diktya Documentation

Release 0.0.1

Leon Sixt

Oct 10, 2018

Contents

CHAPTER 1

diktya.callbacks

class OnEpochEnd(*func*, *every_nth_epoch=10*)
Bases: `keras.callbacks.Callback`

on_epoch_end(*epoch*, *logs={}*)

class SampleGAN(*sample_func*, *discriminator_func*, *z*, *real_data*, *callbacks*, *should_sample_func=None*)
Bases: `keras.callbacks.Callback`

Keras callback that provides samples `on_epoch_end` to other callbacks.

Parameters

- **sample_func** – is called with `z` and should return fake samples.
- **discriminator_func** – Should return the discriminator score.
- **z** – Batch of random vectors
- **real_data** – Batch of real data
- **callbacks** – List of callbacks, called with the generated samples.
- **should_sample_func** (*optional*) – Gets the current epoch and returns a bool if we should sample at the given epoch.

sample()

on_train_begin(*logs=None*)

on_epoch_end(*epoch*, *logs=None*)

class VisualiseGAN(*nb_samples*, *output_dir=None*, *show=False*, *preprocess=None*)
Bases: `keras.callbacks.Callback`

Visualise `nb_samples` fake images from the generator.

> **Warning:** Cannot be used as normal keras callback. Can only be used as callback for the SampleGAN callback.

Parameters

- **nb_samples** – number of samples
- **output_dir** (*optional*) – Save image to this directory. Format is {epoch:05d}.
- **(default** (*show*) – False): Show images as matplotlib plot
- **preprocess** (*optional*) – Apply this preprocessing function to the generated images.

on_train_begin(*logs={}*)

call(*samples*)

on_epoch_end(*epoch*, *logs={}*)

class SaveModels(*models*, *output_dir=None*, *every_epoch=50*, *overwrite=True*, *hdf5_attrs=None*)
Bases: keras.callbacks.Callback

on_epoch_end(*epoch*, *log={}*)

class DotProgressBar
Bases: *diktya.callbacks.OnEpochEnd*

class LearningRateScheduler(*optimizer*, *schedule*)
Bases: keras.callbacks.Callback

Learning rate scheduler

Parameters

- **optimizer** (*keras Optimizer*) – schedule the learning rate of this optimizer
- **schedule** (*dict*) – Dictionary of epoch -> lr_value

on_epoch_end(*epoch*, *logs={}*)

class AutomaticLearningRateScheduler(*optimizer*, *metric='loss'*, *min_improvement=0.001*, *epoch_patience=3*, *factor=0.25*)
Bases: keras.callbacks.Callback

This callback automatically reduces the learning rate of the *optimizer*. If the metric did not improve by at least the min_improvement amount in the last epoch_patience epochs, the learning rate of optimizer will be decreased by factor.

Parameters

- **optimizer** (*keras Optimizer*) – Decrease learning rate of this optimizer
- **metric** (*str*) – Name of the metric
- **min_improvement** (*float*) – minimum-improvement
- **epoch_patience** (*int*) – Number of epochs to wait until the metric decreases
- **factor** (*float*) – Reduce learning rate by this factor

on_train_begin(*logs={}*)

on_epoch_begin(*epoch*, *logs={}*)

on_batch_end(*batch*, *logs={}*)

on_epoch_end(*epoch*, *logs={}*)

class HistoryPerBatch(*output_dir=None*, *extra_metrics=None*)
Bases: keras.callbacks.Callback

Saves the metrics of every batch.

Parameters

- **output_dir** (*optional str*) – Save history and plot to this directory.
- **extra_metrics** (*optional list*) – Also montior this metrics.

batch_history
history of every batch. Use `batch_history[metric_name][epoch_idx][batch_idx]` to index.

epoch_history
history of every epoch. Use `epoch_history[metric_name][epoch_idx]` to index.

static from_config (*batch_history*, *epoch_history*)

history

metrics
List of metrics to montior.

on_epoch_begin (*epoch*, *logs=None*)

on_batch_end (*batch*, *logs={}*)

on_epoch_end (*epoch*, *logs={}*)

plot_callback (*fname=None*, *every_nth_epoch=1*, ***kwargs*)
Returns a keras callback that plots this figure `on_epoch_end`.

