypeError – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises ValueError – If this data-flow is empty.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises `StopIteration` – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)

Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)

Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.
- **stop** – The ending number of the sequence.
- **step** – The step of the sequence. (default `1`)
- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **dtype** – Data type of the numbers. (default `np.int32`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)

Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

DataMapper

class tfsnippet.dataflows.**DataMapper**

Bases: `object`

Base class for all data mappers.

A *DataMapper* is a callable object, which maps input arrays into outputs arrays. Instances of *DataMapper* are usually used as the `mapper` of a *tfsnippet.dataflows.MapperFlow*.

Methods Summary

__call__(*arrays)	Transform the input arrays into outputs.

Methods Documentation

__call__(**arrays*)

Transform the input arrays into outputs.

Parameters ***arrays** – Arrays to be transformed.

Returns The output arrays.

Return type tuple[np.ndarray]

ExtraInfoDataFlow

class tfsnippet.dataflows.**ExtraInfoDataFlow**(*array_count*, *data_length*, *data_shapes*, *batch_size*, *skip_incomplete*, *is_shuffled*)

Bases: `tfsnippet.dataflows.base.DataFlow`

Base class for *DataFlow* subclasses with auxiliary information about the mini-batches.

Attributes Summary

array_count	Get the count of arrays in each mini-batch.
batch_size	Get the size of each mini-batch.

Continued on next page

Table 83 – continued from previous page

current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).
data_length	Get the total length of the data.
data_shapes	Get the shapes of the data in each mini-batch.
is_shuffled	Whether or not the data are first shuffled before iterated through mini-batches?
skip_incomplete	Whether or not to exclude the last mini-batch if it is incomplete?

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.
select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

array_count
Get the count of arrays in each mini-batch.

Returns The count of arrays in each mini-batch.

Return type int

batch_size
Get the size of each mini-batch.

Returns The size of each mini-batch.

Return type int

current_batch
Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

data_length
Get the total length of the data.

Returns The total length of the data.

Return type int

data_shapes
Get the shapes of the data in each mini-batch.

Returns

The shapes of data in a mini-batch. The batch dimension is not included.

Return type tuple[tuple[int]]

is_shuffled
Whether or not the data are first shuffled before iterated through mini-batches?

skip_incomplete
Whether or not to exclude the last mini-batch if it is incomplete?

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a ValueError will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- ValueError – If not even one data flow is specified.
- TypeError – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises ValueError – If this data-flow is empty.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises StopIteration – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)
Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.
- **stop** – The ending number of the sequence.
- **step** – The step of the sequence. (default 1)

- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **dtype** – Data type of the numbers. (default `np.int32`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)

Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

GatherFlow

class tfsnippet.dataflows.**GatherFlow**(*flows*)

Bases: `tfsnippet.dataflows.base.DataFlow`

Gathering multiple data flows into a single flow.

Usage:

```
x_flow = DataFlow.arrays([x], batch_size=256)
y_flow = DataFlow.arrays([y], batch_size=256)
xy_flow = DataFlow.gather([x_flow, y_flow])
```

Attributes Summary

current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).
flows	Get the data flows to be gathered.

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.
select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

current_batch

Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

flows

Get the data flows to be gathered.

Returns The data flows to be gathered.

Return type tuple[*DataFlow*]

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)

- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a `ValueError` will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- `ValueError` – If not even one data flow is specified.
- `TypeError` – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises `ValueError` – If this data-flow is empty.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises StopIteration – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)
Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.
- **stop** – The ending number of the sequence.
- **step** – The step of the sequence. (default 1)
- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **dtype** – Data type of the numbers. (default np.int32)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)
Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

IteratorFactoryFlow

class tfsnippet.dataflows.**IteratorFactoryFlow**(*factory*)
Bases: tfsnippet.dataflows.base.DataFlow

Data flow constructed from an iterator factory.

Usage:

```
x_flow = DataFlow.arrays([x], batch_size=256)
y_flow = DataFlow.arrays([y], batch_size=256)
xy_flow = DataFlow.iterator_factory(lambda: (
    (x, y) for (x,), (y,) in zip(x_flow, y_flow)
))
```

Attributes Summary

current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a `StopIteration`).

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.

Continued on next page

Table 88 – continued from previous page

select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

current_batch
Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a ValueError will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- ValueError – If not even one data flow is specified.
- TypeError – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises ValueError – If this data-flow is empty.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises StopIteration – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)
Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.
- **stop** – The ending number of the sequence.

- **step** – The step of the sequence. (default `1`)
- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **dtype** – Data type of the numbers. (default `np.int32`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)

Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

MapperFlow

class tfsnippet.dataflows.**MapperFlow**(*source*, *mapper*, *array_indices=None*)

Bases: tfsnippet.dataflows.base.DataFlow

Data flow which transforms the mini-batch arrays from source flow by a specified mapper function.

Usage:

```
source_flow = Data.arrays([x, y], batch_size=256)
mapper_flow = source_flow.map(lambda x, y: (x + y,))
```

Attributes Summary

array_indices	Get the indices of the arrays to be processed.
current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).
source	Get the source data flow.

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.
select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

array_indices
Get the indices of the arrays to be processed.

current_batch
Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

source
Get the source data flow.

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)

- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a `ValueError` will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- `ValueError` – If not even one data flow is specified.
- `TypeError` – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises `ValueError` – If this data-flow is empty.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises StopIteration – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)
Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.
- **stop** – The ending number of the sequence.
- **step** – The step of the sequence. (default 1)
- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **dtype** – Data type of the numbers. (default np.int32)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)
Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

SeqFlow

class tfsnippet.dataflows.**SeqFlow**(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Bases: `tfsnippet.dataflows.array_flow.ArrayFlow`

Using number sequence as data source flow.

This *SeqFlow* is particularly used for generating the *seed* number indices, then fetch the actual data by *MapperFlow* according to the seed numbers.

Usage:

```
seq_flow = DataFlow.seq(0, len(x), batch_size=256)
mapper_flow = seq_flow.map(lambda idx: np.stack(
    [fetch_data_by_index(i) for i in idx]
))
```

Attributes Summary

array_count	Get the count of arrays in each mini-batch.
batch_size	Get the size of each mini-batch.
current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a `StopIteration`).
data_length	Get the total length of the data.
data_shapes	Get the shapes of the data in each mini-batch.
is_shuffled	Whether or not the data are first shuffled before iterated through mini-batches?
skip_incomplete	Whether or not to exclude the last mini-batch if it is incomplete?
start	Get the starting number of the sequence.

Continued on next page

Table 91 – continued from previous page

step	Get the step of the sequence.
stop	Get the ending number of the sequence.
the_arrays	Get the tuple of arrays accessed by this *ArrayFlow*.

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.
select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

array_count
Get the count of arrays in each mini-batch.

Returns The count of arrays in each mini-batch.

Return type int

batch_size
Get the size of each mini-batch.

Returns The size of each mini-batch.

Return type int

current_batch
Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

data_length
Get the total length of the data.

Returns The total length of the data.

Return type int

data_shapes
Get the shapes of the data in each mini-batch.

Returns

The shapes of data in a mini-batch. The batch dimension is not included.

Return type tuple[tuple[int]]

is_shuffled
Whether or not the data are first shuffled before iterated through mini-batches?

skip_incomplete
Whether or not to exclude the last mini-batch if it is incomplete?

start
Get the starting number of the sequence.

step
Get the step of the sequence.

stop
Get the ending number of the sequence.

the_arrays
Get the tuple of arrays accessed by this *ArrayFlow*.

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a ValueError will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- ValueError – If not even one data flow is specified.
- TypeError – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises `ValueError` – If this data-flow is empty.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises `StopIteration` – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)
Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.

- **stop** – The ending number of the sequence.
- **step** – The step of the sequence. (default `1`)
- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **dtype** – Data type of the numbers. (default `np.int32`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)

Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

SlidingWindow

class tfsnippet.dataflows.**SlidingWindow**(*data_array*, *window_size*)

Bases: tfsnippet.dataflows.data_mappers.DataMapper

DataMapper for producing sliding windows according to indices.

Usage:

```
data = np.arange(1000)
sw = SlidingWindow(data, window_size=100)

# construct a DataFlow from this SlidingWindow
sw_flow = sw.as_flow(batch_size=64)
# or equivalently
sw_flow = DataFlow.seq(
    0, len(data) - sw.window_size + 1, batch_size=64).map(sw)
```

Attributes Summary

data_array	Get the data array.
window_size	Get the window size.

Methods Summary

__call__(*arrays)	Transform the input arrays into outputs.
as_flow(batch_size[, shuffle, skip_incomplete])	Get a *DataFlow* which iterates through mini-batches of sliding windows upon `data_array`.

Attributes Documentation

data_array
Get the data array.

window_size
Get the window size.

Methods Documentation

__call__ (**arrays*)
Transform the input arrays into outputs.

Parameters ***arrays** – Arrays to be transformed.

Returns The output arrays.

Return type tuple[np.ndarray]

as_flow (*batch_size*, *shuffle=False*, *skip_incomplete=False*)
Get a *DataFlow* which iterates through mini-batches of sliding windows upon `data_array`.

Parameters

- **batch_size** (*int*) – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)

Returns The data flow for sliding windows.

Return type *DataFlow*

ThreadingFlow

class tfsnippet.dataflows.**ThreadingFlow**(*source*, *prefetch*)

Bases: tfsnippet.dataflows.base.DataFlow, tfsnippet.utils.concepts.AutoInitAndCloseable

Data flow to prefetch from the source data flow in a background thread.

Usage:

```
array_flow = DataFlow.arrays([x, y], batch_size=256)
with array_flow.threaded(prefetch=5) as df:
    for epoch in epochs:
        for batch_x, batch_y in df:
            ...
```

Attributes Summary

EPOCH_END	Object to mark an ending position of an epoch.
current_batch	Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).
prefetch_num	Get the number of batches to prefetch.
source	Get the source data flow.

Methods Summary

arrays(arrays, batch_size[, shuffle, ...])	Construct an *ArrayFlow*.
close()	Ensure the internal states are destroyed.
gather(flows)	Gather multiple data flows into a single flow.
get_arrays()	Iterate through the data-flow, collecting mini-batches into arrays.
init()	Ensure the internal states are initialized.
iterator_factory(factory)	Construct a *IteratorFactoryFlow*.
map(mapper[, array_indices])	Construct a *MapperFlow*.
next_batch()	Get the arrays of next mini-batch from the implicit iterator.
select(indices)	Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch.
seq(start, stop[, step, batch_size, ...])	Construct a *SeqFlow*.
threaded(prefetch)	Construct a *ThreadingFlow* from this flow.
to_arrays_flow(batch_size[, shuffle, ...])	Convert this data-flow to a *ArrayFlow*.

Attributes Documentation

EPOCH_END = <object object>

Object to mark an ending position of an epoch.

current_batch

Get the the result of current batch (last call to *next_batch()*, if the implicit iterator has been opened and the last call to *next_batch()* does not raise a StopIteration).

Returns The arrays of current batch.

Return type tuple[np.ndarray] or None

prefetch_num
Get the number of batches to prefetch.

source
Get the source data flow.

Methods Documentation

static arrays(*arrays*, *batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)
Construct an *ArrayFlow*.

Parameters

- **arrays** – List of numpy-like arrays, to be iterated through mini-batches. These arrays should be at least 1-d, with identical first dimension.
- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default False)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default False)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default None, construct a new RandomState).

Returns The data flow from arrays.

Return type tfsnippet.dataflow.ArrayFlow

close()
Ensure the internal states are destroyed.

static gather(*flows*)
Gather multiple data flows into a single flow.

Parameters **flows** (*Iterable[DataFlow]*) – The data flows to gather. At least one data flow should be specified, otherwise a ValueError will be raised.

Returns The gathered data flow.

Return type tfsnippet.dataflow.GatherFlow

Raises

- ValueError – If not even one data flow is specified.
- TypeError – If a specified flow is not a *DataFlow*.

get_arrays()
Iterate through the data-flow, collecting mini-batches into arrays.

Returns The collected arrays.

Return type tuple[np.ndarray]

Raises ValueError – If this data-flow is empty.

init()
Ensure the internal states are initialized.

static iterator_factory(*factory*)
Construct a *IteratorFactoryFlow*.

Parameters **factory** (*() -> Iterator or Iterable*) – A factory method for constructing the mini-batch iterators for each epoch.

Returns The data flow.

Return type tfsnippet.dataflow.IteratorFactoryFlow

map(*mapper*, *array_indices=None*)
Construct a *MapperFlow*.

Parameters

- **mapper** (*(*np.ndarray) -> tuple[np.ndarray]*) – The mapper function, which transforms numpy arrays into a tuple of other numpy arrays.
- **array_indices** (*int or Iterable[int]*) – The indices of the arrays to be processed within a mini-batch.

  If specified, will apply the mapper only on these selected arrays. This will require the mapper to produce exactly the same number of output arrays as the inputs.

  If not specified, apply the mapper on all arrays, and do not require the number of output arrays to match the inputs.

Returns The data flow with *mapper* applied.

Return type tfsnippet.dataflow.MapperFlow

next_batch()
Get the arrays of next mini-batch from the implicit iterator.

Returns The arrays of mini-batch.

Return type tuple[np.ndarray]

Raises StopIteration – If the implicit iterator is exhausted. Note that this error will only be triggered once at the end of an epoch. The next time calling this method, a new epoch will be opened.

select(*indices*)
Construct a *DataFlow*, which selects and rearranges arrays in each mini-batch. For example:

```
flow = DataFlow.arrays([x, y, z], batch_size=64)
flow.select([0, 2, 0])  # selects ``(x, z, x)`` in each mini-batch
```

Parameters **indices** (*Iterable[int]*) – The indices of arrays to select.

Returns The data flow with selected arrays in each mini-batch.

Return type *DataFlow*

static seq(*start*, *stop*, *step=1*, *batch_size=None*, *shuffle=False*, *skip_incomplete=False*, *dtype=<type 'numpy.int32'>*, *random_state=None*)
Construct a *SeqFlow*.

Parameters

- **start** – The starting number of the sequence.
- **stop** – The ending number of the sequence.
- **step** – The step of the sequence. (default 1)

- **batch_size** – Batch size of the data flow. Required.
- **shuffle** (*bool*) – Whether or not to shuffle the numbers before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **dtype** – Data type of the numbers. (default `np.int32`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The data flow from number sequence.

Return type tfsnippet.dataflow.SeqFlow

threaded(*prefetch*)

Construct a *ThreadingFlow* from this flow.

Parameters **prefetch** (*int*) – Number of mini-batches to prefetch ahead. It should be at least 1.

Returns

The background threaded data flow to prefetch mini-batches from this flow.

Return type tfsnippet.dataflow.ThreadingFlow

to_arrays_flow(*batch_size*, *shuffle=False*, *skip_incomplete=False*, *random_state=None*)

Convert this data-flow to a *ArrayFlow*.

This method will iterate through the data-flow, collecting mini-batches into arrays, and then construct an ArrayFlow.

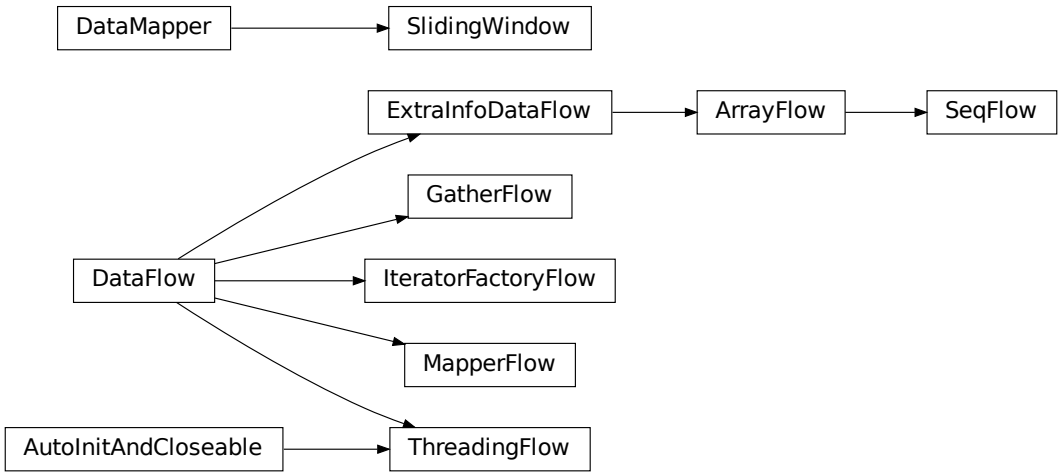
Parameters

- **batch_size** (*int*) – Size of each mini-batch.
- **shuffle** (*bool*) – Whether or not to shuffle data before iterating? (default `False`)
- **skip_incomplete** (*bool*) – Whether or not to exclude the last mini-batch if it is incomplete? (default `False`)
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data before each epoch. (default `None`, construct a new `RandomState`).

Returns The constructed ArrayFlow.

Return type tfsnippet.dataflow.ArrayFlow

Class Inheritance Diagram



2.1.3 tfsnippet.datasets

tfsnippet.datasets Package

Functions

load_cifar10([channels_last, x_shape, ...])	Load the CIFAR-10 dataset as NumPy arrays.
load_cifar100([label_mode, channels_last, ...])	Load the CIFAR-100 dataset as NumPy arrays.
load_fashion_mnist([x_shape, x_dtype, ...])	Load the Fashion MNIST dataset as NumPy arrays.
load_mnist([x_shape, x_dtype, y_dtype, ...])	Load the MNIST dataset as NumPy arrays.

load_cifar10

tfsnippet.datasets.**load_cifar10**(*channels_last=True*, *x_shape=None*, *x_dtype=<type 'numpy.float32'>*, *y_dtype=<type 'numpy.int32'>*, *normalize_x=False*)

Load the CIFAR-10 dataset as NumPy arrays.

Parameters

- **channels_last** (*bool*) – Whether or not to place the channels axis at the last?
- **x_shape** – Reshape each digit into this shape. Default to `(32, 32, 3)` if *channels_last* is `True`, otherwise default to `(3, 32, 32)`.
- **x_dtype** – Cast each digit into this data type. Default *np.float32*.
- **y_dtype** – Cast each label into this data type. Default *np.int32*.

- **normalize_x** (*bool*) – Whether or not to normalize x into `[0, 1]`, by dividing each pixel value with 255.? (default `False`)

Returns

The (train_x, train_y), (test_x, test_y)

Return type (np.ndarray, np.ndarray), (np.ndarray, np.ndarray)

load_cifar100

`tfsnippet.datasets.`**`load_cifar100`**(*label_mode='fine'*, *channels_last=True*, *x_shape=None*, *x_dtype=<type 'numpy.float32'>*, *y_dtype=<type 'numpy.int32'>*, *normalize_x=False*)

Load the CIFAR-100 dataset as NumPy arrays.

Parameters

- **label_mode** – One of {"fine", "coarse"}.
- **channels_last** (*bool*) – Whether or not to place the channels axis at the last? Default `False`.
- **x_shape** – Reshape each digit into this shape. Default to `(32, 32, 3)` if *channels_last* is `True`, otherwise default to `(3, 32, 32)`.
- **x_dtype** – Cast each digit into this data type. Default *np.float32*.
- **y_dtype** – Cast each label into this data type. Default *np.int32*.
- **normalize_x** (*bool*) – Whether or not to normalize x into `[0, 1]`, by dividing each pixel value with 255.? (default `False`)

Returns

The (train_x, train_y), (test_x, test_y)

Return type (np.ndarray, np.ndarray), (np.ndarray, np.ndarray)

load_fashion_mnist

`tfsnippet.datasets.`**`load_fashion_mnist`**(*x_shape=(28, 28)*, *x_dtype=<type 'numpy.float32'>*, *y_dtype=<type 'numpy.int32'>*, *normalize_x=False*)

Load the Fashion MNIST dataset as NumPy arrays.

Homepage: https://github.com/zalandoresearch/fashion-mnist

Parameters

- **x_shape** – Reshape each digit into this shape. Default `(784,)`.
- **x_dtype** – Cast each digit into this data type. Default *np.float32*.
- **y_dtype** – Cast each label into this data type. Default *np.int32*.
- **normalize_x** (*bool*) – Whether or not to normalize x into `[0, 1]`, by dividing each pixel value with 255.? (default `False`)

Returns

The (train_x, train_y), (test_x, test_y)

Return type (np.ndarray, np.ndarray), (np.ndarray, np.ndarray)

load_mnist

tfsnippet.datasets.**load_mnist**(*x_shape=(28, 28)*, *x_dtype=<type 'numpy.float32'>*, *y_dtype=<type 'numpy.int32'>*, *normalize_x=False*)

Load the MNIST dataset as NumPy arrays.

Parameters

- **x_shape** – Reshape each digit into this shape. Default `(28, 28, 1)`.
- **x_dtype** – Cast each digit into this data type. Default *np.float32*.
- **y_dtype** – Cast each label into this data type. Default *np.int32*.
- **normalize_x** (*bool*) – Whether or not to normalize x into `[0, 1]`, by dividing each pixel value with 255.? (default `False`)

Returns

The (train_x, train_y), (test_x, test_y)

Return type (np.ndarray, np.ndarray), (np.ndarray, np.ndarray)

2.1.4 tfsnippet.layers

tfsnippet.layers Package

Functions

act_norm(*args, **kwargs)	ActNorm proposed by (Kingma & Dhariwal, 2018).
as_gated(layer_fn[, sigmoid_bias, default_name])	Wrap a layer function into a gated layer function.
avg_pool2d(*args, **kwargs)	2D average pooling over spatial dimensions.
broadcast_log_det_against_input(log_det, ...)	Broadcast the shape of *log_det* to match the shape of *input*.
conv2d(*args, **kwargs)	2D convolutional layer.
deconv2d(*args, **kwargs)	2D deconvolutional layer.
default_kernel_initializer([weight_norm])	Get the default initializer for layer kernels (i.e., *W* of layers).
dense(*args, **kwargs)	Fully-connected layer.
dropout(*args, **kwargs)	Apply dropout on *input*.
global_avg_pool2d(*args, **kwargs)	2D global average pooling over spatial dimensions.
l2_regularizer(lambda_[, name])	Construct an L2 regularizer that computes the L2 regularization loss.
max_pool2d(*args, **kwargs)	2D max pooling over spatial dimensions.
pixelcnn_2d_input(*args, **kwargs)	Prepare the input for a PixelCNN 2D network (Tim Salimans, 2017).
pixelcnn_2d_output(input)	Get the final output of a PixelCNN 2D network from the previous layer.
pixelcnn_conv2d_resnet(*args, **kwargs)	PixelCNN 2D convolutional ResNet block.
planar_normalizing_flows([n_layers, ...])	Construct a sequential of :class‘PlanarNormalizingFlow‘.
resnet_conv2d_block(*args, **kwargs)	2D convolutional ResNet block.
resnet_deconv2d_block(*args, **kwargs)	2D deconvolutional ResNet block.
resnet_general_block(*args, **kwargs)	A general implementation of ResNet block.

Continued on next page

Table 98 – continued from previous page

shifted_conv2d(*args, **kwargs)	2D convolution with shifted input.
weight_norm(*args, **kwargs)	Weight normalization proposed by (Salimans & Kingma, 2016).

act_norm

tfsnippet.layers.**act_norm**(**args*, ***kwargs*)

ActNorm proposed by (Kingma & Dhariwal, 2018).

Examples:

```
import tfsnippet as spt

# apply act_norm on a dense layer
x = spt.layers.dense(x, units, activation_fn=tf.nn.relu,
                     normalizer_fn=functools.partial(
                         act_norm, initializing=initializing))

# apply act_norm on a conv2d layer
x = spt.layers.conv2d(x, out_channels, (3, 3),
                      channels_last=channels_last,
                      activation_fn=tf.nn.relu,
                      normalizer_fn=functools.partial(
                          act_norm,
                          axis=-1 if channels_last else -3,
                          value_ndims=3,
                          initializing=initializing,
                      ))
```

Parameters

- **input** (*Tensor*) – The input tensor.
- **axis** (*int or Iterable[int]*) – The axis to apply ActNorm. Dimensions not in *axis* will be averaged out when computing the mean of activations. Default *-1*, the last dimension. All items of the *axis* should be covered by *value_ndims*.
- **initializing** (*bool*) – Whether or not to use the input *x* to initialize the layer parameters? (default `True`)
- **scale_type** – One of {"exp", "linear"}. If "exp", `y = (x + bias) * tf.exp(log_scale)`. If "linear", `y = (x + bias) * scale`. Default is "exp".
- **bias_regularizer** – The regularizer for *bias*.
- **bias_constraint** – The constraint for *bias*.
- **log_scale_regularizer** – The regularizer for *log_scale*.
- **log_scale_constraint** – The constraint for *log_scale*.
- **scale_regularizer** – The regularizer for *scale*.
- **scale_constraint** – The constraint for *scale*.
- **trainable** (*bool*) – Whether or not the variables are trainable?
- **epsilon** – Small float to avoid dividing by zero or taking logarithm of zero.
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.

- **scope** (*str*) – The name of the variable scope.

Returns The output after the ActNorm has been applied.

Return type tf.Tensor

as_gated

tfsnippet.layers.**as_gated**(*layer_fn*, *sigmoid_bias=2.0*, *default_name=None*)

Wrap a layer function into a gated layer function.

For example, the following *gated_dense* function:

```
@add_arg_scope
def gated_dense(inputs, units, activation_fn=None, sigmoid_bias=2.,
                name=None, scope=None, **kwargs):
    with tf.name_scope(scope, default_name=name):
        gate = tf.sigmoid(sigmoid_bias +
            dense(inputs, units, scope='gate', **kwargs))
        return gate * dense(
            inputs, units, activation_fn=activation_fn, scope='main',
            **kwargs
        )
```

can be deduced by applying this function:

```
gated_dense = as_gated(dense)
```

Parameters

- **layer_fn** – The layer function to be wrapped.
- **sigmoid_bias** – The constant bias added to the *gate* before applying the sigmoid activation.
- **default_name** – Default name of variable scope.

