
Funsor Documentation

Release 0.0

Uber AI Labs

Oct 15, 2020

1	Operations	1
1.1	Operation classes	1
1.2	Builtin operations	1
1.3	Array operations	3
2	Domains	5
3	Interpretations	7
3.1	Interpreter	7
3.2	Monte Carlo	8
3.3	Memoize	8
4	Funsors	9
4.1	Basic Funsors	9
4.2	Delta	14
4.3	Tensor	15
4.4	Gaussian	18
4.5	Joint	19
4.6	Contraction	19
4.7	Integrate	20
5	Optimizer	21
6	Adjoint Algorithms	23
7	Sum-Product Algorithms	25
8	Affine Pattern Matching	27
9	Testing Utilites	29
10	Pyro-Compatible Distributions	31
10.1	FunsorDistribution Base Class	31
10.2	Hidden Markov Models	32
10.3	Conversion Utilities	36
11	Distribution Funsors	39

12 Mini-Pyro Interface	43
12.1 Mini Pyro	43
13 Einsum Interface	47
14 Indices and tables	49
Python Module Index	51
Index	53

1.1 Operation classes

```
class Op (fn, *, name=None)
```

```
    Bases: multipledispatch.dispatcher.Dispatcher
```

```
class TransformOp (fn, *, name=None)
```

```
    Bases: funsor.ops.op.Op
```

```
    set_inv (fn)
```

Parameters **fn** (*callable*) – A function that inputs an arg y and outputs a value x such that $y = \text{self}(x)$.

```
    set_log_abs_det_jacobian (fn)
```

Parameters **fn** (*callable*) – A function that inputs two args x , y , where $y = \text{self}(x)$, and returns $\log(\text{abs}(\det(\text{dy}/\text{dx})))$.

```
    static inv (x)
```

```
    static log_abs_det_jacobian (x, y)
```

1.2 Builtin 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
nullop = ops.nullop
    Placeholder associative op that unifies with any other op
or_ = ops.or_
pow = ops.pow
reciprocal = ops.reciprocal
safediv = ops.safediv
safesub = ops.safesub
sigmoid = ops.sigmoid
sqrt = ops.sqrt
sub = ops.sub
truediv = ops.truediv
xor = ops.xor

class AssociativeOp(fn, *, name=None)
    Bases: funsor.ops.op.Op

class AddOp(fn, *, name=None)
    Bases: funsor.ops.builtin.AssociativeOp

class MulOp(fn, *, name=None)
    Bases: funsor.ops.builtin.AssociativeOp

class MatmulOp(fn, *, name=None)
    Bases: funsor.ops.op.Op

class SubOp(fn, *, name=None)
    Bases: funsor.ops.op.Op

class NegOp(fn, *, name=None)
    Bases: funsor.ops.op.Op
```

```

class DivOp (fn, *, name=None)
    Bases: funsor.ops.op.Op

class NullOp (fn, *, name=None)
    Bases: funsor.ops.builtin.AssociativeOp

    Placeholder associative op that unifies with any other op

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

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

class ExpOp (fn, *, name=None)
    Bases: funsor.ops.op.TransformOp

class LogOp (fn, *, name=None)
    Bases: funsor.ops.op.TransformOp

class ReciprocalOp (fn, *, name=None)
    Bases: funsor.ops.op.Op

```

1.3 Array operations

```

all = ops.all
amax = ops.amax
amin = ops.amin
any = ops.any
astype = ops.astype
cat = ops.cat
cholesky = ops.cholesky
cholesky_inverse = ops.cholesky_inverse
cholesky_solve = ops.cholesky_solve
clamp = ops.clamp
detach = ops.detach
diagonal = ops.diagonal
einsum = ops.einsum
expand = ops.expand
finfo = ops.finfo
full_like = ops.full_like
is_numeric_array = ops.is_numeric_array
logaddexp = ops.logaddexp
logsumexp = ops.logsumexp
new_arange = ops.new_arange
new_eye = ops.new_eye

```