Parameters

- **fname** (*optional str*) – filename where to save the plot. Default is `{self.output}/history.png`
- **every_nth_epoch** – Plot frequency
- ****kwargs** – Passed to `self.plot(**kwargs)`

save (*fname=None*)

on_train_end (*logs={}*)

plot (*metrics=None*, *fig=None*, *ax=None*, *skip_first_epoch=False*, *use_every_nth_batch=1*, *save_as=None*, *batch_window_size=128*, *percentile=(1, 99)*, *end=None*, *kwargs=None*)
Plots the losses and variance for every epoch.

Parameters

- **metrics** (*list*) – this metric names will be plotted
- **skip_first_epoch** (*bool*) – skip the first epoch. Use full if the first batch has a high loss and brakes the scaling of the loss axis.
- **fig** – matplotlib figure
- **ax** – matplotlib axes
- **save_as** (*str*) – Save figure under this path. If `save_as` is a relative path and `self.output_dir` is set, it is appended to `self.output_dir`.

Returns A tuple of fig, axes

class SaveModelAndWeightsCheckpoint (*filepath*, *monitor='val_loss'*, *verbose=0*, *save_best_only=False*, *mode='auto'*, *hdf5_attrs=None*)
Bases: `keras.callbacks.Callback`

Similiar to keras ModelCheckpoint, but uses `save_model()` to save the model and weights in one file.

filepath can contain named formatting options, which will be filled the value of *epoch* and keys in *logs* (passed in *on_epoch_end*).

For example: if *filepath* is *weights.{epoch:02d}-{val_loss:.2f}.hdf5*, then multiple files will be save with the epoch number and the validation loss.

 # Arguments filepath: string, path to save the model file. monitor: quantity to monitor. verbose: verbosity mode, 0 or 1. save_best_only: if *save_best_only=True*,

> the latest best model according to the validation loss will not be overwritten.

mode: one of {auto, min, max}. If *save_best_only=True*, the decision to overwrite the current save file is made based on either the maximization or the minization of the monitored. For *val_acc*, this should be *max*, for *val_loss* this should be *min*, etc. In *auto* mode, the direction is automatically inferred from the name of the monitored quantity.

hdf5_attrs: Dict of attributes for the hdf5 file.

`save_model`(*fname*, *overwrite=False*, *attrs={}*)

`on_epoch_end`(*epoch*, *logs={}*)

CHAPTER 2

diktya.gan

class GAN(*generator: keras.engine.training.Model*, *discriminator: keras.engine.training.Model*)

Bases: `diktya.models.AbstractModel`

Generative Adversarial Networks (GAN) are a unsupervised learning framework. It consists of a generator and a discriminator network. The generator recieves a noise vector as input and produces some fake data. The discriminator is trained to distinguish between fake data from the generator and real data. The generator is optimized to fool the discriminator. Please refere to Goodwellow et. al for a detail introduction into GANs.

Parameters

- **generator** (*Model*) – model of the generator. Must have one output and one input must be named *z*.
- **discriminator** (*Model*) – model of the discriminator. Must have exaclty one input named *data*. For every sample, the output must be a scalar between 0 and 1.

```
z = Input(shape=(20,), name='z')
data = Input(shape=(1, 32, 32), name='real')

n = 64
fake = sequential([
    Dense(2*16*n, activation='relu'),
    Reshape(2*n, 4, 4),
])(z)

realness = sequential([
    Convolution2D(n, 3, 3, border='same'),
    LeakyRelu(0.3),
    Flatten(),
    Dense(1),
])

generator = Model(z, fake)
generator.compile(Adam(lr=0.0002, beta_1=0.5), 'binary_crossentropy')
```

(continues on next page)

(continued from previous page)

```
discriminator = Model(data, realness)
discriminator.compile(Adam(lr=0.0002, beta_1=0.5), 'binary_crossentropy')
gan = GAN(generator, discriminator)

gan.fit_generator(...)
```

input_names

uses_learning_phase

train_on_batch(*inputs*)

Runs a single weight update on a single batch of data. Updates both generator and discriminator.