Returns The wrapped gated layer function.

Notes

If a layer supports *gated* argument (e.g., spt.layers.dense()), it is generally better to use that argument, instead of using this *as_gated()* wrapper on the layer.

avg_pool2d

tfsnippet.layers.**avg_pool2d**(**args*, ***kwargs*)

2D average pooling over spatial dimensions.

Parameters

- **input** (*Tensor*) – The input tensor, at least 4-d.
- **pool_size** (*int or (int, int)*) – Pooling size over spatial dimensions.
- **strides** (*int or (int, int)*) – Strides over spatial dimensions.

- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **padding** – One of {"valid", "same"}, case in-sensitive.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The output tensor.

Return type tf.Tensor

broadcast_log_det_against_input

tfsnippet.layers.**broadcast_log_det_against_input**(*log_det*, *input*, *value_ndims*, *name=None*)

Broadcast the shape of *log_det* to match the shape of *input*.

Parameters

- **log_det** – Tensor, the log-determinant.
- **input** – Tensor, the input.
- **value_ndims** (*int*) – The number of dimensions of each values sample.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The broadcasted log-determinant.

Return type tf.Tensor

conv2d

tfsnippet.layers.**conv2d**(**args*, ***kwargs*)

2D convolutional layer.

Parameters

- **input** (*Tensor*) – The input tensor, at least 4-d.
- **out_channels** (*int*) – The channel numbers of the output.
- **kernel_size** (*int or (int, int)*) – Kernel size over spatial dimensions.
- **strides** (*int or (int, int)*) – Strides over spatial dimensions.
- **dilations** (*int*) – The dilation factor over spatial dimensions.
- **padding** – One of {"valid", "same"}, case in-sensitive.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **activation_fn** – The activation function.
- **normalizer_fn** – The normalizer function.
- **weight_norm** (*bool or (tf.Tensor) -> tf.Tensor)*) – If `True`, apply *weight_norm()* on *kernel*. *use_scale* will be `True` if *normalizer_fn* is not specified, and `False` otherwise. The axis reduction will be determined by the layer.

If it is a callable function, then it will be used to normalize the *kernel* instead of `weight_norm()`. The user must ensure the axis reduction is correct by themselves.

- **gated** (*bool*) – Whether or not to use gate on output? *output = activation_fn(output) * sigmoid(gate)*.
- **gate_sigmoid_bias** (*Tensor*) – The bias added to *gate* before applying the *sigmoid* activation.
- **kernel** (*Tensor*) – Instead of creating a new variable, use this tensor.
- **kernel_mask** (*Tensor*) – If specified, multiply this mask onto *kernel*, i.e., the actual kernel to use will be *kernel * kernel_mask*.
- **kernel_initializer** – The initializer for *kernel*. Would be `default_kernel_initializer(...)` if not specified.
- **kernel_regularizer** – The regularizer for *kernel*.
- **kernel_constraint** – The constraint for *kernel*.
- **use_bias** (*bool or None*) – Whether or not to use *bias*? If `True`, will always use bias. If `None`, will use bias only if *normalizer_fn* is not given. If `False`, will never use bias. Default is `None`.
- **bias** (*Tensor*) – Instead of creating a new variable, use this tensor.
- **bias_initializer** – The initializer for *bias*.
- **bias_regularizer** – The regularizer for *bias*.
- **bias_constraint** – The constraint for *bias*.
- **trainable** (*bool*) – Whether or not the parameters are trainable?
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.

Returns The output tensor.

Return type tf.Tensor

deconv2d

`tfsnippet.layers.`**`deconv2d`**(**args*, ***kwargs*)

2D deconvolutional layer.

Parameters

- **input** (*Tensor*) – The input tensor, at least 4-d.
- **out_channels** (*int*) – The channel numbers of the deconvolution output.
- **kernel_size** (*int or (int, int)*) – Kernel size over spatial dimensions.
- **strides** (*int or (int, int)*) – Strides over spatial dimensions.
- **padding** – One of {"valid", "same"}, case in-sensitive.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")

- **output_shape** – If specified, use this as the shape of the deconvolution output; otherwise compute the size of each dimension by:

```
output_size = input_size * strides
if padding == 'valid':
    output_size += max(kernel_size - strides, 0)
```

- **activation_fn** – The activation function.
- **normalizer_fn** – The normalizer function.
- **weight_norm** (*bool or (tf.Tensor) -> tf.Tensor)*) – If `True`, apply *weight_norm()* on *kernel*. *use_scale* will be `True` if *normalizer_fn* is not specified, and `False` otherwise. The axis reduction will be determined by the layer.

  If it is a callable function, then it will be used to normalize the *kernel* instead of *weight_norm()*. The user must ensure the axis reduction is correct by themselves.
- **gated** (*bool*) – Whether or not to use gate on output? *output = activation_fn(output) * sigmoid(gate)*.
- **gate_sigmoid_bias** (*Tensor*) – The bias added to *gate* before applying the *sigmoid* activation.
- **kernel** (*Tensor*) – Instead of creating a new variable, use this tensor.
- **kernel_initializer** – The initializer for *kernel*. Would be `default_kernel_initializer(...)` if not specified.
- **kernel_regularizer** – The regularizer for *kernel*.
- **kernel_constraint** – The constraint for *kernel*.
- **use_bias** (*bool or None*) – Whether or not to use *bias*? If `True`, will always use bias. If `None`, will use bias only if *normalizer_fn* is not given. If `False`, will never use bias. Default is `None`.
- **bias** (*Tensor*) – Instead of creating a new variable, use this tensor.
- **bias_initializer** – The initializer for *bias*.
- **bias_regularizer** – The regularizer for *bias*.
- **bias_constraint** – The constraint for *bias*.
- **trainable** (*bool*) – Whether or not the parameters are trainable?
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.

Returns The output tensor.

Return type tf.Tensor

default_kernel_initializer

tfsnippet.layers.**default_kernel_initializer**(*weight_norm=False*)

Get the default initializer for layer kernels (i.e., *W* of layers).

Parameters **weight_norm** – Whether or not to apply weight normalization (Salimans & Kingma, 2016) on the kernel? If is not `False` or `None`, will use `tf.random_normal_initializer(0, .05)`.

Returns The default initializer for kernels.

dense

`tfsnippet.layers.`**`dense`**(**args*, ***kwargs*)

Fully-connected layer.

Roughly speaking, the dense layer is defined as:

```
output = activation_fn(
    normalizer_fn(tf.matmul(input, weight_norm_fn(kernel)) + bias))
```

Parameters

- **input** (*Tensor*) – The input tensor, at least 2-d.
- **units** (*int*) – Number of output units.
- **activation_fn** – The activation function.
- **normalizer_fn** – The normalizer function.
- **weight_norm** (*bool or (tf.Tensor) -> tf.Tensor)*) – If `True`, apply *weight_norm()* on *kernel*. *use_scale* will be `True` if *normalizer_fn* is not specified, and `False` otherwise. The axis reduction will be determined by the layer.

  If it is a callable function, then it will be used to normalize the *kernel* instead of *weight_norm()*. The user must ensure the axis reduction is correct by themselves.
- **gated** (*bool*) – Whether or not to use gate on output? *output = activation_fn(output) * sigmoid(gate)*.
- **gate_sigmoid_bias** (*Tensor*) – The bias added to *gate* before applying the *sigmoid* activation.
- **kernel** (*Tensor*) – Instead of creating a new variable, use this tensor.
- **kernel_initializer** – The initializer for *kernel*. Would be `default_kernel_initializer(...)` if not specified.
- **kernel_regularizer** – The regularizer for *kernel*.
- **kernel_constraint** – The constraint for *kernel*.
- **use_bias** (*bool or None*) – Whether or not to use *bias*? If `True`, will always use bias. If `None`, will use bias only if *normalizer_fn* is not given. If `False`, will never use bias. Default is `None`.
- **bias** (*Tensor*) – Instead of creating a new variable, use this tensor.
- **bias_initializer** – The initializer for *bias*.
- **bias_regularizer** – The regularizer for *bias*.
- **bias_constraint** – The constraint for *bias*.
- **trainable** (*bool*) – Whether or not the variables are trainable?

- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.

Returns The output tensor.

Return type tf.Tensor

dropout

tfsnippet.layers.**dropout**(**args*, ***kwargs*)

Apply dropout on *input*.

Parameters

- **input** (*Tensor*) – The input tensor.
- **rate** (*float or tf.Tensor*) – The rate of dropout.
- **noise_shape** (*tuple[int] or tf.Tensor*) – Shape of the noise. If not specified, use the shape of *input*.
- **training** (*bool or tf.Tensor*) – Whether or not the model is under training stage?
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The dropout transformed tensor.

Return type tf.Tensor

global_avg_pool2d

tfsnippet.layers.**global_avg_pool2d**(**args*, ***kwargs*)

2D global average pooling over spatial dimensions.

Parameters

- **input** (*Tensor*) – The input tensor, at least 4-d.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **keepdims** (*bool*) – Whether or not to keep the reduced spatial dimensions? Default is `False`.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The output tensor.

Return type tf.Tensor

l2_regularizer

tfsnippet.layers.**l2_regularizer**(*lambda_*, *name=None*)

Construct an L2 regularizer that computes the L2 regularization loss:

```
output = lambda_ * 0.5 * sum(input ** 2)
```

Parameters

- **lambda** (*float or Tensor or None*) – The coefficiency of L2 regularizer. If *lambda_* is None, will return None.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

A function that computes the L2 regularization term for input tensor. Will be None if *lambda_* is None.

Return type (tf.Tensor) -> tf.Tensor

max_pool2d

tfsnippet.layers.**max_pool2d**(**args*, ***kwargs*)

2D max pooling over spatial dimensions.

Parameters

- **input** (*Tensor*) – The input tensor, at least 4-d.
- **pool_size** (*int or (int, int)*) – Pooling size over spatial dimensions.
- **strides** (*int or (int, int)*) – Strides over spatial dimensions.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **padding** – One of {"valid", "same"}, case in-sensitive.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The output tensor.

Return type tf.Tensor

pixelcnn_2d_input

tfsnippet.layers.**pixelcnn_2d_input**(**args*, ***kwargs*)

Prepare the input for a PixelCNN 2D network (Tim Salimans, 2017).

This method must be applied on the input once before any other PixelCNN 2D layers, for example:

```
input = ...  # the input x

# prepare for the convolution stack
output = spt.layers.pixelcnn_2d_input(input)

# apply the PixelCNN 2D layers.
for i in range(5):
    output = spt.layers.pixelcnn_conv2d_resnet(
        output,
        out_channels=64,
```

(continues on next page)

(continued from previous page)

```
        vertical_kernel_size=(2, 3),
        horizontal_kernel_size=(2, 2),
        activation_fn=tf.nn.leaky_relu,
        normalizer_fn=spt.layers.batch_norm
    )

# get the final output of the PixelCNN 2D network.
output = pixelcnn_2d_output(output)
```

Parameters

- **input** (*Tensor*) – The input tensor, at least 4-d.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **auxiliary_channel** (*bool*) – Whether or not to add a channel to *input*, with all elements set to *1*?
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The PixelCNN layer output.

Return type *PixelCNN2DOutput*

pixelcnn_2d_output

tfsnippet.layers.**pixelcnn_2d_output**(*input*)

Get the final output of a PixelCNN 2D network from the previous layer.

Parameters

- **input** (PixelCNN2DOutput) – The output from the previous PixelCNN layer.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The output tensor.

Return type tf.Tensor

pixelcnn_conv2d_resnet

tfsnippet.layers.**pixelcnn_conv2d_resnet**(**args*, ***kwargs*)

PixelCNN 2D convolutional ResNet block.

Parameters

- **input** (PixelCNN2DOutput) – The output from the previous PixelCNN layer.
- **out_channels** (*int*) – The channel numbers of the output.
- **conv_fn** – The convolution function for "conv_0" and "conv_1" convolutional layers. See *resnet_general_block()*.
- **vertical_kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for "conv_0" and "conv_1" convolutional layers in the PixelCNN vertical stack.

- **horizontal_kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for "conv_0" and "conv_1" convolutional layers in the PixelCNN horizontal stack.
- **strides** (*int or tuple[int]*) – Strides over spatial dimensions, for "conv_0", "conv_1" and "shortcut" convolutional layers.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **use_shortcut_conv** (*True or None*) – If `True`, force to apply a linear convolution transformation on the shortcut path. If `None` (by default), only use shortcut if necessary.
- **shortcut_conv_fn** – The convolution function for the "shortcut" convolutional layer. If not specified, use *conv_fn*.
- **shortcut_kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for the "shortcut" convolutional layer.
- **activation_fn** – The activation function.
- **normalizer_fn** – The normalizer function.
- **dropout_fn** – The dropout function.
- **gated** (*bool*) – Whether or not to use gate on the output of "conv_1"? *conv_1_output = activation_fn(conv_1_output) * sigmoid(gate)*.
- **gate_sigmoid_bias** (*Tensor*) – The bias added to *gate* before applying the *sigmoid* activation.
- **use_bias** (*bool or None*) – Whether or not to use *bias* in "conv_0" and "conv_1"? If `True`, will always use bias. If `None`, will use bias only if *normalizer_fn* is not given. If `False`, will never use bias. Default is `None`.
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.
- ****kwargs** – Other named arguments passed to "conv_0", "conv_1" and "shortcut" convolutional layers.

Returns The PixelCNN layer output.

Return type *PixelCNN2DOutput*

planar_normalizing_flows

tfsnippet.layers.**planar_normalizing_flows**(*n_layers=1*, *w_initializer=<tensorflow.python.ops.init_ops.RandomNorm object>*, *w_regularizer=None*, *b_initializer=<tensorflow.python.ops.init_ops.Zeros object>*, *b_regularizer=None*, *u_initializer=<tensorflow.python.ops.init_ops.RandomNormal object>*, *u_regularizer=None*, *trainable=True*, *name=None*, *scope=None*)

Construct a sequential of :class'PlanarNormalizingFlow'.

Parameters

- **n_layers** (*int*) – The number of :class'PlanarNormalizingFlow'.
- **w_initializer** – The initializer for parameter *w*.

- **w_regularizer** – The regularizer for parameter *w*.
- **b_regularizer** – The regularizer for parameter *b*.
- **b_initializer** – The initializer for parameter *b*.
- **u_regularizer** – The regularizer for parameter *u*.
- **u_initializer** – The initializer for parameter *u*.
- **trainable** (*bool*) – Whether or not the parameters are trainable? (default `True`)
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.

Returns

A ***SequentialFlow*** if *n_layers* > *1*, or a *PlanarNormalizingFlow* if *n_layers* == *1*.

Return type *SequentialFlow* or *PlanarNormalizingFlow*

See also:

tfsnippet.layers.PlanarNormalizingFlow

resnet_conv2d_block

tfsnippet.layers.**resnet_conv2d_block**(**args*, ***kwargs*)

2D convolutional ResNet block.

Parameters

- **input** (*Tensor*) – The input tensor.
- **out_channels** (*int*) – The channel numbers of the output.
- **kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for "conv_0" and "conv_1" convolutional layers.
- **conv_fn** – The convolution function for "conv_0" and "conv_1" convolutional layers. See *resnet_general_block()*.
- **strides** (*int or tuple[int]*) – Strides over spatial dimensions, for "conv_0", "conv_1" and "shortcut" convolutional layers.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **use_shortcut_conv** (*True or None*) – If `True`, force to apply a linear convolution transformation on the shortcut path. If `None` (by default), only use shortcut if necessary.
- **shortcut_conv_fn** – The convolution function for the "shortcut" convolutional layer. If not specified, use *conv_fn*.
- **shortcut_kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for the "shortcut" convolutional layer.
- **resize_at_exit** (*bool*) – If `True`, resize the spatial dimensions at the "conv_1" convolutional layer. If `False`, resize at the "conv_0" convolutional layer. (see above)
- **after_conv_0** – The function to apply on the output of "conv_0" layer.
- **after_conv_1** – The function to apply on the output of "conv_1" layer.

- **activation_fn** – The activation function.
- **normalizer_fn** – The normalizer function.
- **dropout_fn** – The dropout function.
- **gated** (*bool*) – Whether or not to use gate on the output of "conv_1"? *conv_1_output = activation_fn(conv_1_output) * sigmoid(gate)*.
- **gate_sigmoid_bias** (*Tensor*) – The bias added to *gate* before applying the *sigmoid* activation.
- **use_bias** (*bool or None*) – Whether or not to use *bias* in "conv_0" and "conv_1"? If `True`, will always use bias. If `None`, will use bias only if *normalizer_fn* is not given. If `False`, will never use bias. Default is `None`.
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.
- ****kwargs** – Other named arguments passed to "conv_0", "conv_1" and "shortcut" convolutional layers.

Returns The output tensor.

Return type tf.Tensor

See also:

resnet_general_block()

resnet_deconv2d_block

tfsnippet.layers.**resnet_deconv2d_block**(**args*, ***kwargs*)

2D deconvolutional ResNet block.

Parameters

- **input** (*Tensor*) – The input tensor.
- **out_channels** (*int*) – The channel numbers of the output.
- **kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for "conv_0" and "conv_1" convolutional layers.
- **conv_fn** – The deconvolution function for "conv_0" and "conv_1" deconvolutional layers. See *resnet_general_block()*.
- **strides** (*int or tuple[int]*) – Strides over spatial dimensions, for "conv_0", "conv_1" and "shortcut" deconvolutional layers.
- **output_shape** – If specified, use this as the shape of the deconvolution output; otherwise compute the size of each dimension by:

```
output_size = input_size * strides
if padding == 'valid':
    output_size += max(kernel_size - strides, 0)
```

- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")

- **use_shortcut_conv** (*True or None*) – If `True`, force to apply a linear deconvolution transformation on the shortcut path. If `None` (by default), only use shortcut if necessary.
- **shortcut_conv_fn** – The deconvolution function for the "shortcut" deconvolutional layer. If not specified, use *conv_fn*.
- **shortcut_kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for the "shortcut" deconvolutional layer.
- **resize_at_exit** (*bool*) – If `True`, resize the spatial dimensions at the "conv_1" deconvolutional layer. If `False`, resize at the "conv_0" deconvolutional layer. (see above)
- **after_conv_0** – The function to apply on the output of "conv_0" layer.
- **after_conv_1** – The function to apply on the output of "conv_1" layer.
- **activation_fn** – The activation function.
- **normalizer_fn** – The normalizer function.
- **dropout_fn** – The dropout function.
- **gated** (*bool*) – Whether or not to use gate on the output of "conv_1"? *conv_1_output = activation_fn(conv_1_output) * sigmoid(gate)*.
- **gate_sigmoid_bias** (*Tensor*) – The bias added to *gate* before applying the *sigmoid* activation.
- **use_bias** (*bool or None*) – Whether or not to use *bias* in "conv_0" and "conv_1"? If `True`, will always use bias. If `None`, will use bias only if *normalizer_fn* is not given. If `False`, will never use bias. Default is `None`.
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.
- ****kwargs** – Other named arguments passed to "conv_0", "conv_1" and "shortcut" deconvolutional layers.

Returns The output tensor.

Return type tf.Tensor

See also:

resnet_general_block()

resnet_general_block

tfsnippet.layers.**resnet_general_block**(**args*, ***kwargs*)

A general implementation of ResNet block.

The architecture of this ResNet implementation follows the work "Wide residual networks" (Zagoruyko & Komodakis, 2016). It basically does the following things:

```
shortcut = input
if strides != 1 or in_channels != out_channels or use_shortcut_conv:
    shortcut = shortcut_conv_fn(
        input=shortcut,
        out_channels=out_channels,
        kernel_size=shortcut_kernel_size,
```

(continues on next page)

(continued from previous page)

```
        strides=strides,
        scope='shortcut'
    )

residual = input
residual = conv_fn(
    input=activation_fn(normalizer_fn(residual)),
    out_channels=in_channels if resize_at_exit else out_channels,
    kernel_size=kernel_size,
    strides=strides,
    scope='conv_0'
)
residual = after_conv_0(residual)
residual = dropout_fn(residual)
residual = conv_fn(
    input=activation_fn(normalizer_fn(residual)),
    out_channels=out_channels,
    kernel_size=kernel_size,
    strides=strides,
    scope='conv_1'
)
residual = after_conv_1(residual)

output = shortcut + residual
```

Parameters

- **conv_fn** – The convolution function for “conv_0” and “conv_1” convolutional layers. It must accept the following named arguments:
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.
  - input
  - out_channels
  - kernel_size
  - strides
  - channels_last
  - use_bias
  - scope

  Also, it must accept the named arguments specified in *kwargs*.
- **input** (*Tensor*) – The input tensor.
- **in_channels** (*int*) – The channel numbers of the tensor.
- **out_channels** (*int*) – The channel numbers of the output.
- **kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for “conv_0” and “conv_1” convolutional layers.
- **strides** (*int or tuple[int]*) – Strides over spatial dimensions, for “conv_0”, “conv_1” and “shortcut” convolutional layers.

- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **use_shortcut_conv** (*True or None*) – If `True`, force to apply a linear convolution transformation on the shortcut path. If `None` (by default), only use shortcut if necessary.
- **shortcut_conv_fn** – The convolution function for the "shortcut" convolutional layer. It should accept same named arguments as *conv_fn*. If not specified, use *conv_fn*.
- **shortcut_kernel_size** (*int or tuple[int]*) – Kernel size over spatial dimensions, for the "shortcut" convolutional layer.
- **resize_at_exit** (*bool*) – If `True`, resize the spatial dimensions at the "conv_1" convolutional layer. If `False`, resize at the "conv_0" convolutional layer. (see above)
- **after_conv_0** – The function to apply on the output of "conv_0" layer.
- **after_conv_1** – The function to apply on the output of "conv_1" layer.
- **activation_fn** – The activation function.
- **normalizer_fn** – The normalizer function.
- **dropout_fn** – The dropout function.
- **gated** (*bool*) – Whether or not to use gate on the output of "conv_1"? *conv_1_output = activation_fn(conv_1_output) * sigmoid(gate)*.
- **gate_sigmoid_bias** (*Tensor*) – The bias added to *gate* before applying the *sigmoid* activation.
- **use_bias** (*bool or None*) – Whether or not to use *bias* in "conv_0" and "conv_1"? If `True`, will always use bias. If `None`, will use bias only if *normalizer_fn* is not given. If `False`, will never use bias. Default is `None`.
- ****kwargs** – Other named arguments passed to "conv_0", "conv_1" and "shortcut" convolutional layers.

Returns The output tensor.

Return type tf.Tensor

shifted_conv2d

tfsnippet.layers.**shifted_conv2d**(**args*, ***kwargs*)

2D convolution with shifted input.

This method first pads *input* according to the *kernel_size* and *spatial_shift* arguments, then do 2D convolution (using *conv_fn*) with "VALID" padding.

Parameters

- **input** (*Tensor*) – The input tensor, at least 4-d.
- **out_channels** (*int*) – The channel numbers of the output.
- **kernel_size** (*int or (int, int)*) – Kernel size over spatial dimensions.
- **spatial_shift** – The *spatial_shift* should be a tuple with two elements (corresponding to height and width spatial axes), and the elements can only be -1, 0 or 1.

  If the shift for a specific axis is *-1*, then *kernel_size - 1* zeros will be padded at the end of that axis. If the shift is *0*, then *(kernel_size - 1) // 2* zeros will be padded at the front, and

kernel_size // 2 zeros will be padded at the end that axis. Otherwise if the shift is *1*, then *kernel_size + 1* zeros will be padded at the front of that axis.

- **strides** (*int or (int, int)*) – Strides over spatial dimensions.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **conv_fn** – The 2D convolution function. (default *conv2d()*)
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.
- ****kwargs** – Other named parameters passed to *conv_fn*.

Returns The output tensor.

Return type tf.Tensor

weight_norm

tfsnippet.layers.**weight_norm**(**args*, ***kwargs*)

Weight normalization proposed by (Salimans & Kingma, 2016).

Roughly speaking, the weight normalization is defined as:

```
kernel = scale * kernel / tf.sqrt(
    tf.reduce_sum(kernel ** 2, axis=<dimensions not in `axis`>,
                  keepdims=True)
)
```

This function does not support data-dependent initialization for *scale*. If you do need this feature, you have to turn off *scale*, and use *act_norm()* along with *weight_norm()*.

Parameters

- **kernel** – Tensor, the weight *w* to be normalized.
- **axis** (*int or tuple[int]*) – The axis to apply weight normalization. See above description to know what *axis* exactly is.
- **use_scale** (*bool*) – Whether or not to use *scale*. Default True.
- **scale** (*Tensor*) – Instead of creating a new variable, use this tensor.
- **scale_initializer** – The initializer for *scale*.
- **scale_regularizer** – The regularizer for *scale*.
- **scale_constraint** – The constraint for *scale*.
- **trainable** (*bool*) – Whether or not the variables are trainable?
- **epsilon** – Small float number to avoid dividing by zero.
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.