```
new_zeros = ops.new_zeros
permute = ops.permute
prod = ops.prod
sample = ops.sample
stack = ops.stack
sum = ops.sum
transpose = ops.transpose
triangular_solve = ops.triangular_solve
unsqueeze = ops.unsqueeze
class LogAddExpOp (fn, *, name=None)
    Bases: funsor.ops.builtin.AssociativeOp
class SampleOp (fn, *, name=None)
    Bases: funsor.ops.array.LogAddExpOp
class ReshapeOp (shape)
    Bases: funsor.ops.op.Op
```


Domain

alias of `builtins.type`

class BintType

Bases: `functor.domains.ArrayType`

size

class RealsType

Bases: `functor.domains.ArrayType`

dtype = 'real'

class Bint

Bases: `object`

Factory for bounded integer types:

```
Bint[5]           # integers ranging in {0,1,2,3,4}
Bint[2, 3, 3]    # 3x3 matrices with entries in {0,1}
```

dtype = None

shape = None

class Reals

Bases: `object`

Type of a real-valued array with known shape:

```
Reals[()] = Real # scalar
Reals[8]    # vector of length 8
Reals[3, 3] # 3x3 matrix
```

shape = None

class Real

Bases: `object`

shape = ()

reals (*args)

bint (size)

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.

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.

class StatefulInterpretation

Bases: `object`

Base class for interpreters with instance-dependent state or parameters.

Example usage:

```
class MyInterpretation(StatefulInterpretation):  
  
    def __init__(self, my_param):  
        self.my_param = my_param  
  
@MyInterpretation.register(...)  
def my_impl(interpreter_state, cls, *args):  
    my_param = interpreter_state.my_param
```

(continues on next page)

(continued from previous page)

```
...  
with interpretation(MyInterpretation(my_param=0.1)):  
...
```

```
classmethod register(*args)  
classmethod dispatch(key, *args)  
registry = <funsor.registry.KeyedRegistry object>
```

```
exception PatternMissingError  
Bases: NotImplementedError
```

3.2 Monte Carlo

```
class MonteCarlo(*, rng_key=None, **sample_inputs)  
Bases: funsor.interpreter.StatefulInterpretation
```

A Monte Carlo interpretation of *Integrate* expressions. This falls back to the previous interpreter in other cases.

Parameters `rng_key` –

```
classmethod dispatch(key, *args)  
registry = <funsor.registry.KeyedRegistry object>
```

3.3 Memoize

```
memoize(cache=None)  
Exploit cons-hashing to do implicit common subexpression elimination
```

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; additionally 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, rng_key=None*)Create a Monte Carlo approximation to this funsor by replacing functions of `sampled_vars` with *Deltas*.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.
- **rng_key** (*None or JAX's random.PRNGKey*) – a PRNG state to be used by JAX backend to generate random samples

unscaled_sample (*sampled_vars, sample_inputs, rng_key=None*)

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.tensor.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_funsor (*x, output=None, dim_to_name=None, **kwargs*)

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

Parameters

- **x** – An object.
- **output** (`funsor.domains.Domain`) – An optional output hint.
- **dim_to_name** (`OrderedDict`) – An optional mapping from negative batch dimensions to name strings.

Returns A `Funsor` equivalent to `x`.

Return type `Funsor`

Raises `ValueError`

to_data (*x, name_to_dim=None, **kwargs*)

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`.
- **name_to_dim** (`OrderedDict`) – An optional inputs hint.

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*, *rng_key=None*)

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

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, rng_key=None*)

eager_subs (*subs*)

of_shape (**shape*)

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, rng_key=None)

```

4.3 Tensor

```
ignore_jit_warnings()
```

```
class Tensor (data, inputs=None, dtype='real')
```

Bases: `funsor.terms.Funsor`

Funsor backed by a PyTorch Tensor or a NumPy ndarray.

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** (*numeric_array*) – A PyTorch tensor or NumPy ndarray.
- **inputs** (*OrderedDict*) – An optional mapping from input name (*str*) to datatype (`funsor.domains.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, rng_key=None)
```

```
new_arange (name, *args, **kwargs)
```

Helper to create a named `torch.arange()` or `np.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*

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` or `np.ndarray` that can be broadcast to other tensors with inputs `new_inputs`.

Return type `int` or `float` or `torch.Tensor` or `np.ndarray`

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 or `np.ndarray` s that can be broadcast together to a single data with given `inputs`.

Return type `tuple`

class Function (*fn, output, args*)

Bases: *funsor.terms.Funsor*

Funsor wrapped by a native PyTorch or NumPy 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 native PyTorch or NumPy function to wrap.
- **output** (*type*) – An output domain.
- **args** (*Funsor*) – Funsor arguments.

function (**signature*)

Decorator to wrap a PyTorch/NumPy function, using either type hints or explicit type annotations.

Example:

