Funsor Documentation

Release 0.0

Uber AI Labs

Oct 23, 2019

Funsor Core:

CHAPTER 1

Domains

class Domain

Bases: *funsor.domains.Domain*

An object representing the type and shape of a `Funsor` input or output.

num_elements

size

reals(**shape*)

Construct a real domain of given shape.

bint(*size*)

Construct a bounded integer domain of scalar shape.

find_domain(*op*, **domains*)

Finds the *Domain* resulting when applying `op` to `domains`. :param callable op: An operation. :param Domain *domains: One or more input domains.

CHAPTER 2

Operations

2.1 Built-in operations

abs = ops.abs

add = ops.add

and_ = ops.and_

eq = ops.eq

exp = ops.exp

ge = ops.ge

getitem = ops.GetitemOp(0)
Op encoding an index into one dimension, e.g. `x[:,:,y]` for offset of 2.

gt = ops.gt

invert = ops.invert

le = ops.le

log = ops.log

log1p = ops.log1p

lt = ops.lt

matmul = ops.matmul

max = ops.max

min = ops.min

mul = ops.mul

ne = ops.ne

neg = ops.neg

or_ = ops.or_

pow = ops.pow

safediv = ops.safediv

safesub = ops.safesub

sigmoid = ops.sigmoid

sqrt = ops.sqrt

sub = ops.sub

truediv = ops.truediv

xor = ops.xor

2.2 Operation classes

class Op(*fn*)
Bases: multipledispatch.dispatcher.Dispatcher

class AssociativeOp(*fn*)
Bases: *funsor.ops.Op*

class AddOp(*fn*)
Bases: *funsor.ops.AssociativeOp*

class LogAddExpOp(*fn*)
Bases: *funsor.ops.AssociativeOp*

class SubOp(*fn*)
Bases: *funsor.ops.Op*

class NegOp(*fn*)
Bases: *funsor.ops.Op*

class ReshapeOp(*shape*)
Bases: *funsor.ops.Op*

class GetitemOp(*offset*)
Bases: *funsor.ops.Op*

Op encoding an index into one dimension, e.g. `x[:,:,y]` for offset of 2.

class ExpOp(*fn*)
Bases: funsor.ops.TransformOp

class LogOp(*fn*)
Bases: funsor.ops.TransformOp

class ReciprocalOp(*fn*)
Bases: *funsor.ops.Op*

CHAPTER 3

Interpretations

3.1 Interpreter

set_interpretation(*new*)

interpretation(*new*)

reinterpret(*x*)
Overloaded reinterpretation of a deferred expression.

This handles a limited class of expressions, raising `ValueError` in unhandled cases.

Parameters **x** (*A funsor or data structure holding funsors.*) – An input, typically involving deferred *Funsor* s.

Returns A reinterpreted version of the input.

Raises ValueError

dispatched_interpretation(*fn*)
Decorator to create a dispatched interpretation function.

exception PatternMissingError
Bases: `NotImplementedError`

3.2 Monte Carlo

monte_carlo(*cls*, **args*)
A Monte Carlo interpretation of *Integrate* expressions. This falls back to *eager* in other cases.

monte_carlo_interpretation(***sample_inputs*)
Context manager to set `monte_carlo.sample_inputs` and install the *monte_carlo()* interpretation.

3.3 Memoize

memoize(*cache=None*)

Exploit cons-hashing to do implicit common subexpression elimination

CHAPTER 4

Funsors

4.1 Basic Funsors

reflect(*cls*, **args*, ***kwargs*)
Construct a funsor, populate `._ast_values`, and cons hash. This is the only interpretation allowed to construct funsors.

lazy(*cls*, **args*)
Substitute eagerly but perform ops lazily.

eager(*cls*, **args*)
Eagerly execute ops with known implementations.

eager_or_die(*cls*, **args*)
Eagerly execute ops with known implementations. Disallows lazy *Subs*, *Unary*, *Binary*, and *Reduce*.

Raises `NotImplementedError` no pattern is found.

sequential(*cls*, **args*)
Eagerly execute ops with known implementations; additonally execute vectorized ops sequentially if no known vectorized implementation exists.

moment_matching(*cls*, **args*)
A moment matching interpretation of *Reduce* expressions. This falls back to *eager* in other cases.

class Funsor(*inputs*, *output*, *fresh=None*, *bound=None*)
Bases: `object`

Abstract base class for immutable functional tensors.

Concrete derived classes must implement `__init__()` methods taking hashable `*args` and no optional `**kwargs` so as to support cons hashing.

Derived classes with `.fresh` variables must implement an *eager_subs()* method. Derived classes with `.bound` variables must implement an `_alpha_convert()` method.

Parameters

- **inputs** (*OrderedDict*) – A mapping from input name to domain. This can be viewed as a typed context or a mapping from free variables to domains.
- **output** (Domain) – An output domain.

dtype

shape

quote()

pretty(*maxlen=40*)

item()

requires_grad

reduce(*op*, *reduced_vars=None*)

Reduce along all or a subset of inputs.

Parameters

- **op** (*callable*) – A reduction operation.
- **reduced_vars** (*str, Variable, or set or frozenset thereof.*) – An optional input name or set of names to reduce. If unspecified, all inputs will be reduced.

sample(*sampled_vars*, *sample_inputs=None*)

Create a Monte Carlo approximation to this funsor by replacing functions of `sampled_vars` with *Delta* s.

The result is a *Funsor* with the same `.inputs` and `.output` as the original funsor (plus `sample_inputs` if provided), so that self can be replaced by the sample in expectation computations:

```
y = x.sample(sampled_vars)
assert y.inputs == x.inputs
assert y.output == x.output
exact = (x.exp() * integrand).reduce(ops.add)
approx = (y.exp() * integrand).reduce(ops.add)
```

If `sample_inputs` is provided, this creates a batch of samples scaled samples.

Parameters

- **sampled_vars** (*str, Variable, or set or frozenset thereof.*) – A set of input variables to sample.
- **sample_inputs** (*OrderedDict*) – An optional mapping from variable name to *Domain* over which samples will be batched.

unscaled_sample(*sampled_vars*, *sample_inputs*)

Internal method to draw an unscaled sample. This should be overridden by subclasses.

align(*names*)

Align this funsor to match given `names`. This is mainly useful in preparation for extracting `.data` of a *funsor.torch.Tensor*.

Parameters **names** (*tuple*) – A tuple of strings representing all names but in a new order.

Returns A permuted funsor equivalent to self.

Return type *Funsor*

eager_subs(*subs*)
Internal substitution function. This relies on the user-facing `__call__()` method to coerce non-Funsors to Funsors. Once all inputs are Funsors, *eager_subs()* implementations can recurse to call *Subs*.

eager_unary(*op*)

eager_reduce(*op*, *reduced_vars*)

sequential_reduce(*op*, *reduced_vars*)

moment_matching_reduce(*op*, *reduced_vars*)

abs()

sqrt()

exp()

log()

log1p()

sigmoid()

reshape(*shape*)

sum()

prod()

logsumexp()

all()

any()

min()

max()

to_data(*x*)
Extract a python object from a *Funsor*.

Raises a `ValueError` if free variables remain or if the funsor is lazy.

Parameters **x** – An object, possibly a *Funsor*.

Returns A non-funsor equivalent to `x`.

Raises ValueError if any free variables remain.

Raises PatternMissingError if funsor is not fully evaluated.

class Variable(*name*, *output*)
Bases: *funsor.terms.Funsor*

Funsor representing a single free variable.