Classes

ActNorm([axis, value_ndims, initialized, ...])	ActNorm proposed by (Kingma & Dhariwal, 2018).
BaseFlow(x_value_ndims[, y_value_ndims, ...])	The basic class for normalizing flows.
BaseLayer([name, scope])	Base class for all neural network layers.
CouplingLayer(shift_and_scale_fn[, axis, ...])	A general implementation of the coupling layer (Dinh et al., 2016).
FeatureMappingFlow(axis, value_ndims, **kwargs)	Base class for flows mapping input features to output features.
FeatureShufflingFlow([axis, value_ndims, ...])	An invertible flow which shuffles the order of input features.
InvertFlow(flow[, name, scope])	Turn a *BaseFlow* into its inverted flow.
InvertibleActivation	Base class for intertible activation functions.
InvertibleActivationFlow(activation, value_ndims)	A flow that converts a *InvertibleActivation* into a flow.
InvertibleConv2d([channels_last, ...])	Invertible 1x1 2D convolution proposed in (Kingma & Dhariwal, 2018).
InvertibleDense([strict_invertible, ...])	Invertible dense layer, modified from the invertible 1x1 2d convolution proposed in (Kingma & Dhariwal, 2018).
LeakyReLU([alpha])	Leaky ReLU activation function.
MultiLayerFlow(n_layers, **kwargs)	Base class for multi-layer normalizing flows.
PixelCNN2DOutput(vertical, horizontal)	The output of a PixelCNN 2D layer, including tensors from the vertical and horizontal convolution stacks.
PlanarNormalizingFlow([w_initializer, ...])	A single layer Planar Normalizing Flow (Danilo 2016) with *tanh* activation function, as well as the invertible trick.
ReshapeFlow(x_value_ndims, y_value_shape[, ...])	A flow which reshapes the last *x_value_ndims* of *x* into *y_value_shape*.
SequentialFlow(flows[, name, scope])	Compose a large flow from a sequential of *BaseFlow*.
SpaceToDepthFlow(block_size[, ...])	A flow which computes `y = space_to_depth(x)`, and conversely `x = depth_to_space(y)`.
SplitFlow(split_axis, left[, join_axis, ...])	A flow which splits input *x* into halves, apply different flows on each half, then concat the output together.

ActNorm

class tfsnippet.layers.**ActNorm**(*axis=-1*, *value_ndims=1*, *initialized=False*, *scale_type='exp'*, *bias_regularizer=None*, *bias_constraint=None*, *log_scale_regularizer=None*, *log_scale_constraint=None*, *scale_regularizer=None*, *scale_constraint=None*, *trainable=True*, *epsilon=1e-06*, *name=None*, *scope=None*)

Bases: tfsnippet.layers.flows.base.FeatureMappingFlow

ActNorm proposed by (Kingma & Dhariwal, 2018).

*y = (x + bias) * scale; log_det = y / scale - bias*

bias and *scale* are initialized such that *y* will have zero mean and unit variance for the initial mini-batch of *x*. It can be initialized only through the forward pass. You may need to use *BaseFlow.invert()* to get a inverted flow if you need to initialize the parameters via the opposite direction.

Attributes Summary

axis	Get the feature axis/axes.
explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
value_ndims	Get the number of value dimensions in both *x* and *y*.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(. . .) <==> x(. . .)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, . . .])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Attributes Documentation

axis

Get the feature axis/axes.

Returns

The feature axis/axes, as is specified in the constructor.

Return type int or tuple[int]

explicitly_invertible

Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name

Get the name of this object.

require_batch_dims

Whether or not this flow requires batch dimensions.

value_ndims

Get the number of value dimensions in both *x* and *y*.

Returns The number of value dimensions in both *x* and *y*.

Return type int

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()
Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)
Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log \det \frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

BaseFlow

class tfsnippet.layers.**BaseFlow**(*x_value_ndims*, *y_value_ndims=None*, *require_batch_dims=False*, *name=None*, *scope=None*)
Bases: tfsnippet.layers.base.BaseLayer

The basic class for normalizing flows.

A normalizing flow transforms a random variable *x* into *y* by an (implicitly) invertible mapping $y = f(x)$, whose Jaccobian matrix determinant $\det \frac{\partial f(x)}{\partial x} \neq 0$, thus can derive $\log p(y)$ from given $\log p(x)$.

Attributes Summary

explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)

Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()

Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

BaseLayer

class tfsnippet.layers.**BaseLayer**(*name=None*, *scope=None*)

Bases: tfsnippet.utils.reuse.VarScopeObject

Base class for all neural network layers.

Attributes Summary

name	Get the name of this object.
variable_scope	Get the variable scope of this object.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.

Attributes Documentation

name

Get the name of this object.

variable_scope

Get the variable scope of this object.

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)

Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)

Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

CouplingLayer

class tfsnippet.layers.**CouplingLayer**(*shift_and_scale_fn*, *axis=-1*, *value_ndims=1*, *secondary=False*, *scale_type='linear'*, *sigmoid_scale_bias=2.0*, *epsilon=1e-06*, *name=None*, *scope=None*)

Bases: tfsnippet.layers.flows.base.FeatureMappingFlow

A general implementation of the coupling layer (Dinh et al., 2016).

Basically, a *CouplingLayer* does the following transformation:

```
x1, x2 = split(x)
if secondary:
    x1, x2 = x2, x1

y1 = x1

shift, scale = shift_and_scale_fn(x1, x2.shape[axis])
if scale_type == 'exp':
    y2 = (x2 + shift) * exp(scale)
elif scale_type == 'sigmoid':
    y2 = (x2 + shift) * sigmoid(scale + sigmoid_scale_bias)
elif scale_type == 'linear':
    y2 = (x2 + shift) * scale
else:
    y2 = x2 + shift

if secondary:
    y1, y2 = y2, y1
y = tf.concat([y1, y2], axis=axis)
```

The inverse transformation, and the log-determinants are computed according to the above transformation, respectively.

Attributes Summary

axis	Get the feature axis/axes.
explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
value_ndims	Get the number of value dimensions in both *x* and *y*.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

axis

Get the feature axis/axes.

Returns

The feature axis/axes, as is specified in the constructor.

Return type int or tuple[int]

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform x into y, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing x, nor can it transform x back into y.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

value_ndims
Get the number of value dimensions in both x and y.

Returns The number of value dimensions in both x and y.

Return type int

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in x.

Returns The number of value dimensions in x.

Return type int

y_value_ndims
Get the number of value dimensions in y.

Returns The number of value dimensions in y.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)

Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()

Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

FeatureMappingFlow

class tfsnippet.layers.**FeatureMappingFlow**(*axis*, *value_ndims*, ***kwargs*)

Bases: tfsnippet.layers.flows.base.BaseFlow

Base class for flows mapping input features to output features.

A *FeatureMappingFlow* must not change the value dimensions, i.e., *x_value_ndims* == *y_value_ndims*. Thus, one single argument *value_ndims* replaces the *x_value_ndims* and *y_value_ndims* arguments.

The *FeatureMappingFlow* transforms a specified axis or a list of specified axes. The axis/axes is/are specified via the argument *axis*. All the *axis* must be covered by *value_ndims*.

Attributes Summary

axis	Get the feature axis/axes.
explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
value_ndims	Get the number of value dimensions in both *x* and *y*.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

axis

Get the feature axis/axes.

Returns

The feature axis/axes, as is specified in the constructor.

Return type int or tuple[int]

explicitly_invertible

Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

value_ndims
Get the number of value dimensions in both *x* and *y*.

Returns The number of value dimensions in both *x* and *y*.

Return type int

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises

- `RuntimeError` – If both *compute_x* and *compute_log_det* are set to `False`.
- `RuntimeError` – If the flow is not explicitly invertible.

invert()

Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log \det \frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default `True`.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises `RuntimeError` – If both *compute_y* and *compute_log_det* are set to `False`.

FeatureShufflingFlow

class tfsnippet.layers.**FeatureShufflingFlow**(*axis=-1*, *value_ndims=1*, *random_state=None*, *name=None*, *scope=None*)

Bases: `tfsnippet.layers.flows.base.FeatureMappingFlow`

An invertible flow which shuffles the order of input features.

This type of flow is proposed in (Kingma & Dhariwal, 2018), as a possible replacement to the alternating pattern of coupling layers proposed in (Dinh et al., 2016). Although the experiments have shown that this flow is inferior to learnt feature mappings (e.g., *InvertibleDense* and *InvertibleConv2d*), it is faster than learnt mappings, and is still superior to the vanilla alternating pattern.

Attributes Summary

axis	Get the feature axis/axes.
explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
value_ndims	Get the number of value dimensions in both *x* and *y*.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Attributes Documentation

axis

Get the feature axis/axes.

Returns

The feature axis/axes, as is specified in the constructor.

Return type int or tuple[int]

explicitly_invertible

Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name

Get the name of this object.

require_batch_dims

Whether or not this flow requires batch dimensions.

value_ndims

Get the number of value dimensions in both *x* and *y*.

Returns The number of value dimensions in both *x* and *y*.

Return type int

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()
Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)
Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

InvertFlow

class tfsnippet.layers.**InvertFlow**(*flow*, *name=None*, *scope=None*)
Bases: tfsnippet.layers.flows.base.BaseFlow

Turn a *BaseFlow* into its inverted flow.

This class is particularly useful when the flow is (theoretically) defined in the opposite direction to the direction of network initialization. For example, define *z* -> *x*, but initialized by feeding *x*.

Attributes Summary

explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the original flow, inverted by this *InvertFlow*.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()
Get the original flow, inverted by this *InvertFlow*.

Returns The original flow.

Return type *BaseFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)
Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

InvertibleActivation

class tfsnippet.layers.**InvertibleActivation**

Bases: object

Base class for intertible activation functions.

An invertible activation function is an element-wise transformation $y = f(x)$, where its inverse function $x = f^{-1}(y)$ exists and can be explicitly computed.

Methods Summary

__call__(...) <==> x(...)	
as_flow(value_ndims[, name, scope])	Convert this activation object into a *BaseFlow*.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
transform(x[, compute_y, compute_log_det, ...])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Methods Documentation

__call__(*...*) <==> *x(...)*

as_flow(*value_ndims*, *name=None*, *scope=None*)

Convert this activation object into a *BaseFlow*.

Parameters

- **value_ndims** (*int*) – Number of value dimensions in both *x* and *y*. *x.ndims - value_ndims == log_det.ndims* and *y.ndims - value_ndims == log_det.ndims*.
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.

Returns The flow.

Return type *InvertibleActivationFlow*

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *value_ndims=0*, *name=None*)

Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **value_ndims** (*int*) – Number of value dimensions. *log_det.ndims == y.ndims - value_ndims*.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

x **and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises

- `RuntimeError` – If both *compute_x* and *compute_log_det* are set to `False`.
- `RuntimeError` – If the flow is not explicitly invertible.

transform(*x*, *compute_y=True*, *compute_log_det=True*, *value_ndims=0*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default `True`.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **value_ndims** (*int*) – Number of value dimensions. *log_det.ndims == x.ndims - value_ndims*.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

y **and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises `RuntimeError` – If both *compute_y* and *compute_log_det* are set to `False`.

InvertibleActivationFlow

class tfsnippet.layers.**InvertibleActivationFlow**(*activation*, *value_ndims*, *name=None*, *scope=None*)

Bases: `tfsnippet.layers.flows.base.BaseFlow`

A flow that converts a *InvertibleActivation* into a flow.

Attributes Summary

activation	Get the invertible activation object.
explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
value_ndims	Get the number of value dimensions.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

activation
Get the invertible activation object.

Returns The invertible activation object.

Return type *InvertibleActivation*

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

value_ndims
Get the number of value dimensions.

Returns The number of value dimensions.

Return type int

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()
Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)
Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.

- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises `RuntimeError` – If both *compute_y* and *compute_log_det* are set to `False`.

InvertibleConv2d

class tfsnippet.layers.**InvertibleConv2d**(*channels_last=True*, *strict_invertible=False*, *random_state=None*, *trainable=True*, *name=None*, *scope=None*)

Bases: `tfsnippet.layers.flows.base.FeatureMappingFlow`

Invertible 1x1 2D convolution proposed in (Kingma & Dhariwal, 2018).

Attributes Summary

axis	Get the feature axis/axes.
explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
value_ndims	Get the number of value dimensions in both *x* and *y*.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Attributes Documentation

axis

Get the feature axis/axes.

Returns

The feature axis/axes, as is specified in the constructor.

Return type int or tuple[int]

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform x into y, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing x, nor can it transform x back into y.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

value_ndims
Get the number of value dimensions in both x and y.

Returns The number of value dimensions in both x and y.

Return type int

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in x.

Returns The number of value dimensions in x.

Return type int

y_value_ndims
Get the number of value dimensions in y.

Returns The number of value dimensions in y.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform y into x, and compute the log-determinant of *f^{-1}* at y (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of y.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()

Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

InvertibleDense

class tfsnippet.layers.**InvertibleDense**(*strict_invertible=False*, *random_state=None*, *trainable=True*, *name=None*, *scope=None*)

Bases: tfsnippet.layers.flows.base.FeatureMappingFlow

Invertible dense layer, modified from the invertible 1x1 2d convolution proposed in (Kingma & Dhariwal, 2018).

Attributes Summary

axis	Get the feature axis/axes.
explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
value_ndims	Get the number of value dimensions in both *x* and *y*.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Attributes Documentation

axis
Get the feature axis/axes.

Returns

The feature axis/axes, as is specified in the constructor.

Return type int or tuple[int]

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

value_ndims
Get the number of value dimensions in both *x* and *y*.

Returns The number of value dimensions in both *x* and *y*.

Return type int

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()
Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)
Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

LeakyReLU

class tfsnippet.layers.**LeakyReLU**(*alpha=0.2*)
Bases: tfsnippet.layers.activations.base.InvertibleActivation

Leaky ReLU activation function.

*y = x if x >= 0 else alpha * x*

Methods Summary

__call__(...) <==> x(...)	
as_flow(value_ndims[, name, scope])	Convert this activation object into a *BaseFlow*.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).
transform(x[, compute_y, compute_log_det, ...])	Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Methods Documentation

__call__(*...*) <==> *x(...)*

as_flow(*value_ndims*, *name=None*, *scope=None*)

Convert this activation object into a *BaseFlow*.

Parameters

- **value_ndims** (*int*) – Number of value dimensions in both *x* and *y*. *x.ndims - value_ndims == log_det.ndims* and *y.ndims - value_ndims == log_det.ndims*.
- **name** (*str*) – Default name of the variable scope. Will be uniquified. If not specified, generate one according to the class name.
- **scope** (*str*) – The name of the variable scope.

Returns The flow.

Return type *InvertibleActivationFlow*

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *value_ndims=0*, *name=None*)

Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default `True`.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **value_ndims** (*int*) – Number of value dimensions. *log_det.ndims == y.ndims - value_ndims*.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises

- `RuntimeError` – If both *compute_x* and *compute_log_det* are set to `False`.
- `RuntimeError` – If the flow is not explicitly invertible.

transform(*x*, *compute_y=True*, *compute_log_det=True*, *value_ndims=0*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default `True`.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **value_ndims** (*int*) – Number of value dimensions. *log_det.ndims == x.ndims - value_ndims*.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises `RuntimeError` – If both *compute_y* and *compute_log_det* are set to `False`.

MultiLayerFlow

class tfsnippet.layers.**MultiLayerFlow**(*n_layers*, ***kwargs*)
Bases: tfsnippet.layers.flows.base.BaseFlow

Base class for multi-layer normalizing flows.

Attributes Summary

explicitly_invertible	Whether or not this flow is explicitly invertible?
n_layers	Get the number of flow layers.
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

n_layers
Get the number of flow layers.

Returns The number of flow layers.

Return type int

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()

Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log \det \frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

PixelCNN2DOutput

class tfsnippet.layers.**PixelCNN2DOutput**(*vertical*, *horizontal*)

Bases: object

The output of a PixelCNN 2D layer, including tensors from the vertical and horizontal convolution stacks.

Attributes Summary

horizontal	Get the horizontal convolution stack output.
vertical	Get the vertical convolution stack output.

Attributes Documentation

horizontal

Get the horizontal convolution stack output.

vertical

Get the vertical convolution stack output.

PlanarNormalizingFlow

class tfsnippet.layers.**PlanarNormalizingFlow**(*w_initializer=<tensorflow.python.ops.init_ops.RandomNormal object>*, *w_regularizer=None*, *b_initializer=<tensorflow.python.ops.init_ops.Zeros object>*, *b_regularizer=None*, *u_initializer=<tensorflow.python.ops.init_ops.RandomNormal object>*, *u_regularizer=None*, *trainable=True*, *name=None*, *scope=None*)

Bases: tfsnippet.layers.flows.base.FeatureMappingFlow

A single layer Planar Normalizing Flow (Danilo 2016) with *tanh* activation function, as well as the invertible trick. The *x* and *y* are assumed to be 1-D random variable (i.e., `value_ndims == 1`)

$$\begin{aligned}\mathbf{y} &= \mathbf{x} + \hat{\mathbf{u}}\tanh(\mathbf{w}^\top \mathbf{x} + b)\\ \hat{\mathbf{u}} &= \mathbf{u} + \left[m(\mathbf{w}^\top \mathbf{u}) - (\mathbf{w}^\top \mathbf{u})\right]\cdot \frac{\mathbf{w}}{\|\mathbf{w}\|_2^2}\\ m(a) &= -1 + \log(1+\exp(a))\end{aligned}$$

Attributes Summary

axis	Get the feature axis/axes.
explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
value_ndims	Get the number of value dimensions in both *x* and *y*.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Attributes Documentation

axis

Get the feature axis/axes.

Returns

The feature axis/axes, as is specified in the constructor.

Return type int or tuple[int]

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform x into y, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing x, nor can it transform x back into y.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

value_ndims
Get the number of value dimensions in both x and y.

Returns The number of value dimensions in both x and y.

Return type int

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in x.

Returns The number of value dimensions in x.

Return type int

y_value_ndims
Get the number of value dimensions in y.

Returns The number of value dimensions in y.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform y into x, and compute the log-determinant of *f^{-1}* at y (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of y.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default `True`.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises

- `RuntimeError` – If both *compute_x* and *compute_log_det* are set to `False`.
- `RuntimeError` – If the flow is not explicitly invertible.

invert()

Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log \det \frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default `True`.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises `RuntimeError` – If both *compute_y* and *compute_log_det* are set to `False`.

ReshapeFlow

class tfsnippet.layers.**ReshapeFlow**(*x_value_ndims*, *y_value_shape*, *require_batch_dims=False*, *name=None*, *scope=None*)

Bases: `tfsnippet.layers.flows.base.BaseFlow`

A flow which reshapes the last *x_value_ndims* of *x* into *y_value_shape*.

Usage:

```
# to reshape a conv2d output into dense input
flow = ReshapeFlow(x_value_ndims=3, y_value_shape=[-1])
x = tf.random_normal(shape=[2, 3, 4, 5])
y, log_det = flow.transform(x)

# y == tf.reshape(x, [2, -1])
# log_det == tf.zeros([2])
```

Attributes Summary

explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()
Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform x into y, and compute the log-determinant of f at x (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of x.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default `True`.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises `RuntimeError` – If both *compute_y* and *compute_log_det* are set to `False`.

SequentialFlow

class tfsnippet.layers.**SequentialFlow**(*flows*, *name=None*, *scope=None*)

Bases: tfsnippet.layers.flows.base.MultiLayerFlow

Compose a large flow from a sequential of *BaseFlow*.

Attributes Summary

explicitly_invertible	Whether or not this flow is explicitly invertible?
flows	Get the immutable flow list.
n_layers	Get the number of flow layers.
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of f at x (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform x into y, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing x, nor can it transform x back into y.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

flows
Get the immutable flow list.

Returns The immutable flow list.

Return type tuple[*BaseFlow*]

n_layers
Get the number of flow layers.

Returns The number of flow layers.

Return type int

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in x.

Returns The number of value dimensions in x.

Return type int

y_value_ndims
Get the number of value dimensions in y.

Returns The number of value dimensions in y.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)

Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()

Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

SpaceToDepthFlow

class tfsnippet.layers.**SpaceToDepthFlow**(*block_size*, *channels_last=True*, *name=None*, *scope=None*)

Bases: tfsnippet.layers.flows.base.BaseFlow

A flow which computes `y = space_to_depth(x)`, and conversely `x = depth_to_space(y)`.

Attributes Summary

explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in *x*.
y_value_ndims	Get the number of value dimensions in *y*.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det \frac{\partial f(x)}{\partial x}$).

Attributes Documentation

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__ (*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)
Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det \frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()
Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform x into y, and compute the log-determinant of f at x (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of x.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default `True`.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default `True`.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be `None` if corresponding *compute_?* argument is set to `False`.

Return type (tf.Tensor, tf.Tensor)

Raises `RuntimeError` – If both *compute_y* and *compute_log_det* are set to `False`.

SplitFlow

class tfsnippet.layers.**SplitFlow**(*split_axis*, *left*, *join_axis=None*, *right=None*, *name=None*, *scope=None*)

Bases: `tfsnippet.layers.flows.base.BaseFlow`

A flow which splits input x into halves, apply different flows on each half, then concat the output together.

Basically, a *SplitFlow* performs the following transformation:

```
x1, x2 = split(x, axis=split_axis)
y1, log_det1 = left.transform(x1)
if right is not None:
    y2, log_det2 = right.transform(x2)
else:
    y2, log_det2 = x2, 0.
y = concat([y1, y2], axis=join_axis)
log_det = log_det1 + log_det2
```

Attributes Summary

explicitly_invertible	Whether or not this flow is explicitly invertible?
name	Get the name of this object.
require_batch_dims	Whether or not this flow requires batch dimensions.
variable_scope	Get the variable scope of this object.
x_value_ndims	Get the number of value dimensions in x.
y_value_ndims	Get the number of value dimensions in y.

Methods Summary

__call__(...) <==> x(...)	
apply(input)	Apply the layer on *input*, to produce output.
build([input])	Build the layer, creating all required variables.
inverse_transform(y[, compute_x, ...])	Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).
invert()	Get the inverted flow from this flow.
transform(x[, compute_y, compute_log_det, name])	Transform *x* into *y*, and compute the log-determinant of *f* at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Attributes Documentation

explicitly_invertible
Whether or not this flow is explicitly invertible?

If a flow is not explicitly invertible, then it only supports to transform *x* into *y*, and corresponding $\log p(x)$ into $\log p(y)$. It cannot compute $\log p(y)$ directly without knowing *x*, nor can it transform *x* back into *y*.

Returns

A boolean indicating whether or not the flow is explicitly invertible.

Return type bool

name
Get the name of this object.

require_batch_dims
Whether or not this flow requires batch dimensions.

variable_scope
Get the variable scope of this object.

x_value_ndims
Get the number of value dimensions in *x*.

Returns The number of value dimensions in *x*.

Return type int

y_value_ndims
Get the number of value dimensions in *y*.

Returns The number of value dimensions in *y*.

Return type int

Methods Documentation

__call__(*...*) <==> *x(...)*

apply(*input*)
Apply the layer on *input*, to produce output.

Parameters **input** (*Tensor or list[Tensor]*) – The input tensor, or a list of input tensors.

Returns The output tensor, or a list of output tensors.

build(*input=None*)
Build the layer, creating all required variables.

Parameters **input** (*Tensor or list[Tensor] or None*) – If *build()* is called within *apply()*, it will be the input tensor(s). Otherwise if it is called separately, it will be None.

inverse_transform(*y*, *compute_x=True*, *compute_log_det=True*, *name=None*)

Transform *y* into *x*, and compute the log-determinant of *f^{-1}* at *y* (i.e., $\log\det\frac{\partial f^{-1}(y)}{\partial y}$).

Parameters

- **y** (*Tensor*) – The samples of *y*.
- **compute_x** (*bool*) – Whether or not to compute $x = f^{-1}(y)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

Returns

***x* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises

- RuntimeError – If both *compute_x* and *compute_log_det* are set to False.
- RuntimeError – If the flow is not explicitly invertible.

invert()

Get the inverted flow from this flow.

The *transform()* will become the *inverse_transform()* in the inverted flow, and the *inverse_transform()* will become the *transform()* in the inverted flow.

If the current flow has not been initialized, it must be initialized via *inverse_transform()* in the new flow.

Returns The inverted flow.

Return type *tfsnippet.layers.InvertFlow*

transform(*x*, *compute_y=True*, *compute_log_det=True*, *name=None*)

Transform *x* into *y*, and compute the log-determinant of f at *x* (i.e., $\log\det\frac{\partial f(x)}{\partial x}$).

Parameters

- **x** (*Tensor*) – The samples of *x*.
- **compute_y** (*bool*) – Whether or not to compute $y = f(x)$? Default True.
- **compute_log_det** (*bool*) – Whether or not to compute the log-determinant? Default True.
- **name** (*str*) – If specified, will use this name as the TensorFlow operational name scope.

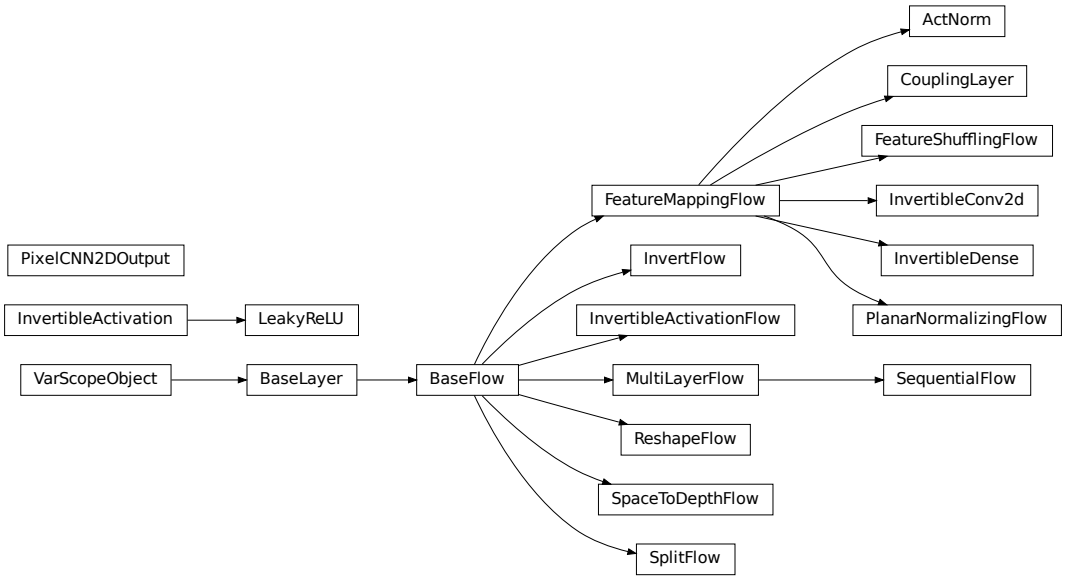
Returns

***y* and the (maybe summed) log-determinant.** The items in the returned tuple might be None if corresponding *compute_?* argument is set to False.