Parameters

- **inputs** (*optional*) – Inputs for both the discriminator and the geneator. It can either be a numpy array, a list or dict.
  - **numpy array**: `real`
  - **list**: `[real]`, `[real, z]`
  - **dict: {'real': real}, {'real': real, 'z': z}**, `{'real': real, 'z': z, 'additional_input', x}`
- **generator_inputs** (*optional dict*) – This inputs will only be passed to the generator.
- **discriminator_inputs** (*optional dict*) – This inputs will only be passed to the discriminator.

Returns A list of metrics. You can get the names of the metrics with `metrics_names()`.

fit_generator(*generator*, *nb_batches_per_epoch*, *nb_epoch*, *batch_size=128*, *verbose=1*, *train_on_batch='train_on_batch'*, *callbacks=[]*)

Fits the generator and discriminator on data generated by a Python generator. The generator is not run in parallel as in keras.

Parameters

- **generator** – the output of the generator must satisfy the train_on_batch method.
- **nb_batches_per_epoch** (*int*) – run that many batches per epoch
- **nb_epoch** (*int*) – run that many epochs
- **batch_size** (*int*) – size of one batch
- **verbose** – verbosity mode
- **callbacks** – list of callbacks.

generate(*inputs=None*, *nb_samples=None*)

Use the generator to generate data.

Parameters

- **inputs** – Dictionary of name to input arrays to the generator. Can include the random noise *z* or some conditional varialbes.
- **nb_samples** – Specifies how many samples will be generated, if *z* is not in the *inputs* dictionary.

Returns A numpy array with the generated data.

random_z(*batch_size=32*)
Samples *z* from uniform distribution between -1 and 1. The returned array is of shape `(batch_size, ) + self.z_shape[1:]`

random_z_point()
Returns one random point in the z space.

interpolate(*x, y, nb_steps=100*)
Interpolates linear between two points in the z-space.

Parameters

- **x** – point in the z-space
- **y** – point in the z-space
- **nb_steps** – interpolate that many points

Returns The generated data from the interpolated points. The data corresponding to `x` and `y` are on the first and last position of the returned array.

neighborhood(*z_point=None, std=0.25, n=128*)
samples the neighborhood of a z_point by adding gaussian noise to it. You can control the standard derivation of the noise with std.

CHAPTER 3

diktya.func_api_helpers

trainable(*model*, *trainable*)
Sets all layers in model to trainable and restores the state afterwards.

> **Warning:** Be aware, that the keras `Model.compile` method is lazy. You might want to call `Model._make_train_function` to force a compilation.

Parameters

- **model** – keras model
- **trainable** (*bool*) – set layer.traiable to this value

Example:

```
model = Model(x, y)
with trainable(model, False):
    # layers of model are now not trainable
    # Do something
    z = model(y)
    [...]

# now the layers of `model` are trainable again
```

get_layer(*keras_tensor*)
Returns the corresponding layer to a keras tensor.

sequential(*layers*, *ns=None*, *trainable=True*)
The functional flexible counter part to the keras Sequential model.

Parameters

- **layers** (*list*) – Can be a arbitrary nested list of layers. The layers will be called sequentially. Can contain `None`'s
- **ns** (*optional str*) – Namespace prefix of the layers

- **trainable** (*optional bool*) – set the layer's trainable attribute to this value.

Returns A function that takes a tensor as input, applies all the layers, and returns the output tensor.

Simple example:

Call a list of layers.

```
x = Input(shape=(32,))
y = sequential([
    Dense(10),
    LeakyReLU(0.4),
    Dense(10, activation='sigmoid'),
])(x)

m = Model(x, y)
```

Advanced example:

Use a function to construct reoccuring blocks. The `conv` functions returns a nested list of layers. This allows one to nicely combine and stack different building blocks function.

```
def conv(n, depth=2, f=3, activation='relu'):
    layers = [
        [
            Convolution2D(n, f, f, border_mode='same'),
            BatchNormalization(),
            Activation(activation)
        ] for _ in range(depth)
    ]
    return layers + [MaxPooling2D()]

x = Input(shape=(32,))
y = sequential([
    conv(32),
    conv(64),
    conv(128),
    Flatten(),
    Dense(10, activation='sigmoid'),
])(x, ns='classifier')

m = Model(x, y)
```

concat (*tensors*, *axis=1*, ***kwargs*)
Wrapper around keras merge function.