Parameters

- **name** (*str*) – A variable name.
- **output** (`funsor.domains.Domain`) – A domain.

eager_subs(*subs*)

class Subs(*arg*, *subs*)
Bases: *funsor.terms.Funsor*

Lazy substitution of the form `x(u=y, v=z)`.

Parameters

- **arg** (Funsor) – A funsor being substituted into.
- **subs** (*tuple*) – A tuple of `(name, value)` pairs, where `name` is a string and `value` can be coerced to a *Funsor* via *to_funsor()*.

unscaled_sample(*sampled_vars*, *sample_inputs*)

class Unary(*op*, *arg*)
Bases: *funsor.terms.Funsor*

Lazy unary operation.

Parameters

- **op** (Op) – A unary operator.
- **arg** (Funsor) – An argument.

class Binary(*op*, *lhs*, *rhs*)
Bases: *funsor.terms.Funsor*

Lazy binary operation.

Parameters

- **op** (Op) – A binary operator.
- **lhs** (Funsor) – A left hand side argument.
- **rhs** (Funsor) – A right hand side argument.

class Reduce(*op*, *arg*, *reduced_vars*)
Bases: *funsor.terms.Funsor*

Lazy reduction over multiple variables.

Parameters

- **op** (Op) – A binary operator.
- **arg** (*funsor*) – An argument to be reduced.
- **reduced_vars** (*frozenset*) – A set of variable names over which to reduce.

class Number(*data*, *dtype=None*)
Bases: *funsor.terms.Funsor*

Funsor backed by a Python number.

Parameters

- **data** (*numbers.Number*) – A python number.
- **dtype** – A nonnegative integer or the string “real”.

item()

eager_unary(*op*)

class Slice(*name*, *start*, *stop*, *step*, *dtype*)
Bases: *funsor.terms.Funsor*

Symbolic representation of a Python `slice` object.

Parameters

- **name** (*str*) – A name for the new slice dimension.

- **start** (*int*) –
- **stop** (*int*) –
- **step** (*int*) – Three args following `slice` semantics.
- **dtype** (*int*) – An optional bounded integer type of this slice.

eager_subs(*subs*)

class Stack(*name*, *parts*)
Bases: *funsor.terms.Funsor*

Stack of funsors along a new input dimension.

Parameters

- **name** (*str*) – The name of the new input variable along which to stack.
- **parts** (*tuple*) – A tuple of Funsors of homogenous output domain.

eager_subs(*subs*)

eager_reduce(*op*, *reduced_vars*)

class Cat(*name*, *parts*, *part_name=None*)
Bases: *funsor.terms.Funsor*

Concatenate funsors along an existing input dimension.

Parameters

- **name** (*str*) – The name of the input variable along which to concatenate.
- **parts** (*tuple*) – A tuple of Funsors of homogenous output domain.

eager_subs(*subs*)

class Lambda(*var*, *expr*)
Bases: *funsor.terms.Funsor*

Lazy inverse to `ops.getitem`.

This is useful to simulate higher-order functions of integers by representing those functions as arrays.

Parameters

- **var** (`Variable`) – A variable to bind.
- **expr** (*funsor*) – A funsor.

class Independent(*fn*, *reals_var*, *bint_var*, *diag_var*)
Bases: *funsor.terms.Funsor*

Creates an independent diagonal distribution.

This is equivalent to substitution followed by reduction:

```
f = ...  # a batched distribution
assert f.inputs['x_i'] == reals(4, 5)
assert f.inputs['i'] == bint(3)

g = Independent(f, 'x', 'i', 'x_i')
assert g.inputs['x'] == reals(3, 4, 5)
assert 'x_i' not in g.inputs
assert 'i' not in g.inputs
```

(continues on next page)

(continued from previous page)

```
x = Variable('x', reals(3, 4, 5))
g == f(x_i=x['i']).reduce(ops.logaddexp, 'i')
```

Parameters

- **fn** (Funsor) – A funsor.
- **reals_var** (*str*) – The name of a real-tensor input.
- **bint_var** (*str*) – The name of a new batch input of `fn`.
- **diag_var** – The name of a smaller-shape real input of `fn`.

unscaled_sample(*sampled_vars*, *sample_inputs*)

eager_subs(*subs*)

of_shape(**shape*)
Decorator to construct a *Funsor* with one free *Variable* per function arg.

to_funsor(**args*, ***kwargs*)
Multiply dispatched method: to_funsor

Convert to a *Funsor*. Only *Funsor* s and scalars are accepted.

param x An object.

param funsor.domains.Domain output An optional output hint.

return A Funsor equivalent to `x`.

rtype Funsor

raises ValueError

Other signatures: Funsor object, object object, Domain Funsor, Domain str, Domain Number Number, Domain Tensor Tensor, Domain ndarray ndarray, Domain

4.2 Delta

solve(*expr*, *value*)
Tries to solve for free inputs of an `expr` such that `expr == value`, and computes the log-abs-det-Jacobian of the resulting substitution.

Parameters

- **expr** (Funsor) – An expression with a free variable.
- **value** (Funsor) – A target value.

Returns A tuple `(name, point, log_abs_det_jacobian)`

Return type tuple

Raises ValueError

class Delta(*terms*)
Bases: *funsor.terms.Funsor*

Normalized delta distribution binding multiple variables.

align(*names*)

eager_subs(*subs*)

eager_reduce(*op*, *reduced_vars*)

unscaled_sample(*sampled_vars*, *sample_inputs*)

4.3 PyTorch

ignore_jit_warnings()

align_tensor(*new_inputs*, *x*, *expand=False*)
Permute and add dims to a tensor to match desired `new_inputs`.

Parameters

- **new_inputs** (*OrderedDict*) – A target set of inputs.
- **x** (`funsor.terms.Funsor`) – A *Tensor* or *Number* .
- **expand** (*bool*) – If False (default), set result size to 1 for any input of `x` not in `new_inputs`; if True expand to `new_inputs` size.

Returns a number or `torch.Tensor` that can be broadcast to other tensors with inputs `new_inputs`.

Return type int or float or *torch.Tensor*

align_tensors(**args*, ***kwargs*)
Permute multiple tensors before applying a broadcasted op.

This is mainly useful for implementing eager funsor operations.

Parameters

- ***args** (`funsor.terms.Funsor`) – Multiple *Tensor* s and *Number* s.
- **expand** (*bool*) – Whether to expand input tensors. Defaults to False.

Returns a pair `(inputs, tensors)` where tensors are all `torch.Tensor` s that can be broadcast together to a single data with given `inputs`.

Return type tuple

class Tensor(*data*, *inputs=None*, *dtype='real'*)
Bases: *funsor.terms.Funsor*

Funsor backed by a PyTorch Tensor.

This follows the `torch.distributions` convention of arranging named "batch" dimensions on the left and remaining "event" dimensions on the right. The output shape is determined by all remaining dims. For example:

```
data = torch.zeros(5,4,3,2)
x = Tensor(data, OrderedDict([("i", bint(5)), ("j", bint(4))]))
assert x.output == reals(3, 2)
```

Operators like `matmul` and `.sum()` operate only on the output shape, and will not change the named inputs.

Parameters

- **data** (`torch.Tensor`) – A PyTorch tensor.
- **inputs** (*OrderedDict*) – An optional mapping from input name (str) to datatype (*Domain*). Defaults to empty.