Return type (tf.Tensor, tf.Tensor)

Raises RuntimeError – If both *compute_y* and *compute_log_det* are set to False.

Class Inheritance Diagram



2.1.5 tfsnippet.ops

tfsnippet.ops Package

Functions

add_n_broadcast(tensors[, name])	Add zero or many tensors with broadcasting.
assert_rank(x, ndims[, message, name])	Assert the rank of *x* is *ndims*.
assert_rank_at_least(x, ndims[, message, name])	Assert the rank of *x* is at least *ndims*.
assert_scalar_equal(a, b[, message, name])	Assert 0-d scalar *a* == *b*.
assert_shape_equal(x, y[, message, name])	Assert the shape of *x* equals to *y*.
bits_per_dimension(log_p, value_size[, ...])	Compute "bits per dimension" of *x*.
broadcast_concat(x, y, axis[, name])	Broadcast *x* and *y*, then concat them along *axis*.
broadcast_to_shape(x, shape[, name])	Broadcast *x* to match *shape*.
broadcast_to_shape_strict(x, shape[, name])	Broadcast *x* to match *shape*.
classification_accuracy(y_pred, y_true[, name])	Compute the classification accuracy for *y_pred* and *y_true*.
convert_to_tensor_and_cast(x[, dtype])	Convert *x* into a `tf.Tensor`, and cast its dtype if required.
depth_to_space(input, block_size[, ...])	Wraps `tf.depth_to_space()`, to support tensors higher than 4-d.

Continued on next page

Table 135 – continued from previous page

flatten_to_ndims(x, ndims[, name])	Flatten the front dimensions of *x*, such that the resulting tensor will have at most *ndims* dimensions.
log_mean_exp(x[, axis, keepdims, name])	Compute $\log \frac{1}{K} \sum_{k=1}^{K} \exp(x_k)$.
log_sum_exp(x[, axis, keepdims, name])	Compute $\log \sum_{k=1}^{K} \exp(x_k)$.
maybe_clip_value(x[, min_val, max_val, name])	Maybe clip the elements of *x*.
pixelcnn_2d_sample(fn, inputs, height, width)	Sample output from a PixelCNN 2D network, pixel-by-pixel.
prepend_dims(x[, ndims, name])	Prepend *[1] * ndims* to the beginning of the shape of *x*.
reshape_tail(input, ndims, shape[, name])	Reshape the tail (last) *ndims* into specified *shape*.
shift(input, shift[, name])	Shift each axis of *input* according to *shift*, but keep identical size.
smart_cond(cond, true_fn, false_fn[, name])	Execute *true_fn* or *false_fn* according to *cond*.
softmax_classification_output(logits[, name])	Get the most possible softmax classification output for each logit.
space_to_depth(input, block_size[, ...])	Wraps `tf.space_to_depth()`, to support tensors higher than 4-d.
transpose_conv2d_axis(input, ..., [, name])	Ensure the channels axis of *input* tensor to be placed at the desired axis.
transpose_conv2d_channels_last_to_x(input, ...)	Ensure the channels axis (known to be the last axis) of *input* tensor to be placed at the desired axis.
transpose_conv2d_channels_x_to_last(input, ...)	Ensure the channels axis of *input* tensor to be placed at the last axis.
unflatten_from_ndims(x, static_front_shape, ...)	The inverse transformation of `flatten()`.

add_n_broadcast

`tfsnippet.ops.`**`add_n_broadcast`**(*tensors*, *name=None*)

Add zero or many tensors with broadcasting.

Parameters

- **tensors** (*Iterable[Tensor]*) – A list of tensors to be summed.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The summed tensor.

Return type tf.Tensor

assert_rank

`tfsnippet.ops.`**`assert_rank`**(*x*, *ndims*, *message=None*, *name=None*)

Assert the rank of *x* is *ndims*.

Parameters

- **x** – A tensor.
- **ndims** (*int or tf.Tensor*) – An integer, or a 0-d integer tensor.
- **message** – Message to display when assertion failed.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

The TensorFlow assertion operation, or None if can be statically asserted.

Return type tf.Operation or None

assert_rank_at_least

tfsnippet.ops.**assert_rank_at_least**(*x*, *ndims*, *message=None*, *name=None*)

Assert the rank of *x* is at least *ndims*.

Parameters

- **x** – A tensor.
- **ndims** (*int or tf.Tensor*) – An integer, or a 0-d integer tensor.
- **message** – Message to display when assertion failed.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

The TensorFlow assertion operation, or None if can be statically asserted.

Return type tf.Operation or None

assert_scalar_equal

tfsnippet.ops.**assert_scalar_equal**(*a*, *b*, *message=None*, *name=None*)

Assert 0-d scalar *a* == *b*.

Parameters

- **a** – A 0-d tensor.
- **b** – A 0-d tensor.
- **message** – Message to display when assertion failed.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

The TensorFlow assertion operation, or None if can be statically asserted.

Return type tf.Operation or None

assert_shape_equal

tfsnippet.ops.**assert_shape_equal**(*x*, *y*, *message=None*, *name=None*)

Assert the shape of *x* equals to *y*.

Parameters

- **x** – A tensor.
- **y** – Another tensor, to compare with *x*.
- **message** – Message to display when assertion failed.

- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

The TensorFlow assertion operation, or None if can be statically asserted.

Return type tf.Operation or None

bits_per_dimension

tfsnippet.ops.**bits_per_dimension**(*log_p*, *value_size*, *scale=256.0*, *name=None*)

Compute "bits per dimension" of *x*.

*BPD(x) = - log(p(x)) / (log(2) * Dim(x))*

If $u = s * x$, then:

*BPD(x) = - (log(p(u)) - log(s) * Dim(x)) / (log(2) * Dim(x))*

Parameters

- **log_p** (*Tensor*) – If *scale* is specified, then it should be *log(p(u))*. Otherwise it should be *log(p(x))*.
- **value_size** (*int or Tensor*) – The size of each *x*, i.e., *Dim(x)*.
- **scale** (*float or Tensor or None*) – The scale *s*, where $u = s * x$, and *log_p* is *log(p(u))*.

Returns The computed "bits per dimension" of *x*.

Return type tf.Tensor

broadcast_concat

tfsnippet.ops.**broadcast_concat**(*x*, *y*, *axis*, *name=None*)

Broadcast *x* and *y*, then concat them along *axis*.

This method cannot deal with all possible situations yet. *x* and *y* must have known number of dimensions, and only the deterministic axes will be broadcasted. You must ensure the non-deterministic axes are properly broadcasted by yourself.

Parameters

- **x** – The tensor *x*.
- **y** – The tensor *y*.
- **axis** – The axis to be concatenated.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The broadcast and concatenated tensor.

Return type tf.Tensor

broadcast_to_shape

tfsnippet.ops.**broadcast_to_shape**(*x*, *shape*, *name=None*)

Broadcast *x* to match *shape*.

If `rank(x) > len(shape)`, only the tail dimensions will be broadcasted to match *shape*.

Parameters

- **x** – A tensor.
- **shape** (*tuple[int] or tf.Tensor*) – Broadcast *x* to match this shape.

Returns The broadcasted tensor.

Return type tf.Tensor

broadcast_to_shape_strict

tfsnippet.ops.**broadcast_to_shape_strict**(*x*, *shape*, *name=None*)

Broadcast *x* to match *shape*.

This method requires *rank(x)* to be less than or equal to *len(shape)*. You may use *broadcast_to_shape()* instead, to allow the cases where `rank(x) > len(shape)`.

Parameters

- **x** – A tensor.
- **shape** (*tuple[int] or tf.Tensor*) – Broadcast *x* to match this shape.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The broadcasted tensor.

Return type tf.Tensor

classification_accuracy

tfsnippet.ops.**classification_accuracy**(*y_pred*, *y_true*, *name=None*)

Compute the classification accuracy for *y_pred* and *y_true*.

Parameters

- **y_pred** – The predicted labels.
- **y_true** – The ground truth labels. Its shape must match *y_pred*.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The accuracy.

Return type tf.Tensor

convert_to_tensor_and_cast

tfsnippet.ops.**convert_to_tensor_and_cast**(*x*, *dtype=None*)

Convert *x* into a `tf.Tensor`, and cast its dtype if required.

Parameters

- **x** – The tensor to be converted into a `tf.Tensor`.
- **dtype** (*tf.DType*) – The data type.

Returns The converted and casted tensor.

Return type tf.Tensor

depth_to_space

tfsnippet.ops.**depth_to_space**(*input*, *block_size*, *channels_last=True*, *name=None*)

Wraps `tf.depth_to_space()`, to support tensors higher than 4-d.

Parameters

- **input** – The input tensor, at least 4-d.
- **block_size** (*int*) – An int >= 2, the size of the spatial block.
- **channels_last** (*bool*) – Whether or not the channels axis is the last axis in the input tensor?
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The output tensor.

Return type tf.Tensor

See also:

`tf.depth_to_space()`

flatten_to_ndims

tfsnippet.ops.**flatten_to_ndims**(*x*, *ndims*, *name=None*)

Flatten the front dimensions of *x*, such that the resulting tensor will have at most *ndims* dimensions.

Parameters

- **x** (*Tensor*) – The tensor to be flatten.
- **ndims** (*int*) – The maximum number of dimensions for the resulting tensor.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns (The flatten tensor, the static front shape, and the front shape), or (the original tensor, None, None)

Return type (tf.Tensor, tuple[int or None], tuple[int] or tf.Tensor) or (tf.Tensor, None, None)

log_mean_exp

tfsnippet.ops.**log_mean_exp**(*x*, *axis=None*, *keepdims=False*, *name=None*)

Compute $\log \frac{1}{K} \sum_{k=1}^{K} \exp(x_k)$.

$$\begin{aligned}\log \frac{1}{K} \sum_{k=1}^{K} \exp(x_k) &= \log \left[\exp(x_{max}) \frac{1}{K} \sum_{k=1}^{K} \exp(x_k - x_{max}) \right] \\ &= x_{max} + \log \frac{1}{K} \sum_{k=1}^{K} \exp(x_k - x_{max}) \\ x_{max} &= \max x_k\end{aligned}$$

Parameters

- **x** (*Tensor*) – The input *x*.
- **axis** (*int or tuple[int]*) – The dimension to take average. Default None, all dimensions.
- **keepdims** (*bool*) – Whether or not to keep the summed dimensions? (default False)
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The computed value.

Return type tf.Tensor

log_sum_exp

tfsnippet.ops.**log_sum_exp**(*x*, *axis=None*, *keepdims=False*, *name=None*)

Compute $\log \sum_{k=1}^{K} \exp(x_k)$.

$$\begin{aligned}\log \sum_{k=1}^{K} \exp(x_k) &= \log \left[\exp(x_{max}) \sum_{k=1}^{K} \exp(x_k - x_{max}) \right] \\ &= x_{max} + \log \sum_{k=1}^{K} \exp(x_k - x_{max}) \\ x_{max} &= \max x_k\end{aligned}$$

Parameters

- **x** (*Tensor*) – The input *x*.
- **axis** (*int or tuple[int]*) – The dimension to take summation. Default None, all dimensions.
- **keepdims** (*bool*) – Whether or not to keep the summed dimensions? (default False)
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The computed value.

Return type tf.Tensor

maybe_clip_value

tfsnippet.ops.**maybe_clip_value**(*x*, *min_val=None*, *max_val=None*, *name=None*)

Maybe clip the elements of *x*.

Parameters

- **x** – The tensor maybe to be clipped.
- **min_val** – The minimum value.
- **max_val** – The maximum value.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The clipped tensor.

Return type tf.Tensor

pixelcnn_2d_sample

tfsnippet.ops.**pixelcnn_2d_sample**(*fn*, *inputs*, *height*, *width*, *channels_last=True*, *start=0*, *end=None*, *back_prop=False*, *parallel_iterations=1*, *swap_memory=False*, *name=None*)

Sample output from a PixelCNN 2D network, pixel-by-pixel.

Parameters

- **fn** – *(i: tf.Tensor, inputs: tuple[tf.Tensor]) -> tuple[tf.Tensor]*, the function to derive the outputs of PixelCNN 2D network at iteration *i*. *inputs* are the pixel-by-pixel outputs gathered through iteration *0* to iteration *i - 1*. The iteration index *i* may range from *0* to *height * width - 1*.
- **inputs** (*Iterable[tf.Tensor]*) – The initial input tensors. All the tensors must be at least 4-d, with identical shape.
- **height** (*int or tf.Tensor*) – The height of the outputs.
- **width** (*int or tf.Tensor*) – The width of the outputs.
- **channels_last** (*bool*) – Whether or not the channel axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **start** (*int or tf.Tensor*) – The start iteration, default *0*.
- **end** (*int or tf.Tensor*) – The end (exclusive) iteration. Default *height * width*.
- **parallel_iterations, swap_memory** (*back_prop,*) – Arguments passed to `tf.while_loop()`.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The final outputs.

Return type tuple[tf.Tensor]

prepend_dims

tfsnippet.ops.**prepend_dims**(*x*, *ndims=1*, *name=None*)

Prepend *[1] * ndims* to the beginning of the shape of *x*.

Parameters

- **x** – The tensor *x*.
- **ndims** – Number of *1* to prepend.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The tensor with prepended dimensions.

Return type tf.Tensor

reshape_tail

tfsnippet.ops.**reshape_tail**(*input*, *ndims*, *shape*, *name=None*)

Reshape the tail (last) *ndims* into specified *shape*.

Usage:

```
x = tf.zeros([2, 3, 4, 5, 6])
reshape_tail(x, 3, [-1])  # output: zeros([2, 3, 120])
reshape_tail(x, 1, [3, 2])  # output: zeros([2, 3, 4, 5, 3, 2])
```

Parameters

- **input** (*Tensor*) – The input tensor, at least *ndims* dimensions.
- **ndims** (*int*) – To reshape this number of dimensions at tail.
- **shape** (*Iterable[int] or tf.Tensor*) – The shape of the new tail.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The reshaped tensor.

Return type tf.Tensor

shift

tfsnippet.ops.**shift**(*input*, *shift*, *name=None*)

Shift each axis of *input* according to *shift*, but keep identical size. The extra content will be discarded if shifted outside the original size. Zeros will be padded to the front or end of shifted axes.

Parameters

- **input** (*Tensor*) – The tensor to be shifted.
- **shift** (*Iterable[int]*) – The shift length for each axes. It must be equal to the rank of *input*. For each axis, if its corresponding shift < 0, then the *input* will be shifted to left by *-shift* at that axis. If its shift > 0, then the *input* will be shifted to right by *shift* at that axis.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The output tensor.

Return type tf.Tensor

smart_cond

tfsnippet.ops.**smart_cond**(*cond*, *true_fn*, *false_fn*, *name=None*)

Execute *true_fn* or *false_fn* according to *cond*.

Parameters

- **cond** (*bool or tf.Tensor*) – A bool constant or a tensor.
- **true_fn** (*() -> tf.Tensor*) – The function of the true branch.
- **false_fn** (*() -> tf.Tensor*) – The function of the false branch.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The output tensor.

Return type tf.Tensor

softmax_classification_output

tfsnippet.ops.**softmax_classification_output**(*logits*, *name=None*)

Get the most possible softmax classification output for each logit.

Parameters

- **logits** – The softmax logits. Its last dimension will be treated as the softmax logits dimension, and will be reduced.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns tf.int32 tensor, the class label for each logit.

Return type tf.Tensor

space_to_depth

tfsnippet.ops.**space_to_depth**(*input*, *block_size*, *channels_last=True*, *name=None*)

Wraps `tf.space_to_depth()`, to support tensors higher than 4-d.

Parameters

- **input** – The input tensor, at least 4-d.
- **block_size** (*int*) – An int >= 2, the size of the spatial block.
- **channels_last** (*bool*) – Whether or not the channels axis is the last axis in the input tensor?
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The output tensor.

Return type tf.Tensor

See also:

`tf.space_to_depth()`

transpose_conv2d_axis

`tfsnippet.ops.`**`transpose_conv2d_axis`**(*input*, *from_channels_last*, *to_channels_last*, *name=None*)

Ensure the channels axis of *input* tensor to be placed at the desired axis.

Parameters

- **input** (*tf.Tensor*) – The input tensor, at least 4-d.
- **from_channels_last** (*bool*) – Whether or not the channels axis is the last axis in *input*? (i.e., the data format is "NHWC")
- **to_channels_last** (*bool*) – Whether or not the channels axis should be the last axis in the output tensor?
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The (maybe) transposed output tensor.

Return type tf.Tensor

transpose_conv2d_channels_last_to_x

`tfsnippet.ops.`**`transpose_conv2d_channels_last_to_x`**(*input*, *channels_last*, *name=None*)

Ensure the channels axis (known to be the last axis) of *input* tensor to be placed at the desired axis.

Parameters

- **input** (*tf.Tensor*) – The input tensor, at least 4-d.
- **channels_last** (*bool*) – Whether or not the channels axis should be the last axis in the output tensor?
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The (maybe) transposed output tensor.

Return type tf.Tensor

transpose_conv2d_channels_x_to_last

`tfsnippet.ops.`**`transpose_conv2d_channels_x_to_last`**(*input*, *channels_last*, *name=None*)

Ensure the channels axis of *input* tensor to be placed at the last axis.

Parameters

- **input** (*tf.Tensor*) – The input tensor, at least 4-d.
- **channels_last** (*bool*) – Whether or not the channels axis is the last axis in the *input* tensor?
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The (maybe) transposed output tensor.

Return type tf.Tensor

unflatten_from_ndims

tfsnippet.ops.**unflatten_from_ndims**(*x*, *static_front_shape*, *front_shape*, *name=None*)

The inverse transformation of `flatten()`.

If both *static_front_shape* is None and *front_shape* is None, *x* will be returned without any change.

Parameters

- **x** (*Tensor*) – The tensor to be unflatten.
- **static_front_shape** (*tuple[int or None] or None*) – The static front shape.
- **front_shape** (*tuple[int] or tf.Tensor or None*) – The front shape.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The unflatten x.

Return type tf.Tensor

2.1.6 tfsnippet.preprocessing

tfsnippet.preprocessing Package

Classes

BaseSampler	Base class for samplers.
BernoulliSampler([dtype, random_state])	A `DataMapper` which can sample 0/1 integers according to the input probability.
UniformNoiseSampler([minval, maxval, dtype, ...])	A `DataMapper` which can add uniform noise onto the input array.

BaseSampler

class tfsnippet.preprocessing.**BaseSampler**

Bases: `tfsnippet.dataflows.data_mappers.DataMapper`

Base class for samplers.

Methods Summary

__call__(*arrays)	Transform the input arrays into outputs.
sample(x)	Sample array according to *x*.

Methods Documentation

__call__(**arrays*)
Transform the input arrays into outputs.

Parameters ***arrays** – Arrays to be transformed.

Returns The output arrays.

Return type tuple[np.ndarray]

sample(*x*)
Sample array according to *x*.

Parameters **x** (*np.ndarray*) – The input *x* array.

Returns The sampled array.

Return type np.ndarray

BernoulliSampler

class tfsnippet.preprocessing.**BernoulliSampler**(*dtype=<type 'numpy.int32'>*, *random_state=None*)
Bases: tfsnippet.preprocessing.samplers.BaseSampler

A DataMapper which can sample 0/1 integers according to the input probability. The input is assumed to be float numbers range within [0, 1) or [0, 1].

Attributes Summary

dtype	Get the data type of the sampled array.

Methods Summary

__call__(*arrays)	Transform the input arrays into outputs.
sample(x)	Sample array according to *x*.

Attributes Documentation

dtype
Get the data type of the sampled array.

Methods Documentation

__call__(**arrays*)
Transform the input arrays into outputs.

Parameters ***arrays** – Arrays to be transformed.

Returns The output arrays.

Return type tuple[np.ndarray]

sample(*x*)
Sample array according to *x*.

Parameters **x** (*np.ndarray*) – The input *x* array.

Returns The sampled array.

Return type np.ndarray

UniformNoiseSampler

class tfsnippet.preprocessing.**UniformNoiseSampler**(*minval=0.0*, *maxval=1.0*, *dtype=None*, *random_state=None*)

Bases: tfsnippet.preprocessing.samplers.BaseSampler

A DataMapper which can add uniform noise onto the input array. The data type of the returned array will be the same as the input array, unless *dtype* is specified at construction.

Attributes Summary

dtype	Get the data type of the sampled array.
maxval	Get the upper bound of the uniform noise (excluded).
minval	Get the lower bound of the uniform noise (included).

Methods Summary

__call__(*arrays)	Transform the input arrays into outputs.
sample(x)	Sample array according to *x*.

Attributes Documentation

dtype
Get the data type of the sampled array.

maxval
Get the upper bound of the uniform noise (excluded).

minval
Get the lower bound of the uniform noise (included).

Methods Documentation

__call__(**arrays*)
Transform the input arrays into outputs.

Parameters ***arrays** – Arrays to be transformed.

Returns The output arrays.

Return type tuple[np.ndarray]

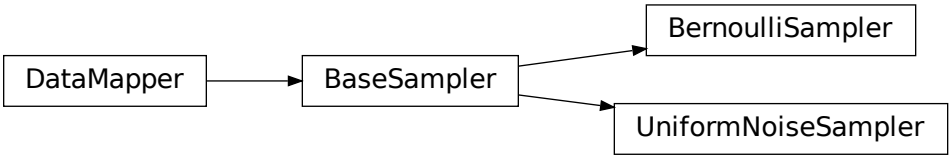
sample(*x*)
Sample array according to *x*.

Parameters **x** (*np.ndarray*) – The input *x* array.

Returns The sampled array.

Return type np.ndarray

Class Inheritance Diagram



2.1.7 tfsnippet.utils

tfsnippet.utils Package

Functions

DocInherit(kclass)	Class decorator to enable *kclass* and all its sub-classes to automatically inherit docstrings from base classes.
add_histogram(tensor[, summary_name, ...])	Add the histogram of *tensor* to the default summary collector, and to *collections*.
add_name_and_scope_arg_doc(method)	Add *name* and *scope* argument to the doc of *method*.
add_name_arg_doc(method)	Add *name* argument to the doc of *method*.
add_summary(summary[, collections])	Add the summary to the default summary collector, and to *collections*.
append_arg_to_doc(doc, arg_doc)	Add the doc for *name* and *scope* argument to the doc string.
append_to_doc(doc, content)	Append content to the doc string.
assert_deps(*args, **kwds)	If `tfsnippet.settings.enable_assertions == True`, open a context that will run *assert_ops*.
camel_to_underscore(name)	Convert a camel-case name to underscore.
concat_shapes(shapes[, name])	Concat shapes from *shapes*.
create_session([lock_memory, ...])	A convenient method to create a TensorFlow session.
default_summary_collector()	Get the *SummaryCollector* object at the top of context stack.
deprecated_arg(old_arg[, new_arg, version])	
ensure_variables_initialized([variables, name])	Ensure variables are initialized.
generate_random_seed()	Generate a new random seed from the default NumPy random state.
get_batch_size(tensor[, axis, name])	Infer the mini-batch size according to *tensor*.
get_cache_root()	Get the cache root directory.

Continued on next page

Table 142 – continued from previous page

get_config_defaults(config)	Get the default config values of *config*.
get_config_validator(type)	Get an instance of *ConfigValidator* for specified *type*.
get_default_scope_name(name[, cls_or_instance])	Generate a valid default scope name.
get_default_session_or_error()	Get the default session.
get_dimension_size(tensor, axis[, name])	Get the size of *tensor* of specified *axis*.
get_dimensions_size(tensor[, axes, name])	Get the size of *tensor* of specified *axes*.
get_model_variables([scope])	Get all model variables (i.e., variables in *MODEL_VARIABLES* collection).
get_rank(tensor[, name])	Get the rank of the tensor.
get_reuse_stack_top()	Get the top of the reuse scope stack.
get_static_shape(tensor)	Get the the static shape of specified *tensor* as a tuple.
get_uninitialized_variables([variables, name])	Get uninitialized variables as a list.
get_variable_ddi(name, initial_value[, ...])	Wraps `tf.get_variable()` to support data-dependent initialization.
get_variables_as_dict([scope, collection])	Get TensorFlow variables as dict.
global_reuse([method_or_scope, _sentinel, scope])	Decorate a function to reuse a variable scope automatically.
humanize_duration(seconds[, short_units])	Format specified time duration as human readable text.
instance_reuse([method_or_scope, _sentinel, ...])	Decorate an instance method to reuse a variable scope automatically.
is_float(x)	Test whether or not *x* is a Python or NumPy float.
is_integer(x)	Test whether or not *x* is a Python or NumPy integer.
is_shape_equal(x, y[, name])	Check whether the shape of *x* equals to *y*.
is_tensor_object(x)	Test whether or not *x* is a tensor object.
is_tensorflow_version_higher_or_equal(version)	Check whether the version of TensorFlow is higher than or equal to *version*.
iter_files(root_dir[, sep])	Iterate through all files in *root_dir*, returning the relative paths of each file.
makedirs(name[, mode, exist_ok])	
maybe_add_histogram(tensor[, summary_name, ...])	If `tfsnippet.settings.auto_histogram == True`, add the histogram of *tensor* via *tfsnippet.add_histogram()*.
maybe_check_numerics(tensor, message[, name])	If `tfsnippet.settings.check_numerics == True`, check the numerics of *tensor*.
maybe_close(*args, **kwds)	Enter a context, and if *obj* has `.close()` method, close it when exiting the context.
minibatch_slices_iterator(length, batch_size)	Iterate through all the mini-batch slices.
model_variable(name[, shape, dtype, ...])	Get or create a model variable.
print_as_table(title, key_values[, hr])	Print a key-value sequence as a table.
register_config_arguments(config, parser[, ...])	Register config to the specified argument parser.
register_config_validator(type, validator_class)	Register a config value validator.
register_tensor_wrapper_class(cls)	Register a sub-class of *TensorWrapper* into TensorFlow type system.
reopen_variable_scope(*args, **kwds)	Reopen the specified *var_scope* and its original name scope.