Parameters

- **tensors** – list of keras tensors
- **axis** – concat on this axis
- **kwargs** – passed to the merge function

Returns The concatenated tensor

rename_layer (*keras_tensor*, *name*)
Renames the layer of the `keras_tensor`

name_tensor (*keras_tensor*, *name*)
Add a layer with this `name` that does nothing.

Usefull to mark a tensor.

keras_copy(*obj*)
Copies a keras object by using the `get_config` method.

predict_wrapper(*func*, *names*)

save_model(*model*, *fname*, *overwrite=False*, *attrs={}*)
Saves the weights and the config of `model` in the HDF5 file `fname`. The model config is saved as: `f.attrs["model"] = model.to_json().encode('utf-8')`, where `f` is the HDF5 file.

load_model(*fname*, *custom_objects={}*)
Loads the model and weights from `fname`. Counterpart to `save_model()`.

get_hdf5_attr(*fname*, *attr_name*, *default=None*)
Returns the toplevel attribute `attr_name` of the hdf5 file `fname`. If `default` is not None and the attribute is not present, then `default` is returned.

CHAPTER 4

diktya.blocks

Note: The functions in this module are espacially usefull together with the *sequential* function.

get_activation(*activation*)

conv2d_block(*n*, *filters=3*, *depth=1*, *border='same'*, *activation='relu'*, *batchnorm=True*, *pooling=None*, *up=False*, *subsample=1*)

2D-Convolutional block consisting of possible muliple repetitions of `Convolution2D`, `BatchNormalization`, and `Activation` layers and can be finished by either a `MaxPooling2D`, a `AveragePooling2D` or a `UpSampling2D` layer.

Parameters

- **n** – number of filters of the convolution layer
- **filters** – shape of the filters are `(filters, filters)`
- **depth** – repeat the convolutional, batchnormalization, activation blocks this many times
- **border** – border_mode of the Convolution2D layer
- **activation** – name or activation or a advanced Activation layer.
- **batchnorm** – use batchnorm layer if true. If it is an integer it indicates the batchnorm mode.
- **pooling** – if given, either `max` or *avg* for MaxPooling2D or AveragePooling2D
- **up** – if true, use a UpSampling2D as last layer. Cannot be true if also pooling is given.

Returns A nested list containing the layers.

resnet(*n*, *filters=3*, *activation='relu'*)

A ResNet block. If the number of filter maps is not equal to `n`, a *conv2d_block()* with `n` filter maps is added.

Parameters

- **n** – number of filters

- **filters** – size of the conv filters

Returns A function that takes a keras tensor as input and runs the resnet block

CHAPTER 5

diktya.distributions

to_radians (*x*)

class JsonConvertable

Bases: `object`

get_config ()

classmethod from_config (*config*)

to_json ()

get (*x*, *custom_objects={}*)

load_from_config (*config*, *custom_objects={}*)

load_from_json (*json_str*, *custom_objects={}*)

class Normalization

Bases: *diktya.distributions.JsonConvertable*

normalize (*array*)

denormalize (*array*)

class ConstantNormalization (*value*)

Bases: *diktya.distributions.JsonConvertable*

normalize (*array*)

denormalize (*array*)

get_config ()

class SubtDivide (*subt*, *scale*)

Bases: *diktya.distributions.Normalization*

normalize (*array*)

denormalize (*array*)

get_config ()

class UnitIntervalTo(*start*, *end*)
Bases: *diktya.distributions.Normalization*

normalize(*array*)

denormalize(*array*)

get_config()

class SinCosAngleNorm
Bases: *diktya.distributions.Normalization*

normalize(*array*)

denormalize(*array*)

class CombineNormalization(*normalizations*)
Bases: *diktya.distributions.Normalization*

normalize(*arr*)

denormalize(*norm_arr*)

get_config()

class Distribution
Bases: *diktya.distributions.JsonConvertable*

sample(*shape*)

default_normalization()

class Zeros
Bases: *diktya.distributions.Distribution*

sample(*shape*)

class Constant(*value*)
Bases: *diktya.distributions.Distribution*

sample(*shape*)

default_normalization()

get_config()

class Normal(*mean*, *std*)
Bases: *diktya.distributions.Distribution*

sample(*shape*)

default_normalization()

get_config()

class TruncNormal(*a*, *b*, *mean*, *std*)
Bases: *diktya.distributions.Distribution*