- **dtype** (*int or the string "real".*) – optional output datatype. Defaults to "real".

item()

clamp_finite()

requires_grad

align(*names*)

eager_subs(*subs*)

eager_unary(*op*)

eager_reduce(*op*, *reduced_vars*)

unscaled_sample(*sampled_vars*, *sample_inputs*)

arange(*name*, **args*, ***kwargs*)

Helper to create a named `torch.arange()` funsor. In some cases this can be replaced by a symbolic *Slice* .

Parameters

- **name** (*str*) – A variable name.
- **start** (*int*) –
- **stop** (*int*) –
- **step** (*int*) – Three args following `slice` semantics.
- **dtype** (*int*) – An optional bounded integer type of this slice.

Return type *Tensor*

materialize(*x*)

Attempt to convert a Funsor to a *Number* or *Tensor* by substituting *arange()* s into its free variables.

Parameters **x** (`Funsor`) – A funsor.

Return type *Funsor*

class Function(*fn*, *output*, *args*)

Bases: *funsor.terms.Funsor*

Funsor wrapped by a PyTorch function.

Functions are assumed to support broadcasting and can be eagerly evaluated on funsors with free variables of int type (i.e. batch dimensions).

Function s are usually created via the *function()* decorator.

Parameters

- **fn** (*callable*) – A PyTorch function to wrap.
- **output** (`funsor.domains.Domain`) – An output domain.
- **args** (`Funsor`) – Funsor arguments.

function(**signature*)

Decorator to wrap a PyTorch function.

Example:

```
@funsor.torch.function(reals(3,4), reals(4,5), reals(3,5))
def matmul(x, y):
    return torch.matmul(x, y)

@funsor.torch.function(reals(10), reals(10, 10), reals())
def mvn_log_prob(loc, scale_tril, x):
    d = torch.distributions.MultivariateNormal(loc, scale_tril)
    return d.log_prob(x)
```

To support functions that output nested tuples of tensors, specify a nested tuple of output types, for example:

```
@funsor.torch.function(reals(8), (reals(), bint(8)))
def max_and_argmax(x):
    return torch.max(x, dim=-1)
```

Parameters ***signature** – A sequence if input domains followed by a final output domain or nested tuple of output domains.

class Einsum(*equation*, *operands*)

Bases: *funsor.terms.Funsor*

Wrapper around `torch.einsum()` to operate on real-valued Funsors.

Note this operates only on the `output` tensor. To perform sum-product contractions on named dimensions, instead use + and *Reduce*.

Parameters

- **equation** (*str*) – An `torch.einsum()` equation.
- **operands** (*tuple*) – A tuple of input funsors.

torch_tensordot(*x*, *y*, *dims*)

Wrapper around `torch.tensordot()` to operate on real-valued Funsors.

Note this operates only on the `output` tensor. To perform sum-product contractions on named dimensions, instead use + and *Reduce*.

Arguments should satisfy:

```
len(x.shape) >= dims
len(y.shape) >= dims
dims == 0 or x.shape[-dims:] == y.shape[:dims]
```

Parameters

- **x** (`Funsor`) – A left hand argument.
- **y** (`Funsor`) – A y hand argument.
- **dims** (*int*) – The number of dimension of overlap of output shape.

Return type *Funsor*

4.4 NumPy

align_array(*new_inputs*, *x*)

Permute and expand an array to match desired `new_inputs`.

Parameters

- **new_inputs** (*OrderedDict*) – A target set of inputs.
- **x** (funsor.terms.Funsor) – A *Array* s or or *Number* .

Returns a number or numpy.ndarray that can be broadcast to other array with inputs new_inputs.

Return type tuple

align_arrays(**args*)

Permute multiple arrays before applying a broadcasted op.

This is mainly useful for implementing eager funsor operations.

Parameters ***args** (funsor.terms.Funsor) – Multiple *Array* s and *Number* s.

Returns a pair (inputs, arrays) where arrayss are all numpy.ndarray s that can be broadcast together to a single data with given inputs.

Return type tuple

class ArrayMeta(*name*, *bases*, *dct*)

Bases: funsor.terms.FunsorMeta

Wrapper to fill in default args and convert between OrderedDict and tuple.

class Array(*data*, *inputs=None*, *dtype='real'*)

Bases: *funsor.terms.Funsor*

Funsor backed by a NumPy Array.

This follows the torch.distributions convention of arranging named "batch" dimensions on the left and remaining "event" dimensions on the right. The output shape is determined by all remaining dims. For example:

```
data = np.zeros((5,4,3,2))
x = Array(data, OrderedDict([("i", bint(5)), ("j", bint(4))]))
assert x.output == reals(3, 2)
```

Operators like matmul and .sum() operate only on the output shape, and will not change the named inputs.

Parameters

- **data** (*np.ndarray*) – A NumPy array.
- **inputs** (*OrderedDict*) – An optional mapping from input name (str) to datatype (*Domain*). Defaults to empty.
- **dtype** (*int or the string "real".*) – optional output datatype. Defaults to "real".

item()

align(*names*)

eager_subs(*subs*)

eager_binary_array_number(*op*, *lhs*, *rhs*)

eager_binary_number_array(*op*, *lhs*, *rhs*)

eager_binary_array_array(*op*, *lhs*, *rhs*)

arange(*name*, *size*)

Helper to create a named numpy.arange() funsor.

Parameters

- **name** (*str*) – A variable name.
- **size** (*int*) – A size.

Return type *Array*

materialize(*x*)

Attempt to convert a Funsor to a *Number* or `numpy.ndarray` by substituting *arange()* s into its free variables.

4.5 Gaussian

class BlockVector(*shape*)

Bases: `object`

Jit-compatible helper to build blockwise vectors. Syntax is similar to `torch.zeros()`

```
x = BlockVector((100, 20))
x[..., 0:4] = x1
x[..., 6:10] = x2
x = x.as_tensor()
assert x.shape == (100, 20)
```

as_tensor()

class BlockMatrix(*shape*)

Bases: `object`

Jit-compatible helper to build blockwise matrices. Syntax is similar to `torch.zeros()`

```
x = BlockMatrix((100, 20, 20))
x[..., 0:4, 0:4] = x11
x[..., 0:4, 6:10] = x12
x[..., 6:10, 0:4] = x12.transpose(-1, -2)
x[..., 6:10, 6:10] = x22
x = x.as_tensor()
assert x.shape == (100, 20, 20)
```

as_tensor()

align_gaussian(*new_inputs*, *old*)

Align data of a Gaussian distribution to a new `inputs` shape.

class Gaussian(*info_vec*, *precision*, *inputs*)

Bases: *funsor.terms.Funsor*

Funsor representing a batched joint Gaussian distribution as a log-density function.

Mathematically, a Gaussian represents the density function:

```
f(x) = < x | info_vec > - 0.5 * < x | precision | x >
     = < x | info_vec - 0.5 * precision @ x >
```

Note that *Gaussian* s are not normalized, rather they are canonicalized to evaluate to zero log density at the origin: `f(0) = 0`. This canonical form is useful in combination with the information filter representation because it allows *Gaussian* s with incomplete information, i.e. zero eigenvalues in the precision matrix. These incomplete distributions arise when making low-dimensional observations on higher dimensional hidden state.