Continued on next page

Table 142 – continued from previous page

resolve_negative_axis(ndims, axis)	Resolve all negative *axis* indices according to *ndims* into positive.
root_variable_scope(*args, **kwds)	Open the root variable scope and its name scope.
scoped_set_config(*args, **kwds)	Set config values within a context scope.
set_cache_root(cache_root)	Set the root cache directory.
set_random_seed(seed)	Generate random seeds for NumPy, TensorFlow and TFSnippet.
split_numpy_array(array[, portion, size, ...])	Split numpy array into two halves, by portion or by size.
split_numpy_arrays(arrays[, portion, size, ...])	Split numpy arrays into two halves, by portion or by size.
validate_enum_arg(arg_name, arg_value, choices)	Validate the value of a enumeration argument.
validate_group_ndims_arg(group_ndims[, name])	Validate the specified value for *group_ndims* argument.
validate_int_tuple_arg(arg_name, arg_value)	Validate an integer or a tuple of integers, as a tuple of integers.
validate_n_samples_arg(value, name)	Validate the *n_samples* argument.
validate_positive_int_arg(arg_name, arg_value)	Validate a positive integer argument.

DocInherit

tfsnippet.utils.**DocInherit**(*kclass*)

Class decorator to enable *kclass* and all its sub-classes to automatically inherit docstrings from base classes.

Usage:

```
import six


@DocInherit
class Parent(object):
    """Docstring of the parent class."""

    def some_method(self):
        """Docstring of the method."""
        ...

class Child(Parent):
    # inherits the docstring of :meth:`Parent`

    def some_method(self):
        # inherits the docstring of :meth:`Parent.some_method`
        ...
```

Parameters **kclass** (*Type*) – The class to decorate.

Returns The decorated class.

add_histogram

tfsnippet.utils.**add_histogram**(*tensor*, *summary_name=None*, *strip_scope=False*, *collections=None*, *name=None*)

Add the histogram of *tensor* to the default summary collector, and to *collections*.

Parameters

- **tensor** – Take histogram of this tensor.
- **summary_name** – Specify the summary name for *tensor*.
- **strip_scope** – If `True`, strip the name scope from *tensor.name* when adding the histogram.
- **collections** – Also add the histogram to these collections. Defaults to *self.collections*.

Returns The serialized histogram tensor of *tensor*.

add_name_and_scope_arg_doc

tfsnippet.utils.**add_name_and_scope_arg_doc**(*method*)
Add *name* and *scope* argument to the doc of *method*.

add_name_arg_doc

tfsnippet.utils.**add_name_arg_doc**(*method*)
Add *name* argument to the doc of *method*.

add_summary

tfsnippet.utils.**add_summary**(*summary*, *collections=None*)
Add the summary to the default summary collector, and to *collections*.

Parameters

- **summary** – TensorFlow summary tensor.
- **collections** – Also add the summary to these collections. Defaults to *self.collections*.

Returns The *summary* tensor.

append_arg_to_doc

tfsnippet.utils.**append_arg_to_doc**(*doc*, *arg_doc*)
Add the doc for *name* and *scope* argument to the doc string.

Parameters

- **doc** – The original doc string.
- **arg_doc** – The argument documentations.

Returns The updated doc string.

Return type str

append_to_doc

tfsnippet.utils.**append_to_doc**(*doc*, *content*)
Append content to the doc string.

Parameters

- **doc** (*str*) – The original doc string.
- **content** (*str*) – The new doc string, which should be a standalone section.

Returns The modified doc string.

Return type str

assert_deps

tfsnippet.utils.**assert_deps**(**args*, ***kwds*)

If `tfsnippet.settings.enable_assertions == True`, open a context that will run *assert_ops*. Otherwise do nothing.

Parameters **assert_ops** (*Iterable[tf.Operation or None]*) – A list of assertion operations. `None` items will be ignored.

Yields *bool* –

A boolean indicate whether or not the assertion operations are not empty, and are executed.

camel_to_underscore

tfsnippet.utils.**camel_to_underscore**(*name*)

Convert a camel-case name to underscore.

concat_shapes

tfsnippet.utils.**concat_shapes**(*shapes*, *name=None*)

Concat shapes from *shapes*.

Parameters

- **shapes** (*Iterable[tuple[int] or tf.Tensor]*) – List of shape tuples or tensors.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The concatenated shape.

Return type tuple[int] or tf.Tensor

create_session

tfsnippet.utils.**create_session**(*lock_memory=True*, *log_device_placement=False*, *allow_soft_placement=True*, ***kwargs*)

A convenient method to create a TensorFlow session.

Parameters

- **lock_memory** (*True or False or float*) –
  - If `True`, lock all free memory.
  - **If `False`, set *allow_growth* to True, i.e., not to lock** all free memory.

– If float, lock this portion of memory.

(default `None`)

- **log_device_placement** (*bool*) – Whether to log the placement of graph nodes. (default `False`)
- **allow_soft_placement** (*bool*) – Whether or not to allow soft placement? (default `True`)
- ****kwargs** – Other named parameters to be passed to *tf.ConfigProto*.

Returns The TensorFlow session.

Return type tf.Session

default_summary_collector

`tfsnippet.utils.`**`default_summary_collector`**`()`

Get the *SummaryCollector* object at the top of context stack.

Returns The summary collector.

Return type *SummaryCollector*

deprecated_arg

`tfsnippet.utils.`**`deprecated_arg`**`(`*old_arg*, *new_arg=None*, *version=None*`)`

ensure_variables_initialized

`tfsnippet.utils.`**`ensure_variables_initialized`**`(`*variables=None*, *name=None*`)`

Ensure variables are initialized.

Parameters

- **variables** (*list[tf.Variable] or dict[str, tf.Variable]*) – Ensure only the variables within this collection to be initialized. If not specified, will ensure all variables within the collection *tf.GraphKeys.GLOBAL_VARIABLES* to be initialized.
- **name** (*str*) – TensorFlow name scope of the graph nodes. (default *ensure_variables_initialized*)

generate_random_seed

`tfsnippet.utils.`**`generate_random_seed`**`()`

Generate a new random seed from the default NumPy random state.

Returns The new random seed.

Return type int

get_batch_size

tfsnippet.utils.**get_batch_size**(*tensor*, *axis=0*, *name=None*)
Infer the mini-batch size according to *tensor*.

Parameters

- **tensor** (*tf.Tensor*) – The input placeholder.
- **axis** (*int*) – The axis of mini-batches. Default is 0.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The batch size.

Return type int or tf.Tensor

get_cache_root

tfsnippet.utils.**get_cache_root**()
Get the cache root directory.

Returns Path of the cache root directory.

Return type str

get_config_defaults

tfsnippet.utils.**get_config_defaults**(*config*)
Get the default config values of *config*.

Parameters **config** – An instance of *Config*, or a class which is a subclass of *Config*.

Returns The default config values of *config*.

Return type dict[str, any]

get_config_validator

tfsnippet.utils.**get_config_validator**(*type*)
Get an instance of *ConfigValidator* for specified *type*.

Parameters **type** – The value type.

Returns The config value validator.

Return type *ConfigValidator*

get_default_scope_name

tfsnippet.utils.**get_default_scope_name**(*name*, *cls_or_instance=None*)
Generate a valid default scope name.

Parameters

- **name** (*str*) – The base name.

- **cls_or_instance** – The class or the instance object, optional. If it has attribute `variable_scope`, then `variable_scope.name` will be used as a hint for the name prefix. Otherwise, its class name will be used as the name prefix.

Returns The generated scope name.

Return type str

get_default_session_or_error

`tfsnippet.utils.`**`get_default_session_or_error`**`()`

Get the default session.

Returns The default session.

Return type tf.Session

Raises `RuntimeError` – If there's no active session.

get_dimension_size

`tfsnippet.utils.`**`get_dimension_size`**`(`*tensor*, *axis*, *name=None*`)`

Get the size of *tensor* of specified *axis*.

Parameters

- **tensor** (*tf.Tensor*) – The tensor to be tested.
- **axis** (*Iterable[int] or None*) – The dimension to be queried.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns An integer or a tensor, the size of queried dimension.

Return type int or tf.Tensor

get_dimensions_size

`tfsnippet.utils.`**`get_dimensions_size`**`(`*tensor*, *axes=None*, *name=None*`)`

Get the size of *tensor* of specified *axes*.

If *axes* is `None`, select the size of all dimensions.

Parameters

- **tensor** (*tf.Tensor*) – The tensor to be tested.
- **axes** (*Iterable[int] or None*) – The dimensions to be selected.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns

A tuple of integers if all selected dimensions have static sizes. Otherwise a tensor.

Return type tuple[int] or tf.Tensor

get_model_variables

tfsnippet.utils.**get_model_variables**(*scope=None*)
Get all model variables (i.e., variables in *MODEL_VARIABLES* collection).

Parameters **scope** – If specified, will obtain variables only within this scope.

Returns The model variables.

Return type list[tf.Variable]

get_rank

tfsnippet.utils.**get_rank**(*tensor*, *name=None*)
Get the rank of the tensor.

Parameters

- **tensor** (*tf.Tensor*) – The tensor to be tested.
- **name** (*str*) – TensorFlow name scope of the graph nodes.
- **name** – Default name of the name scope. If not specified, generate one according to the method name.

Returns The rank.

Return type int or tf.Tensor

get_reuse_stack_top

tfsnippet.utils.**get_reuse_stack_top**()
Get the top of the reuse scope stack.

Returns The top of the reuse scope stack.

Return type tf.VaribleScope

get_static_shape

tfsnippet.utils.**get_static_shape**(*tensor*)
Get the the static shape of specified *tensor* as a tuple.

Parameters **tensor** – The tensor object.

Returns

The static shape tuple, or None if the dimensions of *tensor* is not deterministic.

Return type tuple[int or None] or None

get_uninitialized_variables

tfsnippet.utils.**get_uninitialized_variables**(*variables=None*, *name=None*)
Get uninitialized variables as a list.

Parameters

- **variables** (*list[tf.Variable]*) – Collect only uninitialized variables within this list. If not specified, will collect all uninitialized variables within `tf.GraphKeys.GLOBAL_VARIABLES` collection.
- **name** (*str*) – TensorFlow name scope of the graph nodes.

Returns Uninitialized variables.

Return type list[tf.Variable]

get_variable_ddi

`tfsnippet.utils.`**`get_variable_ddi`**(*name*, *initial_value*, *shape=None*, *dtype=tf.float32*, *initializing=False*, *regularizer=None*, *constraint=None*, *trainable=True*, *collections=None*, ***kwargs*)

Wraps `tf.get_variable()` to support data-dependent initialization.

Parameters

- **name** – Name of the variable.
- **initial_value** – The data-dependent initial value of the variable.
- **shape** – Shape of the variable.
- **dtype** – Data type of the variable.
- **initializing** (*bool*) – Whether or not it is building the graph for data-dependent initialization? Ignored if *initial_value* is absent.
- **regularizer** – Regularizer of the variable.
- **constraint** – Constraint of the variable.
- **trainable** (*bool*) – Whether or not to the variable is trainable?
- **collections** (*Iterable[str]*) – Add the variable to these collections.
- ****kwargs** – Other named parameters passed to `tf.get_variable()`.

Returns The variable or the tensor.

Return type tf.Variable or tf.Tensor

get_variables_as_dict

`tfsnippet.utils.`**`get_variables_as_dict`**(*scope=None*, *collection='variables'*)

Get TensorFlow variables as dict.

Parameters

- **scope** (*str or tf.VariableScope or None*) – If `None`, will collect all the variables within current graph. If a `str` or a `tf.VariableScope`, will collect the variables only from this scope. (default `None`)
- **collection** (*str*) – Collect the variables only from this collection. (default `tf.GraphKeys.GLOBAL_VARIABLES`)

Returns

Dict which maps from names to TensorFlow variables. The names will be the full names of variables if *scope* is not specified, or the *relative names* within the *scope* otherwise. By *relative names* we mean the variable names without the common scope name prefix.

Return type dict[str, tf.Variable]

global_reuse

tfsnippet.utils.**global_reuse**(*method_or_scope=None*, *_sentinel=None*, *scope=None*)

Decorate a function to reuse a variable scope automatically.

The first time to enter a function decorated by this utility will open a new variable scope under the root variable scope. This variable scope will be reused the next time to enter this function. For example:

```
@global_reuse
def foo():
    return tf.get_variable('bar', ...)

bar = foo()
bar_2 = foo()
assert(bar is bar_2)  # should be True
```

By default the name of the variable scope should be chosen according to the name of the decorated method. You can change this behavior by specifying an alternative name, for example:

```
@global_reuse('dense')
def dense_layer(inputs):
    w = tf.get_variable('w', ...)  # name will be 'dense/w'
    b = tf.get_variable('b', ...)  # name will be 'dense/b'
    return tf.matmul(w, inputs) + b
```

If you have two functions sharing the same scope name, they will not use the same variable scope. Instead, one of these two functions will have its scope name added with a suffix '_?', for example:

```
@global_reuse('foo')
def foo_1():
    return tf.get_variable('bar', ...)

@global_reuse('foo')
def foo_2():
    return tf.get_variable('bar', ...)

assert(foo_1().name == 'foo/bar')
assert(foo_2().name == 'foo_1/bar')
```

The variable scope name will depend on the calling order of these two functions, so you should better not guess the scope name by yourself.

Note: If you use Keras, you SHOULD NOT create a Keras layer inside a *global_reuse* decorated function. Instead, you should create it outside the function, and pass it into the function.

See also:

tfsnippet.utils.instance_reuse()

humanize_duration

tfsnippet.utils.**humanize_duration**(*seconds*, *short_units=True*)

Format specified time duration as human readable text.

Parameters

- **seconds** – Number of seconds of the time duration.
- **short_units** (*bool*) – Whether or not to use short units ("d", "h", "m", "s") instead of long units ("day", "hour", "minute", "second")? (default `False`)

Returns The formatted time duration.

Return type str

instance_reuse

`tfsnippet.utils.`**`instance_reuse`**(*method_or_scope=None*, *_sentinel=None*, *scope=None*)

Decorate an instance method to reuse a variable scope automatically.

This decorator should be applied to unbound instance methods, and the instance that owns the methods should have `variable_scope` attribute. The first time to enter a decorated method will open a new variable scope under the *variable_scope* of the instance. This variable scope will be reused the next time to enter this method. For example:

```
class Foo(object):

    def __init__(self, name):
        with tf.variable_scope(name) as vs:
            self.variable_scope = vs

    @instance_reuse
    def bar(self):
        return tf.get_variable('bar', ...)

foo = Foo()
bar = foo.bar()
bar_2 = foo.bar()
assert(bar is bar_2)  # should be True
```

By default the name of the variable scope should be chosen according to the name of the decorated method. You can change this behavior by specifying an alternative name, for example:

```
class Foo(object):

    @instance_reuse('scope_name')
    def foo(self):
        # name will be self.variable_scope.name + '/foo/bar'
        return tf.get_variable('bar', ...)
```

Unlike the behavior of *global_reuse()*, if you have two methods sharing the same scope name, they will indeed use the same variable scope. For example:

```
class Foo(object):

    @instance_reuse('foo')
    def foo_1(self):
        return tf.get_variable('bar', ...)

    @instance_reuse('foo')
    def foo_2(self):
        return tf.get_variable('bar', ...)
```

(continues on next page)

(continued from previous page)

```
    @instance_reuse('foo')
    def foo_2(self):
        return tf.get_variable('bar2', ...)

foo = Foo()
foo.foo_1()  # its name should be variable_scope.name + '/foo/bar'
foo.foo_2()  # should raise an error, because 'bar' variable has
             # been created, but the decorator of `foo_2` does not
             # aware of this, so has not set ``reused = True``.
foo.foo_3()  # its name should be variable_scope.name + '/foo/bar2'
```

The reason to take this behavior is because the TensorFlow seems to have some absurd behavior when using `tf.variable_scope(..., default_name=?)` to uniquify the variable scope name. In some cases we the following absurd behavior would appear:

```
@global_reuse
def foo():
    with tf.variable_scope(None, default_name='bar') as vs:
        return vs

vs1 = foo()  # vs.name == 'foo/bar'
vs2 = foo()  # still expected to be 'foo/bar', but sometimes would be
             # 'foo/bar_1'. this absurd behavior is related to the
             # entering and exiting of variable scopes, which is very
             # hard to diagnose.
```

In order to compensate such behavior, if you have specified the `scope` argument of a *VarScopeObject*, then it will always take the desired variable scope. Also, constructing a *VarScopeObject* within a method or a function decorated by *global_reuse* or *instance_reuse* has been totally disallowed.

See also:

tfsnippet.utils.VarScopeObject, *tfsnippet.utils.global_reuse()*

is_float

tfsnippet.utils.**is_float**(*x*)

Test whether or not *x* is a Python or NumPy float.

Parameters **x** – The object to be tested.

Returns A boolean indicating whether *x* is a Python or NumPy float.

Return type bool

is_integer

tfsnippet.utils.**is_integer**(*x*)

Test whether or not *x* is a Python or NumPy integer.

Parameters **x** – The object to be tested.

Returns A boolean indicating whether *x* is a Python or NumPy integer.

Return type bool

is_shape_equal

tfsnippet.utils.**is_shape_equal**(*x*, *y*, *name=None*)

Check whether the shape of *x* equals to *y*.

Parameters

- **x** – A tensor.
- **y** – Another tensor, to compare with *x*.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The static or dynamic comparison result.

Return type bool or tf.Tensor

is_tensor_object

tfsnippet.utils.**is_tensor_object**(*x*)

Test whether or not *x* is a tensor object.

`tf.Tensor`, `tf.Variable`, *TensorWrapper* and `zhusuan.StochasticTensor` are considered to be tensor objects.

Parameters **x** – The object to be tested.

Returns A boolean indicating whether *x* is a tensor object.

Return type bool

is_tensorflow_version_higher_or_equal

tfsnippet.utils.**is_tensorflow_version_higher_or_equal**(*version*)

Check whether the version of TensorFlow is higher than or equal to *version*.

Parameters **version** (*str*) – Expected version of TensorFlow.

Returns True if higher or equal to, False if not.

Return type bool

iter_files

tfsnippet.utils.**iter_files**(*root_dir*, *sep='/'*)

Iterate through all files in *root_dir*, returning the relative paths of each file. The sub-directories will not be yielded. :param root_dir: The root directory, from which to iterate. :type root_dir: str :param sep: The separator for the relative paths. :type sep: str

Yields *str* – The relative paths of each file.

makedirs

tfsnippet.utils.**makedirs**(*name*, *mode=511*, *exist_ok=False*)

maybe_add_histogram

tfsnippet.utils.**maybe_add_histogram**(*tensor*, *summary_name=None*, *strip_scope=False*, *collections=None*, *name=None*)

If `tfsnippet.settings.auto_histogram == True`, add the histogram of *tensor* via *tfsnippet.add_histogram()*. Otherwise do nothing.

Parameters

- **tensor** – Take histogram of this tensor.
- **summary_name** – Specify the summary name for *tensor*.
- **strip_scope** – If `True`, strip the name scope from *tensor.name* when adding the histogram.
- **collections** – Add the histogram to these collections. Defaults to *[tfsnippet.GraphKeys.AUTO_HISTOGRAM]*.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The serialized histogram tensor of *tensor*.

See also:

tfsnippet.add_histogram()

maybe_check_numerics

tfsnippet.utils.**maybe_check_numerics**(*tensor*, *message*, *name=None*)

If `tfsnippet.settings.check_numerics == True`, check the numerics of *tensor*. Otherwise do nothing.

Parameters

- **tensor** – The tensor to be checked.
- **message** – The message to display when numerical issues occur.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The tensor, whose numerics have been checked.

Return type tf.Tensor

maybe_close

tfsnippet.utils.**maybe_close**(**args*, ***kwds*)

Enter a context, and if *obj* has `.close()` method, close it when exiting the context.

Parameters **obj** – The object maybe to close.

Yields The specified *obj*.

minibatch_slices_iterator

tfsnippet.utils.**minibatch_slices_iterator**(*length*, *batch_size*, *skip_incomplete=False*)

Iterate through all the mini-batch slices.

Parameters

- **length** (*int*) – Total length of data in an epoch.
- **batch_size** (*int*) – Size of each mini-batch.
- **skip_incomplete** (*bool*) – If `True`, discard the final batch if it contains less than *batch_size* number of items. (default `False`)

Yields

slice: Slices of each mini-batch. The last mini-batch may contain less indices than *batch_size*.

model_variable

tfsnippet.utils.**model_variable**(*name*, *shape=None*, *dtype=None*, *initializer=None*, *regularizer=None*, *constraint=None*, *trainable=True*, *collections=None*, ***kwargs*)

Get or create a model variable.

When the variable is created, it will be added to both *GLOBAL_VARIABLES* and *MODEL_VARIABLES* collection.

Parameters

- **name** – Name of the variable.
- **shape** – Shape of the variable.
- **dtype** – Data type of the variable.
- **initializer** – Initializer of the variable.
- **regularizer** – Regularizer of the variable.
- **constraint** – Constraint of the variable.
- **trainable** (*bool*) – Whether or not the variable is trainable?
- **collections** – In addition to *GLOBAL_VARIABLES* and *MODEL_VARIABLES*, also add the variable to these collections.
- ****kwargs** – Other named arguments passed to `tf.get_variable()`.

Returns The variable.

Return type tf.Variable

print_as_table

tfsnippet.utils.**print_as_table**(*title*, *key_values*, *hr='='*)

Print a key-value sequence as a table.

Parameters

- **title** – Title of the table.

- **key_values** – Dict, or key-value sequence.
- **hr** – Character for the horizon line.

register_config_arguments

tfsnippet.utils.**register_config_arguments**(*config*, *parser*, *prefix=None*, *title=None*, *description=None*, *sort_keys=False*)

Register config to the specified argument parser.

Usage:

```
class YourConfig(Config):
    max_epoch = 1000
    learning_rate = 0.01
    activation = ConfigField(
        str, default='leaky_relu', choices=['relu', 'leaky_relu'])

# First, you should obtain an instance of your config object
config = YourConfig()

# You can then parse config values from CLI arguments.
# For example, if sys.argv[1:] == ['--max_epoch=2000']:
from argparse import ArgumentParser
parser = ArgumentParser()
spt.register_config_arguments(config, parser)
parser.parse_args(sys.argv[1:])

# Now you can access the config value `config.max_epoch == 2000`
print(config.max_epoch)
```

Parameters

- **config** (*Config*) – The config object.
- **parser** (*ArgumentParser*) – The argument parser.
- **prefix** (*str*) – Optional prefix of the config keys. *new_config_key* = *prefix* + '.' + *old_config_key*
- **title** (*str*) – If specified, will create an argument group to collect all the config arguments.
- **description** (*str*) – The description of the argument group.
- **sort_keys** (*bool*) – Whether or not to sort the config keys before registering to the parser? (default *False*)

register_config_validator

tfsnippet.utils.**register_config_validator**(*type*, *validator_class*)

Register a config value validator.

Parameters

- **type** – The value type.
- **validator_class** – The validator class type.

register_tensor_wrapper_class

tfsnippet.utils.**register_tensor_wrapper_class**(*cls*)
Register a sub-class of *TensorWrapper* into TensorFlow type system.

Parameters **cls** – The subclass of *TensorWrapper* to be registered.

reopen_variable_scope

tfsnippet.utils.**reopen_variable_scope**(**args*, ***kwds*)
Reopen the specified *var_scope* and its original name scope.

Parameters

- **var_scope** (*tf.VariableScope*) – The variable scope instance.
- ****kwargs** – Named arguments for opening the variable scope.

resolve_negative_axis

tfsnippet.utils.**resolve_negative_axis**(*ndims*, *axis*)
Resolve all negative *axis* indices according to *ndims* into positive.

Usage:

```
resolve_negative_axis(4, [0, -1, -2])  # output: (0, 3, 2)
```

Parameters

- **ndims** (*int*) – Number of total dimensions.
- **axis** (*Iterable[int]*) – The axis indices.

Returns The resolved positive axis indices.

Return type tuple[int]

Raises ValueError – If any index in *axis* is out of range.

root_variable_scope

tfsnippet.utils.**root_variable_scope**(**args*, ***kwds*)
Open the root variable scope and its name scope.

Parameters ****kwargs** – Named arguments for opening the root variable scope.

scoped_set_config

tfsnippet.utils.**scoped_set_config**(**args*, ***kwds*)
Set config values within a context scope.