Normal distribution truncated between [a;b].

sample(*shape*)

length

default_normalization()

get_config()

class Uniform(*low*, *high*)
Bases: *diktya.distributions.Distribution*

sample(*shape*)

length

default_normalization()

get_config()

class Bernoulli
Bases: *diktya.distributions.Distribution*

sample(*shape*)

default_normalization()

class DistributionCollection(*distributions*)
Bases: *diktya.distributions.Distribution*, *diktya.distributions.Normalization*

A collection of multiple distributions:

Parameters distributions – A list of tuples where the tuples have the form * (name, distribution) * (name, distribution, nb_elems) * (name, distribution, nb_elems, normalization) *distribution* must be a subclass of *Distribution*. The *nb_elems* specify how many elements are drawn from the distribution, if omitted it will be set to 1. The *normalization* specifies how it is noramlised. It can be omitted and will then be set to *dist.default_normalization()*.

Example:

```
dist = DistributionCollection([
    ("x_rotation", Normal(0, 1)),
    ("y_rotation", Uniform(-np.pi, np.pi), 1, SinCosAngleNorm()),
    ("center", Normal(0, 2), 2),
])
# Sample 10 vectors from the collection
arr = dist.sample(10)
# The array is a structured numpy array. The keys are the one from
# constructure distributions dictionary.
arr["x_rotation"][0]

# Normalizes the arr samples, according to the normalisation
normed = dist.normalize(arr)

# the normalization/denormalization should be invariant
assert np.allclose(dist.denormalize(normed), arr)
```

sample(*batch_size*)

normalize(*arr*)

denormalize(*norm_arr*)

get_config()

classmethod from_hdf5(*fname*)

examplary_tag_distribution(*nb_bits=12*)

CHAPTER 6

diktya.random_search

fmin (*f*, *space_func*, *n=50*, *n_jobs='n_cpus'*, *verbose=0*)

Minimizes `f` by using random search.

Parameters

- **f** (*function*) – function to optimize. Gets output of space_func as input.
- **space_func** (*function*) – Returns random samples form the search space.
- **n** (*int*) – Number of samples to run. Default 50
- **n_jobs** (*int | str*) – Number of parallel jobs. Use `'n_cpus'`` for same amount as cpus avialable. Default ``'n_cpus'`.

Simple Example:

```
def quadratic_function():
    return (x - 2) ** 2

def space_function():
    return np.random.uniform(-2, 4)

results = fmin(quadratic_function, space_function, n=50)
# sorted by score
print("Min score: {}".format(results[0][0]))
```

CHAPTER 7

diktya.layers.core

class Subtensor(*start*, *stop*, *step=1*, *axis=0*, ***kwargs*)
Bases: `keras.engine.topology.Layer`

Selects only a part of the input.

Parameters

- **start** (*int*) – Start index
- **stop** (*int*) – Stop index
- **axis** (*int*) – Index along this axis

get_output_shape_for(*input_shape*)
Computes the output shape of the layer given an input shape (assumes that the layer will be built to match that input shape).

 # Arguments

input_shape: shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

call(*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments x: input tensor, or list/tuple of input tensors. mask: a masking tensor (or list of tensors). Used mainly in RNNs.

 # Returns: A tensor or list/tuple of tensors.

get_config()
Returns a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by Container (one layer of abstraction above).

class SplitAt(*axis=0*, ***kwargs*)
Bases: `keras.engine.topology.Layer`

get_output_shape_for(*input_shapes*)
Computes the output shape of the layer given an input shape (assumes that the layer will be built to match that input shape).

 # Arguments

input_shape: shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

compute_mask(*x*, *masks=None*)
Computes an output masking tensor, given an input tensor (or list thereof) and an input mask (or list thereof).

 # Arguments input: tensor or list of tensors. input_mask: tensor or list of tensors.

 # Returns

None or a tensor (or list of tensors, one per output tensor of the layer).

call(*xs*, *mask=None*)
This is where the layer's logic lives.