Parameters

- **info_vec** (torch.Tensor) – An optional batched information vector, where `info_vec = precision @ mean`.
- **precision** (torch.Tensor) – A batched positive semidefinite precision matrix.
- **inputs** (*OrderedDict*) – Mapping from name to *Domain* .

log_normalizer

align(*names*)

eager_subs(*subs*)

eager_reduce(*op*, *reduced_vars*)

unscaled_sample(*sampled_vars*, *sample_inputs*)

4.6 Joint

moment_matching_contract_default(**args*)

moment_matching_contract_joint(*red_op*, *bin_op*, *reduced_vars*, *discrete*, *gaussian*)

eager_reduce_exp(*op*, *arg*, *reduced_vars*)

eager_independent_joint(*joint*, *reals_var*, *bint_var*, *diag_var*)

4.7 Contraction

class Contraction(*red_op*, *bin_op*, *reduced_vars*, *terms*)

Bases: *funsor.terms.Funsor*

Declarative representation of a finitary sum-product operation.

After normalization via the `normalize()` interpretation contractions will canonically order their terms by type:

```
Delta, Number, Tensor, Gaussian
```

unscaled_sample(*sampled_vars*, *sample_inputs*)

align(*names*)

GaussianMixture

alias of *funsor.cnf.Contraction*

recursion_reinterpret_contraction(*x*)

eager_contraction_generic_to_tuple(*red_op*, *bin_op*, *reduced_vars*, **terms*)

eager_contraction_generic_recursive(*red_op*, *bin_op*, *reduced_vars*, *terms*)

eager_contraction_to_reduce(*red_op*, *bin_op*, *reduced_vars*, *term*)

eager_contraction_to_binary(*red_op*, *bin_op*, *reduced_vars*, *lhs*, *rhs*)

eager_contraction_tensor(*red_op*, *bin_op*, *reduced_vars*, **terms*)

eager_contraction_gaussian(*red_op*, *bin_op*, *reduced_vars*, *x*, *y*)

normalize_contraction_commutative_canonical_order(*red_op*, *bin_op*, *reduced_vars*, **terms*)

normalize_contraction_commute_joint(*red_op*, *bin_op*, *reduced_vars*, *other*, *mixture*)

normalize_contraction_generic_args(*red_op*, *bin_op*, *reduced_vars*, **terms*)

normalize_trivial(*red_op*, *bin_op*, *reduced_vars*, *term*)

normalize_contraction_generic_tuple(*red_op*, *bin_op*, *reduced_vars*, *terms*)

binary_to_contract(*op*, *lhs*, *rhs*)

reduce_funsor(*op*, *arg*, *reduced_vars*)

unary_neg_variable(*op*, *arg*)

do_fresh_subs(*arg*, *subs*)

distribute_subs_contraction(*arg*, *subs*)

normalize_fuse_subs(*arg*, *subs*)

binary_subtract(*op*, *lhs*, *rhs*)

binary_divide(*op*, *lhs*, *rhs*)

unary_log_exp(*op*, *arg*)

unary_contract(*op*, *arg*)

4.8 Integrate

class Integrate(*log_measure*, *integrand*, *reduced_vars*)

Bases: *funsor.terms.Funsor*

Funsor representing an integral wrt a log density funsor.

Parameters

- **log_measure** (Funsor) – A log density funsor treated as a measure.
- **integrand** (Funsor) – An integrand funsor.
- **reduced_vars** (*str, Variable, or set or frozenset thereof.*) – An input name or set of names to reduce.

CHAPTER 5

Optimizer

unfold(*cls*, **args*)

unfold_contraction_generic_tuple(*red_op*, *bin_op*, *reduced_vars*, *terms*)

optimize(*cls*, **args*)

eager_contract_base(*red_op*, *bin_op*, *reduced_vars*, **terms*)

optimize_contract_finitary_funsor(*red_op*, *bin_op*, *reduced_vars*, *terms*)

apply_optimizer(*x*)

CHAPTER 6

Adjoint Algorithms

class AdjointTape

Bases: object

adjoint(*red_op*, *bin_op*, *root*, *targets*)

adjoint_tensor(*adj_redop*, *adj_binop*, *out_adj*, *data*, *inputs*, *dtype*)

adjoint_binary(*adj_redop*, *adj_binop*, *out_adj*, *op*, *lhs*, *rhs*)

adjoint_reduce(*adj_redop*, *adj_binop*, *out_adj*, *op*, *arg*, *reduced_vars*)

adjoint_contract_unary(*adj_redop*, *adj_binop*, *out_adj*, *sum_op*, *prod_op*, *reduced_vars*, *arg*)

adjoint_contract_generic(*adj_redop*, *adj_binop*, *out_adj*, *sum_op*, *prod_op*, *reduced_vars*, *terms*)

adjoint_contract(*adj_redop*, *adj_binop*, *out_adj*, *sum_op*, *prod_op*, *reduced_vars*, *lhs*, *rhs*)

adjoint_cat(*adj_redop*, *adj_binop*, *out_adj*, *name*, *parts*, *part_name*)

adjoint_subs_tensor(*adj_redop*, *adj_binop*, *out_adj*, *arg*, *subs*)

adjoint_subs_gaussianmixture_gaussianmixture(*adj_redop*, *adj_binop*, *out_adj*, *arg*, *subs*)

adjoint_subs_gaussian_gaussian(*adj_redop*, *adj_binop*, *out_adj*, *arg*, *subs*)

adjoint_subs_gaussianmixture_discrete(*adj_redop*, *adj_binop*, *out_adj*, *arg*, *subs*)

CHAPTER 7

Sum-Product Algorithms

partial_sum_product (*sum_op*, *prod_op*, *factors*, *eliminate=frozenset()*, *plates=frozenset()*)
Performs partial sum-product contraction of a collection of factors.

Returns a list of partially contracted Funsors.

Return type list

sum_product (*sum_op*, *prod_op*, *factors*, *eliminate=frozenset()*, *plates=frozenset()*)
Performs sum-product contraction of a collection of factors.

Returns a single contracted Funsor.

Return type *Funsor*

naive_sequential_sum_product (*sum_op*, *prod_op*, *trans*, *time*, *step*)

sequential_sum_product (*sum_op*, *prod_op*, *trans*, *time*, *step*)
For a funsor `trans` with dimensions `time`, `prev` and `curr`, computes a recursion equivalent to:

```
tail_time = 1 + arange("time", trans.inputs["time"].size - 1)
tail = sequential_sum_product(sum_op, prod_op,
                              trans(time=tail_time),
                              time, {"prev": "curr"})
return prod_op(trans(time=0)(curr="drop"), tail(prev="drop"))
↪reduce(sum_op, "drop")
```

but does so efficiently in parallel in O(log(time)).

Parameters

- **sum_op** (AssociativeOp) – A semiring sum operation.
- **prod_op** (AssociativeOp) – A semiring product operation.
- **trans** (Funsor) – A transition funsor.
- **time** (Variable) – The time input dimension.

- **step** (*dict*) – A dict mapping previous variables to current variables. This can contain multiple pairs of prev->curr variable names.

class MarkovProductMeta(*name*, *bases*, *dct*)
Bases: funsor.terms.FunsorMeta

Wrapper to convert step to a tuple and fill in default step_names.

class MarkovProduct(*sum_op*, *prod_op*, *trans*, *time*, *step*, *step_names*)
Bases: *funsor.terms.Funsor*

Lazy representation of *sequential_sum_product()* .

Parameters

- **sum_op** (AssociativeOp) – A marginalization op.
- **prod_op** (AssociativeOp) – A Bayesian fusion op.
- **trans** (Funsor) – A sequence of transition factors, usually varying along the time input.
- **time** (*str or* Variable) – A time dimension.
- **step** (*dict*) – A str-to-str mapping of "previous" inputs of trans to "current" inputs of trans.
- **step_names** (*dict*) – Optional, for internal use by alpha conversion.

eager_subs(*subs*)

eager_markov_product(*sum_op*, *prod_op*, *trans*, *time*, *step*, *step_names*)

CHAPTER 8

Affine Pattern Matching

is_affine(*fn*)
A sound but incomplete test to determine whether a funsor is affine with respect to all of its real inputs.