Parameters

- **config** (Config) – The config object to set.
- ****kwargs** – The key-value pairs.

set_cache_root

tfsnippet.utils.**set_cache_root**(*cache_root*)
Set the root cache directory.

Parameters **cache_root** (*str*) – The cache root directory. It will be normalized to absolute path.

set_random_seed

tfsnippet.utils.**set_random_seed**(*seed*)
Generate random seeds for NumPy, TensorFlow and TFSnippet.

Parameters **seed** (*int*) – The seed used to generate the separated seeds for all concerning modules.

split_numpy_array

tfsnippet.utils.**split_numpy_array**(*array*, *portion=None*, *size=None*, *shuffle=True*)
Split numpy array into two halves, by portion or by size.

Parameters

- **array** (*np.ndarray*) – A numpy array to be splitted.
- **portion** (*float*) – Portion of the second half. Ignored if *size* is specified.
- **size** (*int*) – Size of the second half.
- **shuffle** (*bool*) – Whether or not to shuffle before splitting?

Returns Splitted two halves of the array.

Return type tuple[np.ndarray]

split_numpy_arrays

tfsnippet.utils.**split_numpy_arrays**(*arrays*, *portion=None*, *size=None*, *shuffle=True*, *random_state=None*)
Split numpy arrays into two halves, by portion or by size.

Parameters

- **arrays** (*Iterable[np.ndarray]*) – Numpy arrays to be splitted.
- **portion** (*float*) – Portion of the second half. Ignored if *size* is specified.
- **size** (*int*) – Size of the second half.
- **shuffle** (*bool*) – Whether or not to shuffle before splitting?
- **random_state** (*RandomState*) – Optional numpy RandomState for shuffling data. (default `None`, construct a new `RandomState`).

Returns Splitted two halves of arrays.

Return type (tuple[np.ndarray], tuple[np.ndarray])

validate_enum_arg

tfsnippet.utils.**validate_enum_arg**(*arg_name*, *arg_value*, *choices*, *nullable=False*)

Validate the value of a enumeration argument.

Parameters

- **arg_name** – Name of the argument.
- **arg_value** – Value of the argument.
- **choices** – Valid choices of the argument value.
- **nullable** – Whether or not the argument can be None?

Returns The validated argument value.

Raises ValueError – If *arg_value* is not valid.

validate_group_ndims_arg

tfsnippet.utils.**validate_group_ndims_arg**(*group_ndims*, *name=None*)

Validate the specified value for *group_ndims* argument.

If the specified *group_ndims* is a dynamic Tensor, additional assertion will be added to the graph node of *group_ndims*.

Parameters

- **group_ndims** – Object to be validated.
- **name** – TensorFlow name scope of the graph nodes. (default "validate_group_ndims")

Returns The validated *group_ndims*.

Return type tf.Tensor or int

Raises ValueError – If the specified *group_ndims* cannot pass validation.

validate_int_tuple_arg

tfsnippet.utils.**validate_int_tuple_arg**(*arg_name*, *arg_value*, *nullable=False*)

Validate an integer or a tuple of integers, as a tuple of integers.

Parameters

- **arg_name** (*str*) – Name of the argument.
- **arg_value** (*int or Iterable[int]*) – An integer, or an iterable collection of integers, to be casted into tuples of integers.
- **nullable** (*bool*) – Whether or not None value is accepted?

Returns The tuple of integers.

Return type tuple[int]

validate_n_samples_arg

tfsnippet.utils.**validate_n_samples_arg**(*value*, *name*)
Validate the *n_samples* argument.

Parameters

- **value** – An int32 value, a int32 `tf.Tensor`, or `None`.
- **name** (*str*) – Name of the argument (in error message).

Returns The validated *n_samples* argument value.

Return type int or tf.Tensor

Raises *TypeError or ValueError or None* – If the value cannot be validated.

validate_positive_int_arg

tfsnippet.utils.**validate_positive_int_arg**(*arg_name*, *arg_value*)
Validate a positive integer argument.

Parameters

- **arg_name** (*str*) – Name of the argument.
- **arg_value** (*int*) – The value to be validated.

Returns The validated positive integer.

Return type int

Classes

AutoInitAndCloseable	Classes with `__init__()` to initialize its internal states, and also `close()` to destroy these states.
BaseRegistry([ignore_case])	A base class for implement a type or object registry.
BoolConfigValidator	Config value validator for boolean values.
CacheDir(name[, cache_root])	Class to manipulate a cache directory.
ClassRegistry([ignore_case])	A subclass of *BaseRegistry*, dedicated for classes.
Config	Base class for defining config values.
ConfigField(type[, default, description, ...])	A config field.
ConfigValidator	Base config value validator.
ConsoleTable(col_count[, col_space, ...])	A class to help format a console table.
ContextStack([initial_factory])	A thread-local context stack for general purpose.
Disposable	Classes which can only be used once.
DisposableContext	Base class for contexts which can only be entered once.
ETA([take_initial_snapshot])	Class to help compute the Estimated Time Ahead (ETA).
EventSource([allowed_event_keys])	An object that may trigger events.
Extractor(archive_file)	The base class for all archive extractors.
FloatConfigValidator	Config value validator for float values.
GraphKeys	Defines TensorFlow graph collection keys for TFSnippet.
InputSpec([shape, dtype])	Class to describe the specification for an input tensor.

Continued on next page

Table 143 – continued from previous page

IntConfigValidator	Config value validator for integer values.
InvertibleMatrix(size[, strict, dtype, ...])	A matrix initialized to be an invertible, orthogonal matrix.
NoReentrantContext	Base class for contexts which are not reentrant (i.e., if there is a context opened by `__enter__`, and it has not called `__exit__`, the `__enter__` cannot be called again).
ParamSpec(*args, **kwargs)	Class to describe the specification for a parameter.
PermutationMatrix(data)	A non-trainable permutation matrix.
RarExtractor(fpath)	Extractor for ".rar" files.
StatisticsCollector([shape])	Computing $\mathrm{E}[X]$ and $\mathrm{Var}[X]$ online.
StrConfigValidator	Config value validator for string values.
SummaryCollector([collections, ...])	Collecting summaries and histograms added by *tfsnippet.add_summary()* and *tfsnippet.add_histogram()*.
TFSnippetConfig	Global configurations of TFSnippet.
TarExtractor(fpath)	Extractor for ".tar", ".tar.gz", ".tgz", ".tar.bz2", ".tbz", ".tbz2", ".tb2", ".tar.xz", ".txz" files.
`TemporaryDirectory`([suffix, prefix, dir])	Create and return a temporary directory.
TensorArgValidator(name)	Class to validate argument values of tensors.
TensorSpec([shape, dtype])	Base class to describe and validate the specification of a tensor.
TensorWrapper	Tensor-like object that wraps a *tf.Tensor* instance.
VarScopeObject([name, scope])	Base class for objects that own a variable scope.
VarScopeRandomState(variable_scope)	A sub-class of `np.random.RandomState`, which uses a variable-scope dependent seed.
ZipExtractor(fpath)	Extractor for ".zip" files.
deprecated([message, version])	Decorate a class, a method or a function to be deprecated.

AutoInitAndCloseable

class tfsnippet.utils.**AutoInitAndCloseable**

Bases: `object`

Classes with *__init__()* to initialize its internal states, and also *close()* to destroy these states. The *__init__()* method can be repeatedly called, which will cause initialization only at the first call. Thus other methods may always call *__init__()* at beginning, which can bring auto-initialization to the class.

A context manager is implemented: *__init__()* is explicitly called when entering the context, while `destroy()` is called when exiting the context.

Methods Summary

close()	Ensure the internal states are destroyed.
init()	Ensure the internal states are initialized.

Methods Documentation

close()

Ensure the internal states are destroyed.

init()
Ensure the internal states are initialized.

BaseRegistry

class tfsnippet.utils.**BaseRegistry**(*ignore_case=False*)
Bases: object

A base class for implement a type or object registry.

Usage:

```
registry = BaseRegistry()
registry.register('MNIST', spt.datasets.MNIST())
```

Attributes Summary

ignore_case	Whether or not to ignore the case?

Methods Summary

get(name)	Get an object.
register(name, obj)	Register an object.

Attributes Documentation

ignore_case
Whether or not to ignore the case?

Methods Documentation

get(*name*)
Get an object.

Parameters **name** (*str*) – Name of the object.

Returns The retrieved object.

Raises KeyError – If *name* is not registered.

register(*name*, *obj*)
Register an object.

Parameters

- **name** (*str*) – Name of the object.
- **obj** – The object.

BoolConfigValidator

class tfsnippet.utils.**BoolConfigValidator**
Bases: tfsnippet.utils.config_utils.ConfigValidator

Config value validator for boolean values.

Methods Summary

validate(value[, strict])	Validate the *value*.

Methods Documentation

validate(*value*, *strict=False*)
Validate the *value*.

Parameters

- **value** – The value to be validated.
- **strict** (*bool*) – If `True`, disable type conversion. If `False`, the validator will try its best to convert the input *value* into desired type.

Returns The validated value.

Raises *ValueError, TypeError* – If the value cannot pass validation.

CacheDir

class tfsnippet.utils.**CacheDir**(*name*, *cache_root=None*)
Bases: `object`

Class to manipulate a cache directory.

Attributes Summary

cache_root	Get the cache root directory.
name	Get the name of this cache directory under *cache_root*.
path	Get the absolute path of this cache directory.

Methods Summary

download(uri[, filename, show_progress, ...])	Download a file into this *CacheDir*.
download_and_extract(uri[, filename, ...])	Download a file into this *CacheDir*, and extract it.
extract_file(archive_file[, extract_dir, ...])	Extract an archive file into this *CacheDir*.
purge_all()	Delete everything in this *CacheDir*.
resolve(sub_path)	Resolve a sub path relative to `self.path`.

Attributes Documentation

cache_root
Get the cache root directory.

name
Get the name of this cache directory under *cache_root*.

path
Get the absolute path of this cache directory.

Methods Documentation

download(*uri*, *filename=None*, *show_progress=None*, *progress_file=<open file '<stderr>', mode 'w'>*, *hasher=None*, *expected_hash=None*)
Download a file into this *CacheDir*.

Parameters

- **uri** (*str*) – The URI to be retrieved.
- **filename** (*str*) – The filename to use as the downloaded file. If *filename* already exists in this *CacheDir*, will not download *uri*. Default None, will automatically infer *filename* according to *uri*.
- **show_progress** (*bool*) – Whether or not to show interactive progress bar? If not specified, will show progress only if *progress_file* is *std.stdout* or *std.stderr*, and if *progress_file.isatty()* is True.
- **progress_file** – The file object where to write the progress. (default sys.stderr)
- **hasher** – A hasher algorithm instance from *hashlib*. If specified, will compute the hash of downloaded content, and validate against *expected_hash*.
- **expected_hash** (*str*) – The expected hash of downloaded content.

Returns The absolute path of the downloaded file.

Return type str

Raises ValueError – If *filename* cannot be inferred.

download_and_extract(*uri*, *filename=None*, *extract_dir=None*, *show_progress=None*, *progress_file=<open file '<stderr>', mode 'w'>*, *hasher=None*, *expected_hash=None*)
Download a file into this *CacheDir*, and extract it.

Parameters

- **uri** (*str*) – The URI to be retrieved.
- **filename** (*str*) – The filename to use as the downloaded file. If *filename* already exists in this *CacheDir*, will not download *uri*. Default None, will automatically infer *filename* according to *uri*.
- **extract_dir** (*str*) – The name to use as the extracted directory. If *extract_dir* already exists in this *CacheDir*, will not extract *archive_file*. Default None, will automatically infer *extract_dir* according to *filename*.
- **show_progress** (*bool*) – Whether or not to show interactive progress bar? If not specified, will show progress only if *progress_file* is *std.stdout* or *std.stderr*, and if *progress_file.isatty()* is True.
- **progress_file** – The file object where to write the progress. (default sys.stderr)
- **hasher** – A hasher algorithm instance from *hashlib*. If specified, will compute the hash of downloaded content, and validate against *expected_hash*.
- **expected_hash** (*str*) – The expected hash of downloaded content.

Returns The absolute path of the extracted directory.

Return type str

Raises `ValueError` – If *filename* or *extract_dir* cannot be inferred.

extract_file(*archive_file*, *extract_dir=None*, *show_progress=None*, *progress_file=<open file '<stderr>', mode 'w'>*)
Extract an archive file into this *CacheDir*.

Parameters

- **archive_file** (*str*) – The path of the archive file.
- **extract_dir** (*str*) – The name to use as the extracted directory. If *extract_dir* already exists in this *CacheDir*, will not extract *archive_file*. Default `None`, will automatically infer *extract_dir* according to *archive_file*.
- **show_progress** (*bool*) – Whether or not to show interactive progress bar? If not specified, will show progress only if *progress_file* is *std.stdout* or *std.stderr*, and if *progress_file.isatty()* is `True`.
- **progress_file** – The file object where to write the progress. (default `sys.stderr`)

Returns The absolute path of the extracted directory.

Return type str

Raises `ValueError` – If *extract_dir* cannot be inferred.

purge_all()
Delete everything in this *CacheDir*.

resolve(*sub_path*)
Resolve a sub path relative to `self.path`.

Parameters **sub_path** – The sub path to resolve.

Returns The resolved absolute path of *sub_path*.

ClassRegistry

class tfsnippet.utils.**ClassRegistry**(*ignore_case=False*)
Bases: `tfsnippet.utils.registry.BaseRegistry`

A subclass of *BaseRegistry*, dedicated for classes.

Usage:

```
Class MyClass(object):

    def __init__(self, value, message):
        ...

registry = ClassRegistry()
registry.register('MyClass', MyClass)

obj = registry.create_object('MyClass', 123, message='message')
```

Attributes Summary

ignore_case	Whether or not to ignore the case?

Methods Summary

construct(name, *args, **kwargs)	Construct an object according to class *name* and arguments.
get(name)	Get an object.
register(name, obj)	Register an object.

Attributes Documentation

ignore_case
Whether or not to ignore the case?

Methods Documentation

construct (*name*, **args*, ***kwargs*)
Construct an object according to class *name* and arguments.

Parameters

- **name** (*str*) – Name of the class.
- ***args** – Arguments passed to the class constructor.
- ****kwargs** – Named arguments passed to the class constructor.

Returns The constructed object.

Raises KeyError – If *name* is not registered.

get (*name*)
Get an object.

Parameters **name** (*str*) – Name of the object.

Returns The retrieved object.

Raises KeyError – If *name* is not registered.

register (*name*, *obj*)
Register an object.

Parameters

- **name** (*str*) – Name of the object.
- **obj** – The object.

Config

class tfsnippet.utils.**Config**
Bases: object

Base class for defining config values.

Derive sub-classes of *Config*, and define config values as public class attributes. For example:

```
class YourConfig(Config):
    max_epoch = 100
    learning_rate = 0.01
    activation = ConfigField(str, default='leaky_relu',
                             choices=['sigmoid', 'relu', 'leaky_relu'])
    l2_regularization = ConfigField(float, default=None)

config = YourConfig()
```

When an attribute is assigned, it will be validated by:

1. If the attribute is defined as a *ConfigField*, then its `validate()` will be used to validate the value.
2. If the attribute is not `None`, and a validator is registered for *type(value)*, then an instance of this type of validator will be used to validate the value.
3. Otherwise if the attribute is not defined, or is `None`, then no validation will be performed.

For example:

```
config.l2_regularization = 5e-4  # okay
config.l2_regularization = 'xxx'  # raise an error
config.activation = 'sigmoid'  # okay
config.activation = 'tanh'  # raise an error
config.new_attribute = 'yyy'  # okay
```

The config object also implements dict-like interface:

```
# to test whether a key exists
print(key in config)

# to iterate through all config values
for key in config:
    print(key, config[key])

# to set a config value
config[key] = value
```

You may get all the config values of a config object as dict:

```
print(config_to_dict(config))
```

Or you may get the default values of the config class as dict:

```
print(config_defaults(YourConfig))
print(config_defaults(config))  # same as the above line
```

Methods Summary

to_dict()	Get the config values as a dict.
update(key_values)	Update the config values from *key_values*.

Methods Documentation

to_dict()
Get the config values as a dict.

Returns The config values.

Return type dict[str, any]

update(*key_values*)
Update the config values from *key_values*.

Parameters **key_values** – A dict, or a sequence of (key, value) pairs.

ConfigField

class tfsnippet.utils.**ConfigField**(*type*, *default=None*, *description=None*, *nullable=False*, *choices=None*)

Bases: object

A config field.

Attributes Summary

choices	Get the valid choices of the config value.
default_value	Get the default config value.
description	Get the config description.
nullable	Whether or not None is a valid value?
type	Get the value type.

Methods Summary

validate(value[, strict])	Validate the config *value*.

Attributes Documentation

choices
Get the valid choices of the config value.

default_value
Get the default config value.

description
Get the config description.

nullable
Whether or not None is a valid value?

type
Get the value type.

Methods Documentation

validate(*value*, *strict=False*)
Validate the config *value*.

Parameters

- **value** – The value to be validated.

- **strict** (*bool*) – If `True`, disable type conversion. If `False`, the validator will try its best to convert the input *value* into desired type.

Returns The validated value.

ConfigValidator

class tfsnippet.utils.**ConfigValidator**
Bases: `object`

Base config value validator.

Methods Summary

validate(value[, strict])	Validate the *value*.

Methods Documentation

validate(*value*, *strict=False*)
Validate the *value*.

Parameters

- **value** – The value to be validated.
- **strict** (*bool*) – If `True`, disable type conversion. If `False`, the validator will try its best to convert the input *value* into desired type.

Returns The validated value.

Raises *ValueError, TypeError* – If the value cannot pass validation.

ConsoleTable

class tfsnippet.utils.**ConsoleTable**(*col_count*, *col_space=3*, *col_align=None*, *expand_col=0*)
Bases: `object`

A class to help format a console table.

Methods Summary

add_config(config[, sort_keys])	Add a config to the table.
add_dict(key_values[, sort_keys])	Add a key-value sequence to the table.
add_hr([c])	Add a horizon line.
add_key_values(key_values[, sort_keys])	Add a key-value sequence to the table.
add_row(columns)	Add a row with full columns.
add_skip()	Add an empty row.
add_title(title[, top_right])	Add row of title.
format()	Format the table to string.

Methods Documentation

add_config(*config*, *sort_keys=False*)
Add a config to the table.

Parameters

- **config** (Config) – A config object.
- **sort_keys** (*bool*) – Whether or not to sort the keys?

add_dict(*key_values*, *sort_keys=False*)
Add a key-value sequence to the table.

Parameters

- **key_values** – Dict, or key-value sequence.
- **sort_keys** (*bool*) – Whether or not to sort the keys?

add_hr(*c='-'*)
Add a horizon line.

Parameters **c** (*str*) – The character of the horizon line.

add_key_values(*key_values*, *sort_keys=False*)
Add a key-value sequence to the table.

Parameters

- **key_values** – Dict, or key-value sequence.
- **sort_keys** (*bool*) – Whether or not to sort the keys?

add_row(*columns*)
Add a row with full columns.

Parameters **columns** (*Iterable[str]*) – The column contents.

add_skip()
Add an empty row.

add_title(*title*, *top_right=None*)
Add row of title.

Parameters

- **title** (*str*) – The title content.
- **top_right** (*str*) – Optional top-right content.

format()
Format the table to string.

Returns The formatted table.

Return type str

ContextStack

class tfsnippet.utils.**ContextStack**(*initial_factory=None*)
Bases: object

A thread-local context stack for general purpose.

Usage:

```
stack = ContextStack()
stack.push(dict())  # push an object to the top of thread local stack
stack.top()[key] = value  # use the top object
stack.pop()  # pop an object from the top of thread local stack
```

Attributes Summary

items

Methods Summary

pop()
push(context)
top()

Attributes Documentation

items

Methods Documentation

pop()

push(*context*)

top()

Disposable

class tfsnippet.utils.**Disposable**
Bases: object

Classes which can only be used once.

DisposableContext

class tfsnippet.utils.**DisposableContext**
Bases: tfsnippet.utils.concepts.NoReentrantContext

Base class for contexts which can only be entered once.

ETA

class tfsnippet.utils.**ETA**(*take_initial_snapshot=True*)
Bases: object

Class to help compute the Estimated Time Ahead (ETA).

Methods Summary

get_eta(progress[, now, take_snapshot])	Get the Estimated Time Ahead (ETA).
take_snapshot(progress[, now])	Take a snapshot of `(progress, now)`, for later computing ETA.

Methods Documentation

get_eta (*progress*, *now=None*, *take_snapshot=True*)
Get the Estimated Time Ahead (ETA).

Parameters

- **progress** – The current progress, range in `[0, 1]`.
- **now** – The current timestamp in seconds. If not specified, use `time.time()`.
- **take_snapshot** (*bool*) – Whether or not to take a snapshot of the specified `(progress, now)`? (default `True`)

Returns

The remaining seconds, or None if the ETA cannot be estimated.

Return type float or None

take_snapshot (*progress*, *now=None*)
Take a snapshot of `(progress, now)`, for later computing ETA.

Parameters

- **progress** – The current progress, range in `[0, 1]`.
- **now** – The current timestamp in seconds. If not specified, use `time.time()`.

EventSource

class tfsnippet.utils.**EventSource** (*allowed_event_keys=None*)
Bases: object

An object that may trigger events.

This class is designed to either be the parent of another class, or be a member of another object. For example:

```
def event_handler(**kwargs):
    print('event triggered: args {}, kwargs {}'.format(args, kwargs))

# use alone
class SomeObject(EventSource):

    def func(self, **kwargs):
        self.fire('some_event', **kwargs)

obj = SomeObject()
obj.on('some_event', event_handler)

# use as a member
class SomeObject(object):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        self.events = EventSource()

    def func(self, **kwargs):
        self.events.fire('some_event', **kwargs)

obj = SomeObject()
obj.events.on('some_event', event_handler)
```

Methods Summary

clear_event_handlers([event_key])	Clear all event handlers.
fire(event_key, *args, **kwargs)	Fire an event.
off(event_key, handler)	De-register an event handler.
on(event_key, handler)	Register a new event handler.
reverse_fire(event_key, *args, **kwargs)	Fire an event, call event handlers in reversed order of registration.

Methods Documentation

clear_event_handlers(*event_key=None*)
Clear all event handlers.

Parameters **event_key** (*str or None*) – If specified, clear all event handlers of this name. Otherwise clear all event handlers.

fire(*event_key*, **args*, ***kwargs*)
Fire an event.

Parameters

- **event_key** (*str*) – The event key.
- ***args** – Arguments to be passed to the event handler.
- ****kwargs** – Named arguments to be passed to the event handler.

Raises KeyError – If *event_key* is not allowed.

off(*event_key*, *handler*)
De-register an event handler.

Parameters

- **event_key** (*str*) – The event key.
- **handler** ((*args, **kwargs) -> any) – The event handler.

Raises ValueError – If *handler* is not a registered event handler of the specified event *event_key*.

on(*event_key*, *handler*)
Register a new event handler.

Parameters

- **event_key** (*str*) – The event key.
- **handler** ((*args, **kwargs) -> any) – The event handler.

Raises KeyError – If *event_key* is not allowed.

reverse_fire(*event_key*, **args*, ***kwargs*)
Fire an event, call event handlers in reversed order of registration.

Parameters

- **event_key** (*str*) – The event key.
- ***args** – Arguments to be passed to the event handler.
- ****kwargs** – Named arguments to be passed to the event handler.

Raises KeyError – If *event_key* is not allowed.

Extractor

class tfsnippet.utils.**Extractor**(*archive_file*)
Bases: object

The base class for all archive extractors.

```
from tfsnippet.utils import Extractor, maybe_close

with Extractor.open('a.zip') as archive_file:
    for name, f in archive_file:
        with maybe_close(f):  # This file object may not be closeable,
                              # thus we surround it by ``maybe_close()``
            print('the content of {} is:'.format(name))
            print(f.read())
```

Methods Summary

close()	Close the extractor.
iter_extract()	Extract files from the archive with an iterator.
open(file_path)	Create an *Extractor* instance for given archive file.

Methods Documentation

close()
Close the extractor.

iter_extract()
Extract files from the archive with an iterator.

You may simply iterate over a *Extractor* object, which is same as calling to this method.

Yields *(str, file-like)* –

Tuples of (name, file-like object), the filename and corresponding file-like object for each file in the archive. The returned file-like object may or may not be closeable. You may surround it by maybe_close().

static open(*file_path*)
Create an *Extractor* instance for given archive file.

Parameters **file_path** (*str*) – The path of the archive file.

Returns The specified extractor instance.

Return type *Extractor*

Raises `IOError` – If the `file_path` is not a supported archive.

FloatConfigValidator

class tfsnippet.utils.**FloatConfigValidator**
Bases: tfsnippet.utils.config_utils.ConfigValidator

Config value validator for float values.

Methods Summary

validate(value[, strict])	Validate the *value*.

Methods Documentation

validate(*value*, *strict=False*)
Validate the *value*.

Parameters

- **value** – The value to be validated.
- **strict** (*bool*) – If `True`, disable type conversion. If `False`, the validator will try its best to convert the input *value* into desired type.

Returns The validated value.

Raises *ValueError, TypeError* – If the value cannot pass validation.

GraphKeys

class tfsnippet.utils.**GraphKeys**
Bases: object

Defines TensorFlow graph collection keys for TFSnippet.

Attributes Summary

AUTO_HISTOGRAM

Attributes Documentation

AUTO_HISTOGRAM = 'TFSNIPPET_AUTO_HISTOGRAM'

InputSpec

class tfsnippet.utils.**InputSpec**(*shape=None*, *dtype=None*)
Bases: tfsnippet.utils.tensor_spec.TensorSpec

Class to describe the specification for an input tensor.