 # Arguments x: input tensor, or list/tuple of input tensors. mask: a masking tensor (or list of tensors). Used mainly in RNNs.

 # Returns: A tensor or list/tuple of tensors.

get_config()
Returns a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by Container (one layer of abstraction above).

class Swap(*a*, *b*, ***kwargs*)
Bases: `keras.engine.topology.Layer`

call(*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments x: input tensor, or list/tuple of input tensors. mask: a masking tensor (or list of tensors). Used mainly in RNNs.

 # Returns: A tensor or list/tuple of tensors.

get_config()
Returns a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by Container (one layer of abstraction above).

class Switch(***kwargs*)
Bases: `keras.engine.topology.Layer`

get_output_shape_for(*input_shape*)
Computes the output shape of the layer given an input shape (assumes that the layer will be built to match that input shape).

 # Arguments

input_shape: shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

call (*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments x: input tensor, or list/tuple of input tensors. mask: a masking tensor (or list of tensors). Used mainly in RNNs.

 # Returns: A tensor or list/tuple of tensors.

class ZeroGradient (***kwargs*)
Bases: `keras.engine.topology.Layer`

Consider the gradient allways zero. Wraps the `theano.gradient.zero_grad` function.

call (*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments x: input tensor, or list/tuple of input tensors. mask: a masking tensor (or list of tensors). Used mainly in RNNs.

 # Returns: A tensor or list/tuple of tensors.

class InBounds (*low=-1*, *high=1*, *clip=True*, *weight=15*, ***kwargs*)
Bases: `keras.engine.topology.Layer`

Between `low` and `high` this layer is the identity. If the value is not in bounds a regularization loss is added to the model.

Parameters

- **low** – lower bound
- **high** – upper bound
- **clip** – Clip output if out of bounds
- **weight** – The regularization loss is multiplied by this

build (*input_shape*)
Creates the layer weights. Must be implemented on all layers that have weights.

 # Arguments

input_shape: Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

compute_loss (*input*, *output*, *input_mask=None*, *output_mask=None*)

call (*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments x: input tensor, or list/tuple of input tensors. mask: a masking tensor (or list of tensors). Used mainly in RNNs.

 # Returns: A tensor or list/tuple of tensors.

get_config ()
Returns a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by Container (one layer of abstraction above).

class BatchLoss (*axis=1*, *normalize=True*, *l1=0.0*, *l2=0.0*, ***kwargs*)
Bases: `keras.engine.topology.Layer`

Regularizes the activation to have `std = 1` and `mean = 0`.

Parameters

- **axis** (*int*) – Axis along to compute the std and mean.
- **normalize** (*bool*) – Normalize the output by the std and mean of the batch during training.
- **weight** (*float*) – Weight of the regularization loss

compute_loss(*input*, *output*, *input_mask=None*, *output_mask=None*)

call(*x*, *mask=None*)

This is where the layer's logic lives.

 # Arguments x: input tensor, or list/tuple of input tensors. mask: a masking tensor (or list of tensors). Used mainly in RNNs.

 # Returns: A tensor or list/tuple of tensors.

get_config()

Returns a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by Container (one layer of abstraction above).

CHAPTER 8

diktya.preprocessing.image

CHAPTER 9

diktya.plot.latexify

9.1 Create native looking matplotlib plots

Adapted from: http://bkanuka.com/articles/native-latex-plots/

figsize (*scale=0.5*, *ratio='golden'*, *textwidth_pt=390.0*)
Returns the figure size as (width, height).

Parameters

- **scale** (*float*) – Scale of the
- **ratio** (*float, str*) – Ratio from height to width. Default is `golden` ratio.
- **textwidth_pt** (*float*) – Width of the latex page. Get this with from LaTeX using ‘‘ he extwidth‘‘

latexify (*rc=None*)
Latexify plots

savefig_pgf_pdf (*fig*, *filename*)
Saves the figure as `{filename}.pgf` and `{filename}.pgf`.

diktya (Greek for networks) contains some complementary utilities for theano and keras .

- Implementation of a *Generative Adversarial Network*.
- Some *usefull helpers* for the keras functional API .
- *Callbacks* for learning rate scheduling and history recording.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Z