Parameters **fn** (Funsor) – A funsor.

Return type bool

affine_inputs(*fn*)
Returns a [sound sub]set of real inputs of `fn` wrt which `fn` is known to be affine.

Parameters **fn** (Funsor) – A funsor.

Returns A set of input names wrt which `fn` is affine.

Return type frozenset

extract_affine(*fn*)
Extracts an affine representation of a funsor, satisfying:

```
x = ...
const, coeffs = extract_affine(x)
y = sum(Einsum(eqn, (coeff, Variable(var, coeff.output)))
        for var, (coeff, eqn) in coeffs.items())
assert_close(y, x)
assert frozenset(coeffs) == affine_inputs(x)
```

The `coeffs` will have one key per input wrt which `fn` is known to be affine (via *affine_inputs()*), and `const` and `coeffs.values` will all be constant wrt these inputs.

The affine approximation is computed by ev evaluating `fn` at zero and each basis vector. To improve performance, users may want to run under the *memoize()* interpretation.

Parameters **fn** (Funsor) – A funsor that is affine wrt the (add,mul) semiring in some subset of its inputs.

Returns A pair `(const, coeffs)` where const is a funsor with no real inputs and `coeffs` is an OrderedDict mapping input name to a `(coefficient, eqn)` pair in einsum form.

Return type tuple

CHAPTER 9

Testing Utiltites

xfail_if_not_implemented(*msg='Not implemented'*)

class ActualExpected
Bases: `funsor.testing.LazyComparison`

Lazy string formatter for test assertions.

id_from_inputs(*inputs*)

assert_close(*actual*, *expected*, *atol=1e-06*, *rtol=1e-06*)

check_funsor(*x*, *inputs*, *output*, *data=None*)
Check dims and shape modulo reordering.

xfail_param(**args*, ***kwargs*)

make_einsum_example(*equation*, *fill=None*, *sizes=(2, 3)*)

assert_equiv(*x*, *y*)
Check that two funsors are equivalent up to permutation of inputs.

random_tensor(*inputs*, *output=reals()*)
Creates a random *funsor.torch.Tensor* with given inputs and output.

random_array(*inputs*, *output*)
Creates a random *funsor.numpy.Array* with given inputs and output.

random_gaussian(*inputs*)
Creates a random *funsor.gaussian.Gaussian* with given inputs.

random_mvn(*batch_shape*, *dim*, *diag=False*)
Generate a random `torch.distributions.MultivariateNormal` with given shape.

make_plated_hmm_einsum(*num_steps*, *num_obs_plates=1*, *num_hidden_plates=0*)

make_chain_einsum(*num_steps*)

make_hmm_einsum(*num_steps*)

CHAPTER 10

Pyro-Compatible Distributions

This interface provides a number of PyTorch-style distributions that use funsors internally to perform inference. These high-level objects are based on a wrapping class: `FunsorDistribution` which wraps a funsor in a PyTorch-distributions-compatible interface. *`FunsorDistribution`* objects can be used directly in Pyro models (using the standard Pyro backend).

10.1 FunsorDistribution Base Class

class FunsorDistribution(*funsor_dist*, *batch_shape=torch.Size([])*, *event_shape=torch.Size([])*, *dtype='real'*, *validate_args=None*)

Bases: `pyro.distributions.torch_distribution.TorchDistribution`

`Distribution` wrapper around a *`Funsor`* for use in Pyro code. This is typically used as a base class for specific funsor inference algorithms wrapped in a distribution interface.

Parameters

- **funsor_dist** (`funsor.terms.Funsor`) – A funsor with an input named "value" that is treated as a random variable. The distribution should be normalized over "value".
- **batch_shape** (*`torch.Size`*) – The distribution's batch shape. This must be in the same order as the input of the `funsor_dist`, but may contain extra dims of size 1.
- **event_shape** – The distribution's event shape.

arg_constraints = {}

support

log_prob(*value*)

sample(*sample_shape=torch.Size([])*)

rsample(*sample_shape=torch.Size([])*)

expand(*batch_shape*, *_instance=None*)

10.2 Hidden Markov Models

class DiscreteHMM(*initial_logits*, *transition_logits*, *observation_dist*, *validate_args=None*)

Bases: *funsor.pyro.distribution.FunsorDistribution*

Hidden Markov Model with discrete latent state and arbitrary observation distribution. This uses [1] to parallelize over time, achieving O(log(time)) parallel complexity.

The event_shape of this distribution includes time on the left:

```
event_shape = (num_steps,) + observation_dist.event_shape
```

This distribution supports any combination of homogeneous/heterogeneous time dependency of `transition_logits` and `observation_dist`. However, because time is included in this distribution's event_shape, the homogeneous+homogeneous case will have a broadcastable event_shape with `num_steps = 1`, allowing *log_prob()* to work with arbitrary length data:

```
# homogeneous + homogeneous case:
event_shape = (1,) + observation_dist.event_shape
```

This class should be interchangeable with `pyro.distributions.hmm.DiscreteHMM`.

References:

[1] Simo Sarkka, Angel F. Garcia-Fernandez (2019) "Temporal Parallelization of Bayesian Filters and Smoothers" https://arxiv.org/pdf/1905.13002.pdf

Parameters

- **initial_logits** (`Tensor`) – A logits tensor for an initial categorical distribution over latent states. Should have rightmost size `state_dim` and be broadcastable to `batch_shape + (state_dim,)`.
- **transition_logits** (`Tensor`) – A logits tensor for transition conditional distributions between latent states. Should have rightmost shape `(state_dim, state_dim)` (old, new), and be broadcastable to `batch_shape + (num_steps, state_dim, state_dim)`.
- **observation_dist** (*Distribution*) – A conditional distribution of observed data conditioned on latent state. The `.batch_shape` should have rightmost size `state_dim` and be broadcastable to `batch_shape + (num_steps, state_dim)`. The `.event_shape` may be arbitrary.

has_rsample

log_prob(*value*)

expand(*batch_shape*, *_instance=None*)

class GaussianHMM(*initial_dist*, *transition_matrix*, *transition_dist*, *observation_matrix*, *observation_dist*, *validate_args=None*)

Bases: *funsor.pyro.distribution.FunsorDistribution*

Hidden Markov Model with Gaussians for initial, transition, and observation distributions. This adapts [1] to parallelize over time to achieve O(log(time)) parallel complexity, however it differs in that it tracks the log normalizer to ensure `log_prob()` is differentiable.

This corresponds to the generative model:

```
z = initial_distribution.sample()
x = []
for t in range(num_steps):
    z = z @ transition_matrix + transition_dist.sample()
    x.append(z @ observation_matrix + observation_dist.sample())
```

The event_shape of this distribution includes time on the left:

```
event_shape = (num_steps,) + observation_dist.event_shape
```

This distribution supports any combination of homogeneous/heterogeneous time dependency of `transition_dist` and `observation_dist`. However, because time is included in this distribution's event_shape, the homogeneous+homogeneous case will have a broadcastable event_shape with `num_steps = 1`, allowing `log_prob()` to work with arbitrary length data:

```
event_shape = (1, obs_dim)  # homogeneous + homogeneous case
```

This class should be compatible with `pyro.distributions.hmm.GaussianHMM`, but additionally supports funsor *adjoint* algorithms.