```
# Using type hints:
@funsor.tensor.function
def matmul(x: Reals[3, 4], y: Reals[4, 5]) -> Reals[3, 5]:
    return torch.matmul(x, y)

# Using explicit type annotations:
@funsor.tensor.function(Reals[3, 4], Reals[4, 5], Reals[3, 5])
def matmul(x, y):
    return torch.matmul(x, y)

@funsor.tensor.function(Reals[10], Reals[10, 10], Reals[10], Real)
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.tensor.function
def max_and_argmax(x: Reals[8]) -> Tuple[Real, Bint[8]]:
    return torch.max(x, dim=-1)
```

Parameters **signature* – A sequence of 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()` or `np.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()` or `np.einsum()` equation.
- **operands** (*tuple*) – A tuple of input funsors.

tensordot (*x, y, dims*)

Wrapper around `torch.tensordot()` or `np.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*

stack (*parts*, *dim=0*)

Wrapper around `torch.stack()` or `np.stack()` to operate on real-valued Funsors.

Note this operates only on the output tensor. To stack funsors in a new named dim, instead use `Stack`.

Parameters

- **parts** (*tuple*) – A tuple of funsors.
- **dim** (*int*) – A torch dim along which to stack.

Return type *Funsor*

4.4 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:

$$f(x) = \langle x \mid \text{info_vec} \rangle - 0.5 * \langle x \mid \text{precision} \mid x \rangle \\ = \langle x \mid \text{info_vec} - 0.5 * \text{precision} @ x \rangle$$

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, rng_key=None*)

4.5 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*)

4.6 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, rng_key=None*)**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)

```

4.7 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.

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*)

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)
```

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 $O(\log(\text{time}))$.

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.

mixed_sequential_sum_product (*sum_op, prod_op, trans, time, step, num_segments=None*)

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")
```

by mixing parallel and serial scan algorithms over *num_segments* segments.

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.
- **num_segments** (*int*) – number of segments for the first stage

naive_sarkka_bilmes_product (*sum_op, prod_op, trans, time_var, global_vars=frozenset()*)

sarkka_bilmes_product (*sum_op, prod_op, trans, time_var, global_vars=frozenset(), num_periods=1*)

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*)

Affine Pattern Matching

is_affine (*fn*)

A sound but incomplete test to determine whether a functor is affine with respect to all of its real inputs.

Parameters *fn* (`Functor`) – A functor.

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* (`Functor`) – A functor.

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

Return type `frozenset`

extract_affine (*fn*)

Extracts an affine representation of a functor, 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* (`Functor`) – A functor that is affine wrt the (add,mul) semiring in some subset of its inputs.

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

Return type `tuple`


```
xfail_if_not_implemented (msg='Not implemented')  
class ActualExpected  
    Bases: funsor.testing.LazyComparison  
    Lazy string formatter for test assertions.  
id_from_inputs (inputs)  
is_array (x)  
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.  
rand (*args)  
randint (low, high, size)  
randn (*args)  
zeros (*args)  
ones (*args)  
empty (*args)  
random_tensor (inputs, output=Real)  
    Creates a random funsor.tensor.Tensor 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*)

Pyro-Compatible Distributions

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

10.1 FunctorDistribution Base Class