Mostly identical with *TensorSpec*.

Attributes Summary

dtype	Get the data type of the tensor.
shape	Get the shape specification.
value_ndims	Get the value shape ndims.
value_shape	Get the value shape (the shape excluding leading "...").

Methods Summary

validate(name, x)	Validate the input tensor *x*.

Attributes Documentation

dtype
Get the data type of the tensor.

Returns The data type, or None.

Return type tf.DType or None

shape
Get the shape specification.

Returns The value shape, or None.

Return type tuple[int or str or None] or None

value_ndims
Get the value shape ndims.

Returns The value shape ndims, or None.

Return type int or None

value_shape
Get the value shape (the shape excluding leading "...").

Returns The value shape, or None.

Return type tuple[int or str or None] or None

Methods Documentation

validate (*name*, *x*)
Validate the input tensor *x*.

Parameters

- **name** (*str*) – The name of the tensor, used in error messages.
- **x** – The input tensor.

Returns The validated tensor.

Raises *ValueError, TypeError* – If *x* cannot pass validation.

IntConfigValidator

class tfsnippet.utils.**IntConfigValidator**

Bases: tfsnippet.utils.config_utils.ConfigValidator

Config value validator for integer values.

Methods Summary

validate(value[, strict])	Validate the *value*.

Methods Documentation

validate(*value*, *strict=False*)

Validate the *value*.

Parameters

- **value** – The value to be validated.
- **strict** (*bool*) – If True, disable type conversion. If False, the validator will try its best to convert the input *value* into desired type.

Returns The validated value.

Raises *ValueError, TypeError* – If the value cannot pass validation.

InvertibleMatrix

class tfsnippet.utils.**InvertibleMatrix**(*size*, *strict=False*, *dtype=tf.float32*, *epsilon=1e-06*, *trainable=True*, *random_state=None*, *name=None*, *scope=None*)

Bases: tfsnippet.utils.reuse.VarScopeObject

A matrix initialized to be an invertible, orthogonal matrix.

If *strict* is False, then there is no guarantee that the matrix will keep invertible during training. Otherwise, the matrix will be derived through a variant of PLU decomposition as proposed in (Kingma & Dhariwal, 2018):

$$\mathbf{M} = \mathbf{PL}(\mathbf{U} + \mathrm{diag}(\mathbf{sign} \odot \exp(\mathbf{s})))$$

where *P* is a permutation matrix, *L* is a lower triangular matrix with all its diagonal elements equal to one, *U* is an upper triangular matrix with all its diagonal elements equal to zero, *sign* is a vector of *{-1, 1}*, and *s* is a vector. *P* and *sign* are fixed variables, while *L*, *U*, *s* are trainable variables.

A *random_state* can be specified via the constructor. If it is not specified, an instance of *VarScopeRandomState* will be created according to the variable scope name of the object.

Attributes Summary

inv_matrix	Get the inverted matrix.
log_det	Get the log-determinant of the matrix.
matrix	Get the matrix tensor.
name	Get the name of this object.
shape	Get the shape of the matrix.
variable_scope	Get the variable scope of this object.

Attributes Documentation

inv_matrix
Get the inverted matrix.

Returns The inverted matrix tensor.

Return type tf.Tensor

log_det
Get the log-determinant of the matrix.

Returns The log-determinant tensor.

Return type tf.Tensor

matrix
Get the matrix tensor.

Returns The matrix tensor.

Return type tf.Tensor or tf.Variable

name
Get the name of this object.

shape
Get the shape of the matrix.

Returns The shape of the matrix.

Return type (int, int)

variable_scope
Get the variable scope of this object.

NoReentrantContext

class tfsnippet.utils.**NoReentrantContext**
Bases: object

Base class for contexts which are not reentrant (i.e., if there is a context opened by __enter__, and it has not called __exit__, the __enter__ cannot be called again).

ParamSpec

class tfsnippet.utils.**ParamSpec**(**args*, ***kwargs*)
Bases: tfsnippet.utils.tensor_spec.TensorSpec

Class to describe the specification for a parameter.

Unlike *TensorSpec*, the shape of the parameter must be fully determined, i.e., without any unknown dimension, and the ndims must be identical to the specification.

Attributes Summary

dtype	Get the data type of the tensor.
shape	Get the shape specification.
value_ndims	Get the value shape ndims.
value_shape	Get the value shape (the shape excluding leading "...").

Methods Summary

validate(name, x)	Validate the input tensor *x*.

Attributes Documentation

dtype
Get the data type of the tensor.

Returns The data type, or None.

Return type tf.DType or None

shape
Get the shape specification.

Returns The value shape, or None.

Return type tuple[int or str or None] or None

value_ndims
Get the value shape ndims.

Returns The value shape ndims, or None.

Return type int or None

value_shape
Get the value shape (the shape excluding leading "...").

Returns The value shape, or None.

Return type tuple[int or str or None] or None

Methods Documentation

validate (*name*, *x*)
Validate the input tensor *x*.

Parameters

- **name** (*str*) – The name of the tensor, used in error messages.
- **x** – The input tensor.

Returns The validated tensor.

Raises *ValueError, TypeError* – If *x* cannot pass validation.

PermutationMatrix

class tfsnippet.utils.**PermutationMatrix**(*data*)

Bases: object

A non-trainable permutation matrix.

Attributes Summary

col_permutation	Get the column permutation indices.
row_permutation	Get the row permutation indices.
shape	Get the shape of this permutation matrix.

Methods Summary

det()	Get the determinant of this permutation matrix.
get_numpy_matrix([dtype])	Get the numpy permutation matrix.
inv()	Get the inverse permutation matrix of this matrix.
left_mult(input[, name])	Left multiply to *input* matrix.
right_mult(input[, name])	Right multiply to *input* matrix.

Attributes Documentation

col_permutation

Get the column permutation indices.

Returns The column permutation indices.

Return type tuple[int]

row_permutation

Get the row permutation indices.

Returns The row permutation indices.

Return type tuple[int]

shape

Get the shape of this permutation matrix.

Returns The shape of this permutation matrix.

Return type (int, int)

Methods Documentation

det()

Get the determinant of this permutation matrix.

Returns The determinant of this permutation matrix.

Return type float

get_numpy_matrix(*dtype=<type 'numpy.int32'>*)
Get the numpy permutation matrix.

Parameters **dtype** – The data type of the returned matrix.

Returns A 2-d numpy matrix.

Return type np.ndarray

inv()
Get the inverse permutation matrix of this matrix.

Returns The inverse permutation matrix.

Return type *PermutationMatrix*

left_mult(*input*, *name=None*)
Left multiply to *input* matrix.

output = matmul(self, input)

Parameters

- **input** (*np.ndarray or tf.Tensor*) – The input matrix, whose shape must be `(self.shape[1], ?)`.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The result of multiplication.

Return type np.ndarray or tf.Tensor

right_mult(*input*, *name=None*)
Right multiply to *input* matrix.

output = matmul(input, self)

Parameters

- **input** (*np.ndarray or tf.Tensor*) – The input matrix, whose shape must be `(?, self.shape[0])`.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The result of multiplication.

Return type np.ndarray or tf.Tensor

RarExtractor

class tfsnippet.utils.**RarExtractor**(*fpath*)
Bases: tfsnippet.utils.archive_file.Extractor

Extractor for ".rar" files.

Methods Summary

close()	Close the extractor.
iter_extract()	Extract files from the archive with an iterator.

Continued on next page

Table 172 – continued from previous page

open(file_path)	Create an *Extractor* instance for given archive file.

Methods Documentation

close()
Close the extractor.

iter_extract()
Extract files from the archive with an iterator.

You may simply iterate over a *Extractor* object, which is same as calling to this method.

Yields *(str, file-like)* –

Tuples of (name, file-like object), the filename and corresponding file-like object for each file in the archive. The returned file-like object may or may not be closeable. You may surround it by `maybe_close()`.

static open(*file_path*)
Create an *Extractor* instance for given archive file.

Parameters **file_path** (*str*) – The path of the archive file.

Returns The specified extractor instance.

Return type *Extractor*

Raises `IOError` – If the `file_path` is not a supported archive.

StatisticsCollector

class tfsnippet.utils.**StatisticsCollector**(*shape=()*)
Bases: `object`

Computing $\mathrm{E}[X]$ and $\mathrm{Var}[X]$ online.

Attributes Summary

counter	Get the counter of collected values.
has_value	Whether or not any value has been collected?
mean	Get the mean of the values, i.e., $\mathrm{E}[X]$.
shape	Get the shape of the values.
square	Get $\mathrm{E}[X^2]$ of the values.
stddev	Get the std of the values, i.e., $\sqrt{\mathrm{Var}[X]}$.
var	Get the variance of the values, i.e., $\mathrm{Var}[X]$.
weight_sum	Get the weight summation.

Methods Summary

collect(values[, weight])	Update the statistics from values.
reset()	Reset the collector to initial state.

Attributes Documentation

counter
Get the counter of collected values.

has_value
Whether or not any value has been collected?

mean
Get the mean of the values, i.e., $\mathrm{E}[X]$.

shape
Get the shape of the values.

square
Get $\mathrm{E}[X^2]$ of the values.

stddev
Get the std of the values, i.e., $\sqrt{\mathrm{Var}[X]}$.

var
Get the variance of the values, i.e., $\mathrm{Var}[X]$.

weight_sum
Get the weight summation.

Methods Documentation

collect (*values*, *weight=1.0*)
Update the statistics from values.

This method uses the following equation to update *mean* and *square*:

$$\frac{\sum_{i=1}^{n} w_i f(x_i)}{\sum_{j=1}^{n} w_j} = \frac{\sum_{i=1}^{m} w_i f(x_i)}{\sum_{j=1}^{m} w_j} + \frac{\sum_{i=m+1}^{n} w_i}{\sum_{j=1}^{n} w_j} \left(\frac{\sum_{i=m+1}^{n} w_i f(x_i)}{\sum_{j=m+1}^{n} w_j} - \frac{\sum_{i=1}^{m} w_i f(x_i)}{\sum_{j=1}^{m} w_j} \right)$$

Parameters

- **values** – Values to be collected in batch, numpy array or scalar whose shape ends with `self.shape`. The leading shape in front of `self.shape` is regarded as the batch shape.
- **weight** – Weights of the *values*, should be broadcastable against the batch shape. (default is 1)

Raises `ValueError` – If the shape of *values* does not end with *self.shape*.

reset ()
Reset the collector to initial state.

StrConfigValidator

class tfsnippet.utils.**StrConfigValidator**
Bases: tfsnippet.utils.config_utils.ConfigValidator

Config value validator for string values.

Methods Summary

validate(value[, strict])	Validate the *value*.

Methods Documentation

validate(*value*, *strict=False*)
Validate the *value*.

Parameters

- **value** – The value to be validated.
- **strict** (*bool*) – If True, disable type conversion. If False, the validator will try its best to convert the input *value* into desired type.

Returns The validated value.

Raises *ValueError, TypeError* – If the value cannot pass validation.

SummaryCollector

class tfsnippet.utils.**SummaryCollector**(*collections=None*, *no_add_to_collections=False*)
Bases: object

Collecting summaries and histograms added by *tfsnippet.add_summary()* and *tfsnippet.add_histogram()*. For example:

```
collector = SummaryCollector()
with collector.as_default():
    spt.add_summary(...)
summary_op = collector.merge_summary()
```

You may also use this collector to capture the auto histogram generated by layers from *tfsnippet.layers*, for example:

```
collector = SummaryCollector()
with collector.as_default(auto_histogram=True):
    y = spt.layers.dense(x, ...)
summary_op = collector.merge_summary()
```

Attributes Summary

collections	Get the summary collections.
summary_list	Get the list of captured summaries.

Methods Summary

add_histogram(tensor[, summary_name, ...])	Add the histogram of *tensor* to this collector and to *collections*.
add_summary(summary[, collections])	Add the summary to this collector and to *collections*.

Continued on next page

Table 177 – continued from previous page

as_default(**kwds)	Push this *SummaryCollector* to the top of context stack, and enter a scoped context.
merge_summary()	Merge all the captured summaries into one operation.

Attributes Documentation

collections
Get the summary collections.

summary_list
Get the list of captured summaries.

Methods Documentation

add_histogram(*tensor*, *summary_name=None*, *strip_scope=False*, *collections=None*, *name=None*)
Add the histogram of *tensor* to this collector and to *collections*.

Parameters

- **tensor** – Take histogram of this tensor.
- **summary_name** – Specify the summary name for *tensor*.
- **strip_scope** – If True, strip the name scope from *tensor.name* when adding the histogram.
- **collections** – Also add the histogram to these collections. Defaults to *self.collections*.
- **name** (*str*) – Default name of the name scope. If not specified, generate one according to the method name.

Returns The serialized histogram tensor of *tensor*.

add_summary(*summary*, *collections=None*)
Add the summary to this collector and to *collections*.

Parameters

- **summary** – TensorFlow summary tensor.
- **collections** – Also add the summary to these collections. Defaults to *self.collections*.

Returns The *summary* tensor.

as_default(***kwds*)
Push this *SummaryCollector* to the top of context stack, and enter a scoped context.

Parameters **auto_histogram** (*bool*) – If specified, set the config value of *tfsnippet.settings.auto_histogram* within the context.

Yields This summary collector object.

merge_summary()
Merge all the captured summaries into one operation.

Returns

The merged operation, or None if no summary has been captured.

Return type tf.Operation or None

TFSnippetConfig

class tfsnippet.utils.**TFSnippetConfig**
Bases: tfsnippet.utils.config_utils.Config

Global configurations of TFSnippet.

Attributes Summary

auto_histogram
check_numerics
enable_assertions
file_cache_checksum

Methods Summary

to_dict()	Get the config values as a dict.
update(key_values)	Update the config values from *key_values*.

Attributes Documentation

auto_histogram = <tfsnippet.utils.config_utils.ConfigField object>

check_numerics = <tfsnippet.utils.config_utils.ConfigField object>

enable_assertions = <tfsnippet.utils.config_utils.ConfigField object>

file_cache_checksum = <tfsnippet.utils.config_utils.ConfigField object>

Methods Documentation

to_dict()
Get the config values as a dict.

Returns The config values.

Return type dict[str, any]

update(*key_values*)
Update the config values from *key_values*.

Parameters **key_values** – A dict, or a sequence of (key, value) pairs.

TarExtractor

class tfsnippet.utils.**TarExtractor**(*fpath*)
Bases: tfsnippet.utils.archive_file.Extractor

Extractor for ".tar", ".tar.gz", ".tgz", ".tar.bz2", ".tbz", ".tbz2", ".tb2", ".tar.xz", ".txz" files.

Methods Summary

close()	Close the extractor.
iter_extract()	Extract files from the archive with an iterator.
open(file_path)	Create an *Extractor* instance for given archive file.

Methods Documentation

close()
Close the extractor.

iter_extract()
Extract files from the archive with an iterator.

You may simply iterate over a *Extractor* object, which is same as calling to this method.

Yields *(str, file-like)* –

Tuples of (name, file-like object), the filename and corresponding file-like object for each file in the archive. The returned file-like object may or may not be closeable. You may surround it by `maybe_close()`.

static open(*file_path*)
Create an *Extractor* instance for given archive file.

Parameters **file_path** (*str*) – The path of the archive file.

Returns The specified extractor instance.

Return type *Extractor*

Raises `IOError` – If the `file_path` is not a supported archive.

TensorArgValidator

class tfsnippet.utils.**TensorArgValidator**(*name*)
Bases: `object`

Class to validate argument values of tensors.

Methods Summary

require_int32(value)	Require *value* to be an 32-bit integer.
require_non_negative(value)	Require *value* to be non-negative, i.e., `value >= 0`.
require_positive(value)	Require *value* to be positive, i.e., `value > 0`.

Methods Documentation

require_int32(*value*)
Require *value* to be an 32-bit integer.

Parameters **value** – Value to be validated. If `is_tensor_object(value) == True`, it will be casted into a `tf.Tensor` with *dtype* as `tf.int32`. If otherwise `is_integer(value) == True`, the type will not be casted, but its value will be checked to ensure it falls between *-2**31 ~ 2**31-1*.

Returns The validated value.

Raises `TypeError` – If specified *value* cannot be casted into int32, or the value is out of range.

require_non_negative(*value*)
Require *value* to be non-negative, i.e., `value >= 0`.

Parameters **value** – Value to be validated. If `is_tensor_object(value) == True`, additional assertion will be added to *value*. Otherwise it will be validated against `value >= 0` immediately.

Returns The validated value.

Raises `ValueError` – If specified *value* is not non-negative.

require_positive(*value*)
Require *value* to be positive, i.e., `value > 0`.

Parameters **value** – Value to be validated. If `is_tensor_object(value) == True`, additional assertion will be added to *value*. Otherwise it will be validated against `value > 0` immediately.

Returns The validated value.

Raises `ValueError` – If specified *value* is not non-negative.

TensorSpec

class tfsnippet.utils.**TensorSpec**(*shape=None*, *dtype=None*)
Bases: `object`

Base class to describe and validate the specification of a tensor.

Attributes Summary

dtype	Get the data type of the tensor.
shape	Get the shape specification.
value_ndims	Get the value shape ndims.
value_shape	Get the value shape (the shape excluding leading "...").

Methods Summary

validate(name, x)	Validate the input tensor *x*.

Attributes Documentation

dtype
Get the data type of the tensor.

Returns The data type, or None.

Return type tf.DType or None

shape
Get the shape specification.

Returns The value shape, or None.

Return type tuple[int or str or None] or None

value_ndims
Get the value shape ndims.

Returns The value shape ndims, or None.

Return type int or None

value_shape
Get the value shape (the shape excluding leading "...").

Returns The value shape, or None.

Return type tuple[int or str or None] or None

Methods Documentation

validate(*name*, *x*)
Validate the input tensor *x*.

Parameters

- **name** (*str*) – The name of the tensor, used in error messages.
- **x** – The input tensor.

Returns The validated tensor.

Raises *ValueError, TypeError* – If *x* cannot pass validation.

TensorWrapper

class tfsnippet.utils.**TensorWrapper**
Bases: object

Tensor-like object that wraps a *tf.Tensor* instance.

This class is typically used to implement *super-tensor* classes, adding auxiliary methods to a `tf.Tensor`. The derived classes should call *register_rensor_wrapper* to register themselves into TensorFlow type system.

Access to any undefined attributes, properties and methods will be transparently proxied to the wrapped tensor. Also, *TensorWrapper* can be directly used in mathematical expressions and most TensorFlow arithmetic functions. For example, `TensorWrapper(...) + tf.exp(TensorWrapper(...))`.

On the other hand, *TensorWrapper* are neither `tf.Tensor` nor sub-classes of `tf.Tensor`, i.e., `isinstance(TensorWrapper(...), tf.Tensor) == False`. This is essential for sub-classes of *TensorWrapper* being converted correctly to `tf.Tensor` by `tf.convert_to_tensor()`, using the official type conversion system of TensorFlow.

All the attributes defined in sub-classes of *TensorWrapper* must have names starting with `_self_`. The properties and methods are not restricted by this rule.

An example of inheriting *TensorWrapper* is shown as follows:

```
class MyTensorWrapper(TensorWrapper):

    def __init__(self, wrapped, flag):
        super(MyTensorWrapper, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        self._self_wrapped = wrapped
        self._self_flag = flag

    @property
    def tensor(self):
        return self._self_wrapped

    @property
    def flag(self):
        return self._self_flag

register_tensor_wrapper_class(MyTensorWrapper)

# tests
t = MyTensorWrapper(tf.constant(0., dtype=tf.float32), flag=123)
assert(t.dtype == tf.float32)
assert(t.flag == 123)
```

Attributes Summary

tensor	Get the wrapped `tf.Tensor`.

Attributes Documentation

tensor

Get the wrapped `tf.Tensor`. Derived classes must override this to return the actual wrapped tensor.

Returns The wrapped tensor.

Return type tf.Tensor

VarScopeObject

class tfsnippet.utils.**VarScopeObject**(*name=None*, *scope=None*)

Bases: object

Base class for objects that own a variable scope.

The *VarScopeObject* can be used along with *instance_reuse()*, for example:

```
class YourVarScopeObject(VarScopeObject):

    @instance_reuse
    def foo(self):
        return tf.get_variable('bar', ...)

o = YourVarScopeObject('object_name')
o.foo()  # You should get a variable with name "object_name/foo/bar"
```

To build variables in the constructor of derived classes, you may use `reopen_variable_scope(self.variable_scope)` to open the original variable scope and its name scope, right after the constructor of *VarScopeObject* has been called, for example:

```
class YourVarScopeObject(VarScopeObject):

    def __init__(self, name=None, scope=None):
        super(YourVarScopeObject, self).__init__(name=name, scope=scope)
        with reopen_variable_scope(self.variable_scope):
            self.w = tf.get_variable('w', ...)
```

See also:

tfsnippet.utils.instance_reuse().

Attributes Summary

name	Get the name of this object.
variable_scope	Get the variable scope of this object.

Attributes Documentation

name
Get the name of this object.

variable_scope
Get the variable scope of this object.

VarScopeRandomState

class tfsnippet.utils.**VarScopeRandomState**(*variable_scope*)
Bases: mtrand.RandomState

A sub-class of np.random.RandomState, which uses a variable-scope dependent seed. It is guaranteed for a *VarScopeRandomState* initialized with the same global seed and variable scopes with the same name to produce exactly the same random sequence.

Attributes Summary

poisson_lam_max	

Methods Summary

beta(a, b[, size])	Draw samples from a Beta distribution.
binomial(n, p[, size])	Draw samples from a binomial distribution.
bytes(length)	Return random bytes.
chisquare(df[, size])	Draw samples from a chi-square distribution.
choice(a[, size, replace, p])	Generates a random sample from a given 1-D array
dirichlet(alpha[, size])	Draw samples from the Dirichlet distribution.
exponential([scale, size])	Draw samples from an exponential distribution.
f(dfnum, dfden[, size])	Draw samples from an F distribution.
gamma(shape[, scale, size])	Draw samples from a Gamma distribution.
geometric(p[, size])	Draw samples from the geometric distribution.

Continued on next page

Table 187 – continued from previous page

get_state()	Return a tuple representing the internal state of the generator.
gumbel([loc, scale, size])	Draw samples from a Gumbel distribution.
hypergeometric(ngood, nbad, nsample[, size])	Draw samples from a Hypergeometric distribution.
laplace([loc, scale, size])	Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).
logistic([loc, scale, size])	Draw samples from a logistic distribution.
lognormal([mean, sigma, size])	Draw samples from a log-normal distribution.
logseries(p[, size])	Draw samples from a logarithmic series distribution.
multinomial(n, pvals[, size])	Draw samples from a multinomial distribution.
multivariate_normal(mean, cov[, size, ...)	Draw random samples from a multivariate normal distribution.
negative_binomial(n, p[, size])	Draw samples from a negative binomial distribution.
noncentral_chisquare(df, nonc[, size])	Draw samples from a noncentral chi-square distribution.
noncentral_f(dfnum, dfden, nonc[, size])	Draw samples from the noncentral F distribution.
normal([loc, scale, size])	Draw random samples from a normal (Gaussian) distribution.
pareto(a[, size])	Draw samples from a Pareto II or Lomax distribution with specified shape.
permutation(x)	Randomly permute a sequence, or return a permuted range.
poisson([lam, size])	Draw samples from a Poisson distribution.
power(a[, size])	Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
rand(d0, d1, ..., dn)	Random values in a given shape.
randint(low[, high, size, dtype])	Return random integers from *low* (inclusive) to *high* (exclusive).
randn(d0, d1, ..., dn)	Return a sample (or samples) from the "standard normal" distribution.
random_integers(low[, high, size])	Random integers of type np.int between *low* and *high*, inclusive.
random_sample([size])	Return random floats in the half-open interval [0.0, 1.0).
rayleigh([scale, size])	Draw samples from a Rayleigh distribution.
seed([seed])	Seed the generator.
set_global_seed(seed)	Set the global random seed for all new *VarScopeRandomState*.
set_state(state)	Set the internal state of the generator from a tuple.
shuffle(x)	Modify a sequence in-place by shuffling its contents.
standard_cauchy([size])	Draw samples from a standard Cauchy distribution with mode = 0.
standard_exponential([size])	Draw samples from the standard exponential distribution.
standard_gamma(shape[, size])	Draw samples from a standard Gamma distribution.
standard_normal([size])	Draw samples from a standard Normal distribution (mean=0, stdev=1).
standard_t(df[, size])	Draw samples from a standard Student's t distribution with *df* degrees of freedom.

Continued on next page

Table 187 – continued from previous page

tomaxint([size])	Random integers between 0 and `sys.maxint`, inclusive.
triangular(left, mode, right[, size])	Draw samples from the triangular distribution over the interval `[left, right]`.
uniform([low, high, size])	Draw samples from a uniform distribution.
vonmises(mu, kappa[, size])	Draw samples from a von Mises distribution.
wald(mean, scale[, size])	Draw samples from a Wald, or inverse Gaussian, distribution.
weibull(a[, size])	Draw samples from a Weibull distribution.
zipf(a[, size])	Draw samples from a Zipf distribution.

Attributes Documentation

poisson_lam_max = 9.223372006484771e+18

Methods Documentation

beta(*a*, *b*, *size=None*)

Draw samples from a Beta distribution.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1} dt.$$

It is often seen in Bayesian inference and order statistics.

Parameters

- **a** (*float or array_like of floats*) – Alpha, positive (>0).
- **b** (*float or array_like of floats*) – Beta, positive (>0).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` and `b` are both scalars. Otherwise, `np.broadcast(a, b).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized beta distribution.