References:

[1] Simo Sarkka, Angel F. Garcia-Fernandez (2019) "Temporal Parallelization of Bayesian Filters and Smoothers" https://arxiv.org/pdf/1905.13002.pdf

Variables

- **hidden_dim** (*int*) – The dimension of the hidden state.
- **obs_dim** (*int*) – The dimension of the observed state.

Parameters

- **initial_dist** (*MultivariateNormal*) – A distribution over initial states. This should have batch_shape broadcastable to `self.batch_shape`. This should have event_shape `(hidden_dim,)`.
- **transition_matrix** (*Tensor*) – A linear transformation of hidden state. This should have shape broadcastable to `self.batch_shape + (num_steps, hidden_dim, hidden_dim)` where the rightmost dims are ordered `(old, new)`.
- **transition_dist** (*MultivariateNormal*) – A process noise distribution. This should have batch_shape broadcastable to `self.batch_shape + (num_steps,)`. This should have event_shape `(hidden_dim,)`.
- **transition_matrix** – A linear transformation from hidden to observed state. This should have shape broadcastable to `self.batch_shape + (num_steps, hidden_dim, obs_dim)`.
- **observation_dist** (*MultivariateNormal or Normal*) – An observation noise distribution. This should have batch_shape broadcastable to `self.batch_shape + (num_steps,)`. This should have event_shape `(obs_dim,)`.

has_rsample = True

arg_constraints = {}

class GaussianMRF(*initial_dist*, *transition_dist*, *observation_dist*, *validate_args=None*)

Bases: *funsor.pyro.distribution.FunsorDistribution*

Temporal Markov Random Field with Gaussian factors for initial, transition, and observation distributions. This adapts [1] to parallelize over time to achieve O(log(time)) parallel complexity, however it differs in that it tracks the log normalizer to ensure `log_prob()` is differentiable.

The event_shape of this distribution includes time on the left:

```
event_shape = (num_steps,) + observation_dist.event_shape
```

This distribution supports any combination of homogeneous/heterogeneous time dependency of `transition_dist` and `observation_dist`. However, because time is included in this distribution's event_shape, the homogeneous+homogeneous case will have a broadcastable event_shape with `num_steps = 1`, allowing `log_prob()` to work with arbitrary length data:

```
event_shape = (1, obs_dim)  # homogeneous + homogeneous case
```

This class should be compatible with `pyro.distributions.hmm.GaussianMRF` , but additionally supports funsor *adjoint* algorithms.

References:

[1] Simo Sarkka, Angel F. Garcia-Fernandez (2019) "Temporal Parallelization of Bayesian Filters and Smoothers" https://arxiv.org/pdf/1905.13002.pdf

Variables

- **hidden_dim** (*int*) – The dimension of the hidden state.
- **obs_dim** (*int*) – The dimension of the observed state.

Parameters

- **initial_dist** (*MultivariateNormal*) – A distribution over initial states. This should have batch_shape broadcastable to `self.batch_shape`. This should have event_shape `(hidden_dim,)`.
- **transition_dist** (*MultivariateNormal*) – A joint distribution factor over a pair of successive time steps. This should have batch_shape broadcastable to `self.batch_shape + (num_steps,)`. This should have event_shape `(hidden_dim + hidden_dim,)` (old+new).
- **observation_dist** (*MultivariateNormal*) – A joint distribution factor over a hidden and an observed state. This should have batch_shape broadcastable to `self.batch_shape + (num_steps,)`. This should have event_shape `(hidden_dim + obs_dim,)`.

has_rsample = True

class SwitchingLinearHMM(*initial_logits*, *initial_mvn*, *transition_logits*, *transition_matrix*, *transition_mvn*, *observation_matrix*, *observation_mvn*, *exact=False*, *validate_args=None*)

Bases: *funsor.pyro.distribution.FunsorDistribution*

Switching Linear Dynamical System represented as a Hidden Markov Model.

This corresponds to the generative model:

```
z = Categorical(logits=initial_logits).sample()
y = initial_mvn[z].sample()
x = []
for t in range(num_steps):
```

(continues on next page)

(continued from previous page)

```
    z = Categorical(logits=transition_logits[t, z]).sample()
    y = y @ transition_matrix[t, z] + transition_mvn[t, z].sample()
    x.append(y @ observation_matrix[t, z] + observation_mvn[t, z].sample())
```

Viewed as a dynamic Bayesian network:

```
z[t-1] ----> z[t] ---> z[t+1]         Discrete latent class
   |  \       |  \       |   \
   | y[t-1] ----> y[t] ----> y[t+1]   Gaussian latent state
   |   /      |   /      |   /
   V  /       V  /       V  /
x[t-1]       x[t]      x[t+1]         Gaussian observation
```

Let `class` be the latent class, `state` be the latent multivariate normal state, and `value` be the observed multivariate normal value.

Parameters

- **initial_logits** (Tensor) – Represents `p(class[0])`.
- **initial_mvn** (*MultivariateNormal*) – Represents `p(state[0] | class[0])`.
- **transition_logits** (Tensor) – Represents `p(class[t+1] | class[t])`.
- **transition_matrix** (Tensor) –
- **transition_mvn** (*MultivariateNormal*) – Together with `transition_matrix`, this represents `p(state[t], state[t+1] | class[t])`.
- **observation_matrix** (Tensor) –
- **observation_mvn** (*MultivariateNormal*) – Together with `observation_matrix`, this represents `p(value[t+1], state[t+1] | class[t+1])`.
- **exact** (*bool*) – If True, perform exact inference at cost exponential in `num_steps`. If False, use a *moment_matching()* approximation and use parallel scan algorithm to reduce parallel complexity to logarithmic in `num_steps`. Defaults to False.

has_rsample = True

arg_constraints = {}

log_prob(*value*)

expand(*batch_shape*, *_instance=None*)

filter(*value*)

Compute posterior over final state given a sequence of observations.

Parameters **value** (Tensor) – A sequence of observations.

Returns A posterior distribution over latent states at the final time step, represented as a pair `(cat, mvn)`, where `Categorical` distribution over mixture components and `mvn` is a `MultivariateNormal` with rightmost batch dimension ranging over mixture components. This can then be used to initialize a sequential Pyro model for prediction.

Return type tuple

10.3 Conversion Utilities

This module follows a convention for converting between funsors and PyTorch distribution objects. This convention is compatible with NumPy/PyTorch-style broadcasting. Following PyTorch distributions (and Tensorflow distributions), we consider "event shapes" to be on the right and broadcast-compatible "batch shapes" to be on the left.

This module also aims to be forgiving in inputs and pedantic in outputs: methods accept either the superclass `torch.distributions.Distribution` objects or the subclass `pyro.distributions.TorchDistribution` objects. Methods return only the narrower subclass `pyro.distributions.TorchDistribution` objects.

tensor_to_funsor (*tensor*, *event_inputs=()*, *event_output=0*, *dtype='real'*)

Convert a `torch.Tensor` to a *`funsor.torch.Tensor`*.

Note this should not touch data, but may trigger a `torch.Tensor.reshape()` op.

Parameters

- **tensor** (`torch.Tensor`) – A PyTorch tensor.
- **event_inputs** (*`tuple`*) – A tuple of names for rightmost tensor dimensions. If `tensor` has these names, they will be converted to `result.inputs`.
- **event_output** (*`int`*) – The number of tensor dimensions assigned to `result.output`. These must be on the right of any `event_input` dimensions.