```
class FunctorDistribution (functor_dist, batch_shape=torch.Size([]), event_shape=torch.Size([]),
                          dtype='real', validate_args=None)
```

Bases: `pyro.distributions.torch_distribution.TorchDistribution`

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

Parameters

- **functor_dist** (`functor.terms.Functor`) – A functor 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 `functor_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)
```

```
functordistribution_to_functor (pyro_dist, output=None, dim_to_name=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 $O(\log(\text{time}))$ 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 $O(\log(\text{time}))$ 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 $O(\log(\text{time}))$ 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 $p(\text{class}[0])$.
- **initial_mvn** (*MultivariateNormal*) – Represents $p(\text{state}[0] \mid \text{class}[0])$.
- **transition_logits** (*Tensor*) – Represents $p(\text{class}[t+1] \mid \text{class}[t])$.
- **transition_matrix** (*Tensor*) –
- **transition_mvn** (*MultivariateNormal*) – Together with `transition_matrix`, this represents $p(\text{state}[t], \text{state}[t+1] \mid \text{class}[t])$.
- **observation_matrix** (*Tensor*) –
- **observation_mvn** (*MultivariateNormal*) – Together with `observation_matrix`, this represents $p(\text{value}[t+1], \text{state}[t+1] \mid \text{class}[t+1])$.
- **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.tensor.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.tensor.Tensor`

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

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

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

Parameters

- **funsor** (`funsor.tensor.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`

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

Convert a PyTorch distribution to a Funsor.

Parameters `torch.distributions.Distribution` – A PyTorch distribution.

Returns A funsor.

Return type `funsor.terms.Funsor`

mvn_to_funsor (*pyro_dist*, *event_inputs*=(), *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_inputs** (`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 `Real`. 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`

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/JAX 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)
```

```
    has_rsample
```

```
    has_enumerate_support
```

```
    unscaled_sample (sampled_vars, sample_inputs, rng_key=None)
```

```
    enumerate_support (expand=False)
```

```
class Beta(concentration1, concentration0, value='value')
```

```
Bases: funsor.distribution.Distribution
```

```
    dist_class
```

```
    alias of pyro.distributions.torch.Beta
```

```
class BernoulliProbs(probs, value='value')
```

```
Bases: funsor.distribution.Distribution
```

```
    dist_class
```

```
    alias of funsor.torch.distributions._PyroWrapper_BernoulliProbs
```

```
class BernoulliLogits(logits, value='value')
```

```
Bases: funsor.distribution.Distribution
```

```
dist_class
    alias of funsor.torch.distributions._PyroWrapper_BernoulliLogits

class Binomial (total_count, probs, value='value')
    Bases: funsor.distribution.Distribution

    dist_class
        alias of pyro.distributions.torch.Binomial

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

    dist_class
        alias of pyro.distributions.torch.Categorical

class CategoricalLogits (logits, value='value')
    Bases: funsor.distribution.Distribution

    dist_class
        alias of funsor.torch.distributions._PyroWrapper_CategoricalLogits

class Delta (v, log_density, value='value')
    Bases: funsor.distribution.Distribution

    dist_class
        alias of pyro.distributions.delta.Delta

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

    dist_class
        alias of pyro.distributions.torch.Dirichlet

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

    dist_class
        alias of pyro.distributions.conjugate.DirichletMultinomial

class Gamma (concentration, rate, value='value')
    Bases: funsor.distribution.Distribution

    dist_class
        alias of pyro.distributions.torch.Gamma

class GammaPoisson (concentration, rate, value='value')
    Bases: funsor.distribution.Distribution

    dist_class
        alias of pyro.distributions.conjugate.GammaPoisson

class Multinomial (total_count, probs, value='value')
    Bases: funsor.distribution.Distribution

    dist_class
        alias of pyro.distributions.torch.Multinomial

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

    dist_class
        alias of pyro.distributions.torch.MultivariateNormal
```

```
class NonreparameterizedBeta (concentration1, concentration0, value='value')  
    Bases: funsor.distribution.Distribution  
  
    dist_class  
        alias of pyro.distributions.testing.fakes.NonreparameterizedBeta  
  
class NonreparameterizedDirichlet (concentration, value='value')  
    Bases: funsor.distribution.Distribution  
  
    dist_class  
        alias of pyro.distributions.testing.fakes.NonreparameterizedDirichlet  
  
class NonreparameterizedGamma (concentration, rate, value='value')  
    Bases: funsor.distribution.Distribution  
  
    dist_class  
        alias of pyro.distributions.testing.fakes.NonreparameterizedGamma  
  
class NonreparameterizedNormal (loc, scale, value='value')  
    Bases: funsor.distribution.Distribution  
  
    dist_class  
        alias of pyro.distributions.testing.fakes.NonreparameterizedNormal  
  
class Normal (loc, scale, value='value')  
    Bases: funsor.distribution.Distribution  
  
    dist_class  
        alias of pyro.distributions.torch.Normal  
  
class Poisson (rate, value='value')  
    Bases: funsor.distribution.Distribution  
  
    dist_class  
        alias of pyro.distributions.torch.Poisson  
  
class VonMises (loc, concentration, value='value')  
    Bases: funsor.distribution.Distribution  
  
    dist_class  
        alias of pyro.distributions.torch.VonMises
```