Return type ndarray or scalar

binomial(*n*, *p*, *size=None*)

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

Parameters

- **n** (*int or array_like of ints*) – Parameter of the distribution, >= 0. Floats are also accepted, but they will be truncated to integers.

- **p** (*float or array_like of floats*) – Parameter of the distribution, >= 0 and <=1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the n trials.

Return type ndarray or scalar

See also:

scipy.stats.binom() probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

References

Examples

Draw samples from the distribution:

```
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

bytes(*length*)

Return random bytes.

Parameters **length** (*int*) – Number of random bytes.

Returns **out** – String of length *length*.

Return type str

Examples

```
>>> np.random.bytes(10)
' eh\x85\x022SZ\xbf\xa4' #random
```

chisquare(*df*, *size=None*)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

Parameters

- **df** (*float or array_like of floats*) – Number of degrees of freedom, should be > 0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized chi-square distribution.

Return type ndarray or scalar

Raises `ValueError` – When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

Notes

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

References

Examples

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

choice(*a*, *size=None*, *replace=True*, *p=None*)
Generates a random sample from a given 1-D array

New in version 1.7.0.

Parameters

- **a** (*1-D array-like or int*) – If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a were np.arange(a)
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.
- **replace** (*boolean, optional*) – Whether the sample is with or without replacement
- **p** (*1-D array-like, optional*) – The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

Returns **samples** – The generated random samples

Return type single item or ndarray

Raises `ValueError` – If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

See also:

randint(), *shuffle()*, *permutation()*

Examples

Generate a uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3)
array([0, 3, 4])
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0])
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False)
array([3,1,0])
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0])
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'],
      dtype='|S11')
```

dirichlet(*alpha*, *size=None*)

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. Dirichlet pdf is the conjugate prior of a multinomial in Bayesian inference.

Parameters

- **alpha** (*array*) – Parameter of the distribution (k dimension for sample of dimension k).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns **samples** – The drawn samples, of shape (size, alpha.ndim).

Return type ndarray,

Raises `ValueError` – If any value in alpha is less than or equal to zero

Notes

$$X \approx \prod_{i=1}^{k} x_i^{\alpha_i - 1}$$

Uses the following property for computation: for each dimension, draw a random sample y_i from a standard gamma generator of shape *alpha_i*, then $X = \frac{1}{\sum_{i=1}^{k} y_i}(y_1, \ldots, y_n)$ is Dirichlet distributed.

References

Examples

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```
>>> s = np.random.dirichlet((10, 5, 3), 20).transpose()
```

```
>>> import matplotlib.pyplot as plt
>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
```

exponential(*scale=1.0*, *size=None*)

Draw samples from an exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3]_.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

Parameters

- **scale** (*float or array_like of floats*) – The scale parameter, $\beta = 1/\lambda$.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized exponential distribution.

Return type ndarray or scalar

References

f (*dfnum*, *dfden*, *size=None*)

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

Parameters

- **dfnum** (*float or array_like of floats*) – Degrees of freedom in numerator, should be > 0.
- **dfden** (*float or array_like of float*) – Degrees of freedom in denominator, should be > 0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum` and `dfden` are both scalars. Otherwise, `np.broadcast(dfnum, dfden).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Fisher distribution.

Return type ndarray or scalar

See also:

scipy.stats.f() probability density function, distribution or cumulative density function, etc.

Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

References

Examples

An example from Glantz[1], pp 47-40:

Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is :

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

gamma (*shape*, *scale=1.0*, *size=None*)
Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

Parameters

- **shape** (*float or array_like of floats*) – The shape of the gamma distribution. Should be greater than zero.
- **scale** (*float or array_like of floats, optional*) – The scale of the gamma distribution. Should be greater than zero. Default is equal to 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` and `scale` are both scalars. Otherwise, `np.broadcast(shape, scale).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized gamma distribution.

Return type ndarray or scalar

See also:

scipy.stats.gamma() probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 2.  # mean=4, std=2*sqrt(2)
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

geometric(*p*, *size=None*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where *p* is the probability of success of an individual trial.

Parameters

- **p** (*float or array_like of floats*) – The probability of success of an individual trial.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized geometric distribution.

Return type ndarray or scalar

Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

get_state()

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

Returns

out – The returned tuple has the following items:

1. the string 'MT19937'.
2. a 1-D array of 624 unsigned integer keys.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

Return type tuple(str, ndarray of 624 uints, int, int, float)

See also:

set_state()

Notes

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

gumbel(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

Parameters

- **loc** (*float or array_like of floats, optional*) – The location of the mode of the distribution. Default is 0.
- **scale** (*float or array_like of floats, optional*) – The scale parameter of the distribution. Default is 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Gumbel distribution.

Return type ndarray or scalar

See also:

`scipy.stats.gumbel_l()`, `scipy.stats.gumbel_r()`, `scipy.stats.genextreme()`, *`weibull()`*

Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

References

Examples

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = np.random.normal(mu, beta, 1000)
...    means.append(a.mean())
```

(continues on next page)

(continued from previous page)

```
...     maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, density=True)
>>> beta = np.std(maxima) * np.sqrt(6) / np.pi
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...           * np.exp(-np.exp(-(bins - mu)/beta)),
...           linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...           * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...           linewidth=2, color='g')
>>> plt.show()
```

hypergeometric (*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

Parameters

- **ngood** (*int or array_like of ints*) – Number of ways to make a good selection. Must be nonnegative.
- **nbad** (*int or array_like of ints*) – Number of ways to make a bad selection. Must be nonnegative.
- **nsample** (*int or array_like of ints*) – Number of items sampled. Must be at least 1 and at most `ngood + nbad`.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `ngood`, `nbad`, and `nsample` are all scalars. Otherwise, `np.broadcast(ngood, nbad, nsample).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized hypergeometric distribution.

Return type ndarray or scalar

See also:

scipy.stats.hypergeom() probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{g}{x}\binom{b}{n-x}}{\binom{g+b}{n}},$$

where $0 \leq x \leq n$ and $n - b \leq x \leq g$

for P(x) the probability of x successes, g = ngood, b = nbad, and n = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

References

Examples

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> from matplotlib.pyplot import hist
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

laplace(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

Parameters

- **loc** (*float or array_like of floats, optional*) – The position, μ, of the distribution peak. Default is 0.
- **scale** (*float or array_like of floats, optional*) – λ, the exponential decay. Default is 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Laplace distribution.

Return type ndarray or scalar

Notes

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x-\mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

References

Examples

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```
>>> g = (1/(scale * np.sqrt(2 * np.pi)) *
...      np.exp(-(x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x,g)
```

logistic(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a logistic distribution.

Samples are drawn from a logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

Parameters

- **loc** (*float or array_like of floats, optional*) – Parameter of the distribution. Default is 0.
- **scale** (*float or array_like of floats, optional*) – Parameter of the distribution. Should be greater than zero. Default is 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized logistic distribution.

Return type ndarray or scalar

See also:

scipy.stats.logistic() probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

References

Examples

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...     return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

lognormal (*mean=0.0*, *sigma=1.0*, *size=None*)
Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

Parameters

- **mean** (*float or array_like of floats, optional*) – Mean value of the underlying normal distribution. Default is 0.
- **sigma** (*float or array_like of floats, optional*) – Standard deviation of the underlying normal distribution. Should be greater than zero. Default is 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `sigma` are both scalars. Otherwise, `np.broadcast(mean, sigma).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized log-normal distribution.

Return type ndarray or scalar

See also:

scipy.stats.lognorm() probability density function, distribution, cumulative density function, etc.

Notes

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x\sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

References

Examples

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, density=True, align='mid')
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...     a = 10. + np.random.random(100)
...     b.append(np.product(a))
```

```
>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, density=True, align='mid')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```

logseries(*p*, *size=None*)

Draw samples from a logarithmic series distribution.

Samples are drawn from a log series distribution with specified shape parameter, 0 < p < 1.

Parameters

- **p** (*float or array_like of floats*) – Shape parameter for the distribution. Must be in the range (0, 1).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized logarithmic series distribution.

Return type ndarray or scalar

See also:

scipy.stats.logser() probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The log series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

References

Examples

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s)
```

plot against distribution

```
>>> def logseries(k, p):
...     return -p**k/(k*log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/
             logseries(bins, a).max(), 'r')
>>> plt.show()
```

multinomial(*n*, *pvals*, *size=None*)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.

Parameters

- **n** (*int*) – Number of experiments.
- **pvals** (*sequence of floats, length p*) – Probabilities of each of the `p` different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out – The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Return type ndarray

Examples

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3],
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded die is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5 + [2/7.])
array([11, 16, 14, 17, 16, 26])
```

The probability inputs should be normalized. As an implementation detail, the value of the last entry is ignored and assumed to take up any leftover probability mass, but this should not be relied on. A biased coin which has twice as much weight on one side as on the other should be sampled like so:

```
>>> np.random.multinomial(100, [1.0 / 3, 2.0 / 3])  # RIGHT
array([38, 62])
```

not like:

```
>>> np.random.multinomial(100, [1.0, 2.0])  # WRONG
array([100,   0])
```

multivariate_normal (*mean*, *cov*[, *size*, *check_valid*, *tol*])
Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

Parameters

- **mean** (*1-D array_like, of length N*) – Mean of the N-dimensional distribution.
- **cov** (*2-D array_like, of shape (N, N)*) – Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.
- **size** (*int or tuple of ints, optional*) – Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.
- **check_valid** (*{ 'warn', 'raise', 'ignore' }, optional*) – Behavior when the covariance matrix is not positive semidefinite.
- **tol** (*float, optional*) – Tolerance when checking the singular values in covariance matrix. cov is cast to double before the check.

Returns

out – The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Return type ndarray

Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

References

Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True]
```

negative_binomial(*n*, *p*, *size=None*)

Draw samples from a negative binomial distribution.

Samples are drawn from a negative binomial distribution with specified parameters, *n* successes and *p* probability of success where *n* is an integer > 0 and *p* is in the interval [0, 1].

Parameters

- **n** (*int or array_like of ints*) – Parameter of the distribution, > 0. Floats are also accepted, but they will be truncated to integers.
- **p** (*float or array_like of floats*) – Parameter of the distribution, >= 0 and <=1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

Returns out – Drawn samples from the parameterized negative binomial distribution, where each sample is equal to N, the number of failures that occurred before a total of n successes was reached.

Return type ndarray or scalar

Notes

The probability density for the negative binomial distribution is

$$P(N;n,p) = \binom{N+n-1}{N} p^n (1-p)^N,$$

where n is the number of successes, p is the probability of success, and $N + n$ is the number of trials. The negative binomial distribution gives the probability of N failures given n successes, with a success on the last trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

References

Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...    probability = sum(s<i) / 100000.
...    print i, "wells drilled, probability of one success =", probability
```

noncentral_chisquare (*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

Parameters

- **df** (*float or array_like of floats*) – Degrees of freedom, should be > 0.

  Changed in version 1.10.0: Earlier NumPy versions required dfnum > 1.

- **nonc** (*float or array_like of floats*) – Non-centrality, should be non-negative.

- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` and `nonc` are both scalars. Otherwise, `np.broadcast(df, nonc).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized noncentral chi-square distribution.

Return type ndarray or scalar

Notes

The probability density function for the noncentral Chi-square distribution is

$$P(x;df,nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} \P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

References

Examples

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), density=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), density=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

noncentral_f (*dfnum*, *dfden*, *nonc*, *size=None*)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

Parameters

- **dfnum** (*float or array_like of floats*) – Numerator degrees of freedom, should be > 0.

  Changed in version 1.14.0: Earlier NumPy versions required dfnum > 1.

- **dfden** (*float or array_like of floats*) – Denominator degrees of freedom, should be > 0.

- **nonc** (*float or array_like of floats*) – Non-centrality parameter, the sum of the squares of the numerator means, should be >= 0.

- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum`, `dfden`, and `nonc` are all scalars. Otherwise, `np.broadcast(dfnum, dfden, nonc).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized noncentral Fisher distribution.

Return type ndarray or scalar

Notes

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

References

Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, density=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, density=True)
>>> import matplotlib.pyplot as plt
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

normal (*loc=0.0*, *scale=1.0*, *size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2]_.

Parameters

- **loc** (*float or array_like of floats*) – Mean ("centre") of the distribution.
- **scale** (*float or array_like of floats*) – Standard deviation (spread or "width") of the distribution.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized normal distribution.

Return type ndarray or scalar

See also:

scipy.stats.norm() probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

References

Examples

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
```

```
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```

pareto(*a*, *size=None*)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

Parameters

- **a** (*float or array_like of floats*) – Shape of the distribution. Should be greater than zero.

- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Pareto distribution.

Return type ndarray or scalar

See also:

scipy.stats.lomax() probability density function, distribution or cumulative density function, etc.

scipy.stats.genpareto() probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]_. It is one of the so-called "fat-tailed" distributions.

References

Examples

Draw samples from the distribution:

```
>>> a, m = 3., 2.  # shape and mode
>>> s = (np.random.pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, _ = plt.hist(s, 100, density=True)
>>> fit = a*m**a / bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')
>>> plt.show()
```

permutation(*x*)

Randomly permute a sequence, or return a permuted range.

If *x* is a multi-dimensional array, it is only shuffled along its first index.

Parameters **x** (*int or array_like*) – If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

Returns **out** – Permuted sequence or array range.

Return type ndarray

Examples

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])
```

```
>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12])
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
```

poisson (*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

Parameters

- **lam** (*float or array_like of floats*) – Expectation of interval, should be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `np.array(lam).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Poisson distribution.

Return type ndarray or scalar

Notes

The Poisson distribution

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ.

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

References

Examples

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, density=True)
>>> plt.show()
```

Draw each 100 values for lambda 100 and 500:

```
>>> s = np.random.poisson(lam=(100., 500.), size=(100, 2))
```

power(*a*, *size=None*)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

Parameters

- **a** (*float or array_like of floats*) – Parameter of the distribution. Should be greater than zero.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized power distribution.

Return type ndarray or scalar

Raises `ValueError` – If a < 1.

Notes

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \leq x \leq 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

References

Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
```

(continues on next page)

(continued from previous page)

```
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

```
>>> plt.figure()
>>> plt.hist(rvs, bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```

rand(*d0*, *d1*, ..., *dn*)

Random values in a given shape.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

Parameters **d1, .., dn** (*d0,*) – The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

Returns **out** – Random values.

Return type ndarray, shape `(d0, d1, ..., dn)`

See also:

`random()`

Notes

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample .

Examples

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

randint(*low*, *high=None*, *size=None*, *dtype='l'*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Parameters

- **low** (*int*) – Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).
- **high** (*int, optional*) – If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.
- **dtype** (*dtype, optional*) – Desired dtype of the result. All dtypes are determined by their name, i.e., 'int64', 'int', etc, so byteorder is not available and a specific precision may have different C types depending on the platform. The default value is 'np.int'.

  New in version 1.11.0.

Returns **out** – *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

Return type int or ndarray of ints

See also:

random.random_integers() similar to *randint*, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

randn(*d0*, *d1*, ..., *dn*)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use *numpy.random.standard_normal* instead.

Parameters **d1, .., dn** (*d0,*) – The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.

Returns **Z** – A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

Return type ndarray or float

See also:

standard_normal() Similar, but takes a tuple as its argument.

Notes

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
```

Examples

```
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],  #random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]]) #random
```

random_integers (*low*, *high=None*, *size=None*)

Random integers of type np.int between *low* and *high*, inclusive.

Return random integers of type np.int from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*]. The np.int type translates to the C long type used by Python 2 for "short" integers and its precision is platform dependent.

This function has been deprecated. Use randint instead.

Deprecated since version 1.11.0.

Parameters

- **low** (*int*) – Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).
- **high** (*int, optional*) – If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns **out** – *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

Return type int or ndarray of ints

See also:

randint() Similar to *random_integers*, only for the half-open interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

Notes

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

Examples

```
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3,2))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, density=True)
>>> plt.show()
```

random_sample(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add a:

```
(b - a) * random_sample() + a
```

Parameters **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns **out** – Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Return type float or ndarray of floats

Examples

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

rayleigh(*scale=1.0*, *size=None*)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

Parameters

- **scale** (*float or array_like of floats, optional*) – Scale, also equals the mode. Should be >= 0. Default is 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Rayleigh distribution.

Return type ndarray or scalar

Notes

The probability density function for the Rayleigh distribution is

$$P(x;scale) = \frac{x}{scale^2}e^{\frac{-x^2}{2\cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

References

Examples

Draw values from the distribution and plot the histogram

```
>>> from matplotlib.pyplot import hist
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, density=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003
```

seed(*seed=None*)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

Parameters **seed** (*int or 1-d array_like, optional*) – Seed for *RandomState*. Must be convertible to 32 bit unsigned integers.

See also:

`RandomState()`

classmethod set_global_seed(*seed*)

Set the global random seed for all new *VarScopeRandomState*.

If not set, the default global random seed is *0*.

Parameters **seed** (*int*) – The global random seed.

set_state(*state*)

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

Parameters **state** (*tuple(str, ndarray of 624 uints, int, int, float)*) – The *state* tuple has the following items:

1. the string 'MT19937', specifying the Mersenne Twister algorithm.
2. a 1-D array of 624 unsigned integers `keys`.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

Returns **out** – Returns 'None' on success.

Return type None

See also:

get_state()

Notes

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: `state = ('MT19937', keys, pos)`.

References

shuffle(*x*)

Modify a sequence in-place by shuffling its contents.

This function only shuffles the array along the first axis of a multi-dimensional array. The order of sub-arrays is changed but their contents remains the same.

Parameters **x** (*array_like*) – The array or list to be shuffled.

Returns

Return type None

Examples

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

Multi-dimensional arrays are only shuffled along the first axis:

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5],
       [6, 7, 8],
       [0, 1, 2]])
```

standard_cauchy(*size=None*)

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

Parameters **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns **samples** – The drawn samples.

Return type ndarray or scalar

Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi\gamma\left[1 + (\frac{x-x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

References

Examples

Draw samples and plot the distribution:

```
>>> import matplotlib.pyplot as plt
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

standard_exponential (*size=None*)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

Parameters **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns **out** – Drawn samples.

Return type float or ndarray

Examples

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

standard_gamma (*shape*, *size=None*)

Draw samples from a standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

Parameters

- **shape** (*float or array_like of floats*) – Parameter, should be > 0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` is a scalar. Otherwise, `np.array(shape).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized standard gamma distribution.

Return type ndarray or scalar

See also:

scipy.stats.gamma() probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/ \
...                       (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

standard_normal (*size=None*)

Draw samples from a standard Normal distribution (mean=0, stdev=1).

Parameters **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns **out** – Drawn samples.

Return type float or ndarray

Examples

```
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311, #random
       -0.38672696, -0.4685006 ])                                #random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

standard_t (*df*, *size=None*)

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

Parameters

- **df** (*float or array_like of floats*) – Degrees of freedom, should be > 0.

- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized standard Student's t distribution.

Return type ndarray or scalar

Notes

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\,\Gamma(\frac{df}{2})}\left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gosset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

References

Examples

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in kilojoules (kJ) is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, density=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> np.sum(s<t) / float(len(s))
0.0090699999999999999  #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

tomaxint(*size=None*)

Random integers between 0 and `sys.maxint`, inclusive.

Return a sample of uniformly distributed random integers in the interval [0, `sys.maxint`].

Parameters **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns **out** – Drawn samples, with shape *size*.

Return type ndarray

See also:

randint() Uniform sampling over a given half-open interval of integers.

random_integers() Uniform sampling over a given closed interval of integers.

Examples

```
>>> RS = np.random.mtrand.RandomState() # need a RandomState object
>>> RS.tomaxint((2,2,2))
array([[[1170048599, 1600360186],
        [ 739731006, 1947757578]],
       [[1871712945,  752307660],
        [1601631370, 1479324245]]])
>>> import sys
>>> sys.maxint
2147483647
>>> RS.tomaxint((2,2,2)) < sys.maxint
array([[[ True,  True],
        [ True,  True]],
       [[ True,  True],
        [ True,  True]]])
```

triangular(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution over the interval `[left, right]`.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

Parameters

- **left** (*float or array_like of floats*) – Lower limit.
- **mode** (*float or array_like of floats*) – The value where the peak of the distribution occurs. The value should fulfill the condition `left <= mode <= right`.
- **right** (*float or array_like of floats*) – Upper limit, should be larger than *left*.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `left`, `mode`, and `right` are all scalars. Otherwise, `np.broadcast(left, mode, right).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized triangular distribution.

Return type ndarray or scalar

Notes

The probability density function for the triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

References

Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
...              density=True)
>>> plt.show()
```

uniform(*low=0.0*, *high=1.0*, *size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters

- **low** (*float or array_like of floats, optional*) – Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.
- **high** (*float or array_like of floats*) – Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized uniform distribution.

Return type ndarray or scalar

See also:

randint() Discrete uniform distribution, yielding integers.

random_integers() Discrete uniform distribution over the closed interval `[low, high]`.

random_sample() Floats uniformly distributed over `[0, 1)`.

random() Alias for *random_sample*.

rand() Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

vonmises (*mu*, *kappa*, *size=None*)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

Parameters

- **mu** (*float or array_like of floats*) – Mode ("center") of the distribution.
- **kappa** (*float or array_like of floats*) – Dispersion of the distribution, has to be >=0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mu` and `kappa` are both scalars. Otherwise, `np.broadcast(mu, kappa).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized von Mises distribution.

Return type ndarray or scalar

See also:

scipy.stats.vonmises() probability density function, distribution, or cumulative density function, etc.

Notes

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

References

Examples

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import i0
>>> plt.hist(s, 50, density=True)
>>> x = np.linspace(-np.pi, np.pi, num=51)
>>> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa))
>>> plt.plot(x, y, linewidth=2, color='r')
>>> plt.show()
```

wald(*mean*, *scale*, *size=None*)

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Parameters

- **mean** (*float or array_like of floats*) – Distribution mean, should be > 0.
- **scale** (*float or array_like of floats*) – Scale parameter, should be >= 0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `scale` are both scalars. Otherwise, `np.broadcast(mean, scale).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Wald distribution.

Return type ndarray or scalar

Notes

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

References

Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, density=True)
>>> plt.show()
```

weibull(*a*, *size=None*)

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

Parameters

- **a** (*float or array_like of floats*) – Shape parameter of the distribution. Must be nonnegative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Weibull distribution.

Return type ndarray or scalar

See also:

`scipy.stats.weibull_max()`, `scipy.stats.weibull_min()`, `scipy.stats.genextreme()`, *gumbel()*

Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

References

Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
```

```
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

zipf (*a*, *size=None*)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter $a > 1$.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

Parameters

- **a** (*float or array_like of floats*) – Distribution parameter. Should be greater than 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

Returns **out** – Drawn samples from the parameterized Zipf distribution.

Return type ndarray or scalar

See also:

scipy.stats.zipf() probability density function, distribution, or cumulative density function, etc.

Notes

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

References

Examples

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy import special
```

Truncate s values at 50 so plot is interesting:

```
>>> count, bins, ignored = plt.hist(s[s<50], 50, density=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a) / special.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

ZipExtractor

class tfsnippet.utils.**ZipExtractor**(*fpath*)

Bases: tfsnippet.utils.archive_file.Extractor

Extractor for ".zip" files.

Methods Summary

close()	Close the extractor.
iter_extract()	Extract files from the archive with an iterator.
open(file_path)	Create an *Extractor* instance for given archive file.

Methods Documentation

close()

Close the extractor.

iter_extract()
Extract files from the archive with an iterator.

You may simply iterate over a *Extractor* object, which is same as calling to this method.

Yields *(str, file-like)* –

Tuples of (name, file-like object), the filename and corresponding file-like object for each file in the archive. The returned file-like object may or may not be closeable. You may surround it by `maybe_close()`.

static open(*file_path*)
Create an *Extractor* instance for given archive file.

Parameters **file_path** (*str*) – The path of the archive file.

Returns The specified extractor instance.

Return type *Extractor*

Raises `IOError` – If the `file_path` is not a supported archive.

deprecated

class tfsnippet.utils.**deprecated**(*message=''*, *version=None*)
Bases: `object`

Decorate a class, a method or a function to be deprecated.

Usage:

```
@deprecated()
def some_function():
    ...

@deprecated()
class SomeClass:
    ...
```

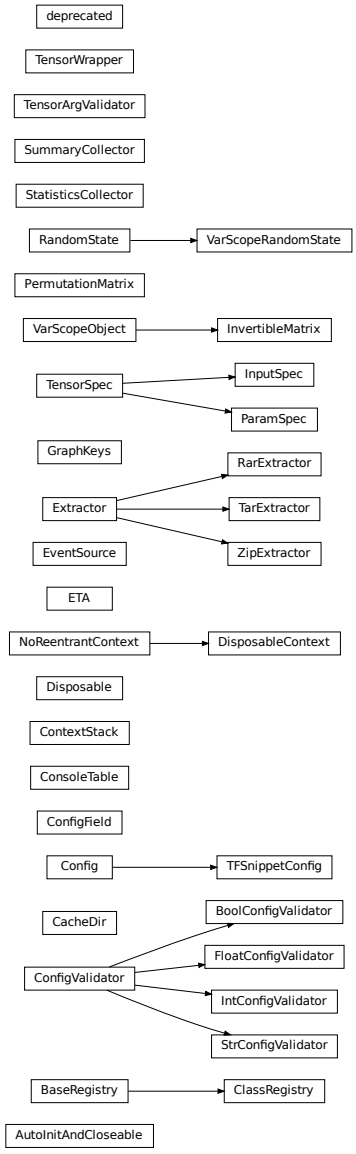
Methods Summary

__call__(...) <==> x(...)

Methods Documentation

__call__(...) <==> *x*(...)

Class Inheritance Diagram



CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z