Returns A funsor.

Return type *funsor.torch.Tensor*

funsor_to_tensor (*funsor_*, *ndims*, *event_inputs=()*)

Convert a *`funsor.torch.Tensor`* to a `torch.Tensor`.

Note this should not touch data, but may trigger a `torch.Tensor.reshape()` op.

Parameters

- **funsor** (`funsor.torch.Tensor`) – A funsor.
- **ndims** (*`int`*) – The number of result dims, `== result.dim()`.
- **event_inputs** (*`tuple`*) – Names assigned to rightmost dimensions.

Returns A PyTorch tensor.

Return type *torch.Tensor*

mvn_to_funsor (*pyro_dist*, *event_dims=()*, *real_inputs={}*)

Convert a joint `torch.distributions.MultivariateNormal` distribution into a *`Funsor`* with multiple real inputs.

This should satisfy:

```
sum(d.num_elements for d in real_inputs.values())
  == pyro_dist.event_shape[0]
```

Parameters

- **pyro_dist** (*`torch.distributions.MultivariateNormal`*) – A multivariate normal distribution over one or more variables of real or vector or tensor type.
- **event_dims** (*`tuple`*) – A tuple of names for rightmost dimensions. These will be assigned to `result.inputs` of type `bint`.

- **real_inputs** (*OrderedDict*) – A dict mapping real variable name to appropriately sized `reals()`. The sum of all `.numel()` of all real inputs should be equal to the `pyro_dist` dimension.

Returns A funsor with given `real_inputs` and possibly additional bint inputs.

Return type *funsor.terms.Funsor*

funsor_to_mvn (*gaussian*, *ndims*, *event_inputs=()*)

Convert a *Funsor* to a `pyro.distributions.MultivariateNormal` , dropping the normalization constant.

Parameters

- **gaussian** (funsor.gaussian.Gaussian *or* *funsor.joint.Joint*) – A Gaussian funsor.
- **ndims** (*int*) – The number of batch dimensions in the result.
- **event_inputs** (*tuple*) – A tuple of names to assign to rightmost dimensions.

Returns a multivariate normal distribution.

Return type pyro.distributions.MultivariateNormal

funsor_to_cat_and_mvn (*funsor_*, *ndims*, *event_inputs*)

Converts a labeled gaussian mixture model to a pair of distributions.

Parameters

- **funsor** (*funsor.joint.Joint*) – A Gaussian mixture funsor.
- **ndims** (*int*) – The number of batch dimensions in the result.

Returns A pair `(cat, mvn)`, where `cat` is a `Categorical` distribution over mixture components and `mvn` is a `MultivariateNormal` with rightmost batch dimension ranging over mixture components.

class AffineNormal (*matrix*, *loc*, *scale*, *value_x*, *value_y*)

Bases: *funsor.terms.Funsor*

Represents a conditional diagonal normal distribution over a random variable `Y` whose mean is an affine function of a random variable `X`. The likelihood of `X` is thus:

```
AffineNormal(matrix, loc, scale).condition(y).log_density(x)
```

which is equivalent to:

```
Normal(x @ matrix + loc, scale).to_event(1).log_prob(y)
```

Parameters

- **matrix** (`Funsor`) – A transformation from `X` to `Y`. Should have rightmost shape `(x_dim, y_dim)`.
- **loc** (`Funsor`) – A constant offset for `Y`'s mean. Should have output shape `(y_dim,)`.
- **scale** (`Funsor`) – Standard deviation for `Y`. Should have output shape `(y_dim,)`.
- **value_x** (`Funsor`) – A value `X`.
- **value_y** (`Funsor`) – A value `Y`.

matrix_and_mvn_to_funsor(*matrix*, *mvn*, *event_dims=()*, *x_name='value_x'*, *y_name='value_y'*)

Convert a noisy affine function to a Gaussian. The noisy affine function is defined as:

```
y = x @ matrix + mvn.sample()
```

The result is a non-normalized Gaussian funsor with two real inputs, `x_name` and `y_name`, corresponding to a conditional distribution of real vector `y` given real vector ``x`.

Parameters

- **matrix** (`torch.Tensor`) – A matrix with rightmost shape `(x_size, y_size)`.
- **mvn** (*torch.distributions.MultivariateNormal or torch.distributions.Independent of torch.distributions.Normal*) – A multivariate normal distribution with `event_shape == (y_size,)`.
- **event_dims** (*tuple*) – A tuple of names for rightmost dimensions. These will be assigned to `result.inputs` of type `bint`.
- **x_name** (*str*) – The name of the `x` random variable.
- **y_name** (*str*) – The name of the `y` random variable.

Returns A funsor with given `real_inputs` and possibly additional bint inputs.

Return type *funsor.terms.Funsor*

dist_to_funsor(*pyro_dist*, *event_inputs=()*)

Convert a PyTorch distribution to a Funsor.

This is currently implemented for only a subset of distribution types.

Parameters **torch.distribution.Distribution** – A PyTorch distribution.

Returns A funsor.

Return type *funsor.terms.Funsor*

CHAPTER 11

Distribution Funsors

This interface provides a number of standard normalized probability distributions implemented as funsors.

class Distribution(**args*)
Bases: *funsor.terms.Funsor*

Funsor backed by a PyTorch distribution object.

Parameters ***args** – Distribution-dependent parameters. These can be either funsors or objects that can be coerced to funsors via *to_funsor()* . See derived classes for details.

dist_class = 'defined by derived classes'

eager_reduce(*op*, *reduced_vars*)

classmethod eager_log_prob(***params*)

class BernoulliLogits(*logits*, *value=None*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Bernoulli .

Parameters

- **logits** (Funsor) – Log likelihood ratio of 1. This should equal log(p1 / p0).
- **value** (Funsor) – Optional observation in {0,1}.

dist_class
alias of pyro.distributions.torch.Bernoulli

Bernoulli(*probs=None*, *logits=None*, *value='value'*)
Wraps pyro.distributions.Bernoulli .

This dispatches to either BernoulliProbs or *BernoulliLogits* to accept either probs or logits args.

Parameters

- **probs** (Funsor) – Probability of 1.
- **value** (Funsor) – Optional observation in {0,1}.

class Beta(*concentration1*, *concentration0*, *value=None*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Beta .

Parameters

- **concentration1** (Funsor) – Positive concentration parameter.
- **concentration0** (Funsor) – Positive concentration parameter.
- **value** (Funsor) – Optional observation in (0,1).

dist_class
alias of pyro.distributions.torch.Beta

class Binomial(*total_count*, *probs*, *value=None*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Binomial .

Parameters

- **total_count** (Funsor) – Total number of trials.
- **probs** (Funsor) – Probability of each positive trial.
- **value** (Funsor) – Optional integer observation (encoded as "real").

dist_class
alias of pyro.distributions.torch.Binomial

class Categorical(*probs*, *value='value'*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Categorical .

Parameters

- **probs** (Funsor) – Probability vector over outcomes.
- **value** (Funsor) – Optional bouded integer observation.

dist_class
alias of pyro.distributions.torch.Categorical

class Delta(*v*, *log_density=0*, *value='value'*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Delta .

Parameters

- **v** (Funsor) – The unique point of concentration.
- **log_density** (Funsor) – Optional density (used by transformed distributions).
- **value** (Funsor) – Optional observation of similar domain as v.

dist_class
alias of pyro.distributions.delta.Delta

class Dirichlet(*concentration*, *value='value'*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Dirichlet .

Parameters

- **concentration** (Funsor) – Positive concentration vector.
- **value** (Funsor) – Optional observation in the unit simplex.

dist_class
alias of pyro.distributions.torch.Dirichlet

class DirichletMultinomial(*concentration*, *total_count*, *value='value'*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.DirichletMultinomial .

Parameters

- **concentration** (Funsor) – Positive concentration vector.
- **total_count** (Funsor) – Total number of trials.
- **value** (Funsor) – Optional observation in the unit simplex.

dist_class
alias of pyro.distributions.conjugate.DirichletMultinomial

LogNormal(*loc*, *scale*, *value='value'*)
Wraps pyro.distributions.LogNormal .

Parameters

- **loc** (Funsor) – Mean of the untransformed Normal distribution.
- **scale** (Funsor) – Standard deviation of the untransformed Normal distribution.
- **value** (Funsor) – Optional real observation.

class Multinomial(*total_count*, *probs*, *value=None*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Multinomial .

Parameters

- **probs** (Funsor) – Probability vector over outcomes.
- **total_count** (Funsor) – Total number of trials.
- **value** (Funsor) – Optional value in the unit simplex.

dist_class
alias of pyro.distributions.torch.Multinomial

class Normal(*loc*, *scale*, *value='value'*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Normal .

Parameters

- **loc** (Funsor) – Mean.
- **scale** (Funsor) – Standard deviation.
- **value** (Funsor) – Optional real observation.

dist_class
alias of pyro.distributions.torch.Normal

class MultivariateNormal(*loc*, *scale_tril*, *value='value'*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.MultivariateNormal .

Parameters

- **loc** (Funsor) – Mean vector.
- **scale_tril** (Funsor) – Lower Cholesky factor of the covariance matrix.
- **value** (Funsor) – Optional real vector observation.

dist_class
alias of pyro.distributions.torch.MultivariateNormal

class Poisson(*rate*, *value=None*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Poisson .

Parameters

- **rate** (Funsor) – Mean parameter.
- **value** (Funsor) – Optional integer observation (coded as "real").

dist_class
alias of pyro.distributions.torch.Poisson

class Gamma(*concentration*, *rate*, *value=None*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.Gamma .

Parameters

- **concentration** (Funsor) – Positive concentration parameter.
- **rate** (Funsor) – Positive rate parameter.
- **value** (Funsor) – Optional positive observation.

dist_class
alias of pyro.distributions.torch.Gamma

class VonMises(*loc*, *concentration*, *value=None*)
Bases: *funsor.distributions.Distribution*

Wraps pyro.distributions.VonMises .

Parameters

- **loc** (Funsor) – A location angle.
- **concentration** (Funsor) – Positive concentration parameter.
- **value** (Funsor) – Optional angular observation.

dist_class
alias of pyro.distributions.von_mises.VonMises

CHAPTER 12

Mini-Pyro Interface

This interface provides a backend for the Pyro probabilistic programming language. This interface is intended to be used indirectly by writing standard Pyro code and setting `pyro_backend("funsor")`. See examples/minipyro.py for example usage.

12.1 Mini Pyro

This file contains a minimal implementation of the Pyro Probabilistic Programming Language. The API (method signatures, etc.) match that of the full implementation as closely as possible. This file is independent of the rest of Pyro, with the exception of the `pyro.distributions` module.

An accompanying example that makes use of this implementation can be found at examples/minipyro.py.

class Distribution(*funsor_dist*, *sample_inputs=None*)
Bases: `object`

log_prob(*value*)

expand_inputs(*name*, *size*)

get_param_store()

class Messenger(*fn=None*)
Bases: `object`

process_message(*msg*)

postprocess_message(*msg*)

class trace(*fn=None*)
Bases: *funsor.minipyro.Messenger*

postprocess_message(*msg*)

get_trace(**args*, ***kwargs*)

class replay(*fn*, *guide_trace*)
Bases: *funsor.minipyro.Messenger*

process_message(*msg*)

class block(*fn=None*, *hide_fn=<function block.<lambda>>*)
Bases: *funsor.minipyro.Messenger*

process_message(*msg*)

class seed(*fn=None*, *rng_seed=None*)
Bases: *funsor.minipyro.Messenger*

class CondIndepStackFrame(*name*, *size*, *dim*)
Bases: tuple

dim
Alias for field number 2

name
Alias for field number 0

size
Alias for field number 1

class PlateMessenger(*fn*, *name*, *size*, *dim*)
Bases: *funsor.minipyro.Messenger*

process_message(*msg*)

tensor_to_funsor(*value*, *cond_indep_stack*, *output*)

class log_joint(*fn=None*)
Bases: *funsor.minipyro.Messenger*

process_message(*msg*)

postprocess_message(*msg*)

apply_stack(*msg*)

sample(*name*, *fn*, *obs=None*, *infer=None*)

param(*name*, *init_value=None*, *constraint=Real()*, *event_dim=None*)

plate(*name*, *size*, *dim*)

class PyroOptim(*optim_args*)
Bases: object

class Adam(*optim_args*)
Bases: *funsor.minipyro.PyroOptim*

TorchOptimizer
alias of torch.optim.adam.Adam

class ClippedAdam(*optim_args*)
Bases: *funsor.minipyro.PyroOptim*

TorchOptimizer
alias of pyro.optim.clipped_adam.ClippedAdam

class SVI(*model*, *guide*, *optim*, *loss*)
Bases: object

step(**args*, ***kwargs*)

Expectation(*log_probs*, *costs*, *sum_vars*, *prod_vars*)

elbo(*model*, *guide*, **args*, ***kwargs*)

class ELBO(***kwargs*)
Bases: object

class Trace_ELBO(***kwargs*)
Bases: *funsor.minipyro.ELBO*

class TraceMeanField_ELBO(***kwargs*)
Bases: *funsor.minipyro.ELBO*

class TraceEnum_ELBO(***kwargs*)
Bases: *funsor.minipyro.ELBO*

class Jit(*fn*, ***kwargs*)
Bases: object

class Jit_ELBO(*elbo*, ***kwargs*)
Bases: *funsor.minipyro.ELBO*

JitTrace_ELBO(***kwargs*)

JitTraceMeanField_ELBO(***kwargs*)

JitTraceEnum_ELBO(***kwargs*)

CHAPTER 13

Einsum Interface

This interface implements tensor variable elimination among tensors. In particular it does not implement continuous variable elimination.

naive_contract_einsum(*eqn*, **terms*, ***kwargs*)
Use for testing Contract against einsum

naive_einsum(*eqn*, **terms*, ***kwargs*)
Implements standard variable elimination.

naive_plated_einsum(*eqn*, **terms*, ***kwargs*)
Implements Tensor Variable Elimination (Algorithm 1 in [Obermeyer et al 2019])

[Obermeyer et al 2019] Obermeyer, F., Bingham, E., Jankowiak, M., Chiu, J., Pradhan, N., Rush, A., and Goodman, N. Tensor Variable Elimination for Plated Factor Graphs, 2019

einsum(*eqn*, **terms*, ***kwargs*)
Top-level interface for optimized tensor variable elimination.

Parameters

- **equation** (*str*) – An einsum equation.
- ***terms** (funsor.terms.Funsor) – One or more operands.
- **plates** (*set*) – Optional keyword argument denoting which funsor dimensions are plate dimensions. Among all input dimensions (from terms): dimensions in plates but not in outputs are product-reduced; dimensions in neither plates nor outputs are sum-reduced.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

X