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("functor")`. 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 (functor_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: functor.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*)

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** (*functor.terms.Functor*) – One or more operands.
- **plates** (*set*) – Optional keyword argument denoting which functor 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`

f

`functor.adjoint`, 23
`functor.affine`, 27
`functor.cnf`, 19
`functor.delta`, 14
`functor.domains`, 5
`functor.einsum`, 47
`functor.gaussian`, 18
`functor.integrate`, 20
`functor.interpreter`, 7
`functor.joint`, 19
`functor.memoize`, 8
`functor.minipyro`, 43
`functor.montecarlo`, 8
`functor.ops.array`, 3
`functor.ops.builtin`, 1
`functor.ops.op`, 1
`functor.optimizer`, 21
`functor.pyro.convert`, 36
`functor.pyro.distribution`, 31
`functor.pyro.hmm`, 32
`functor.sum_product`, 25
`functor.tensor`, 15
`functor.terms`, 9
`functor.testing`, 29
`functor.torch.distributions`, 39

A

abs (in module *funsor.ops.builtin*), 1
 abs () (*Funsor* method), 11
 ActualExpected (class in *funsor.testing*), 29
 Adam (class in *funsor.minipyro*), 44
 add (in module *funsor.ops.builtin*), 1
 AddOp (class in *funsor.ops.builtin*), 2
 adjoint () (*AdjointTape* method), 23
 adjoint_binary () (in module *funsor.adjoint*), 23
 adjoint_cat () (in module *funsor.adjoint*), 23
 adjoint_contract () (in module *funsor.adjoint*), 23
 adjoint_contract_generic () (in module *funsor.adjoint*), 23
 adjoint_contract_unary () (in module *funsor.adjoint*), 23
 adjoint_reduce () (in module *funsor.adjoint*), 23
 adjoint_subs_gaussian_gaussian () (in module *funsor.adjoint*), 23
 adjoint_subs_gaussianmixture_discrete () (in module *funsor.adjoint*), 23
 adjoint_subs_gaussianmixture_gaussianmixture () (in module *funsor.adjoint*), 23
 adjoint_subs_tensor () (in module *funsor.adjoint*), 23
 adjoint_tensor () (in module *funsor.adjoint*), 23
 AdjointTape (class in *funsor.adjoint*), 23
 affine_inputs () (in module *funsor.affine*), 27
 AffineNormal (class in *funsor.pyro.convert*), 37
 align () (*Contraction* method), 19
 align () (*Delta* method), 14
 align () (*Funsor* method), 10
 align () (*Gaussian* method), 19
 align () (*Tensor* method), 15
 align_gaussian () (in module *funsor.gaussian*), 18
 align_tensor () (in module *funsor.tensor*), 16
 align_tensors () (in module *funsor.tensor*), 16
 all (in module *funsor.ops.array*), 3
 all () (*Funsor* method), 11
 amax (in module *funsor.ops.array*), 3

amin (in module *funsor.ops.array*), 3
 and_ (in module *funsor.ops.builtin*), 1
 any (in module *funsor.ops.array*), 3
 any () (*Funsor* method), 11
 apply_optimizer () (in module *funsor.optimizer*), 21
 apply_stack () (in module *funsor.minipyro*), 44
 arg_constraints (*FunsorDistribution* attribute), 31
 arg_constraints (*GaussianHMM* attribute), 33
 arg_constraints (*SwitchingLinearHMM* attribute), 35
 as_tensor () (*BlockMatrix* method), 18
 as_tensor () (*BlockVector* method), 18
 assert_close () (in module *funsor.testing*), 29
 assert_equiv () (in module *funsor.testing*), 29
 AssociativeOp (class in *funsor.ops.builtin*), 2
 astype (in module *funsor.ops.array*), 3

B

BernoulliLogits (class in *funsor.torch.distributions*), 39
 BernoulliProbs (class in *funsor.torch.distributions*), 39
 Beta (class in *funsor.torch.distributions*), 39
 Binary (class in *funsor.terms*), 12
 binary_divide () (in module *funsor.cnf*), 20
 binary_subtract () (in module *funsor.cnf*), 20
 binary_to_contract () (in module *funsor.cnf*), 20
 Binomial (class in *funsor.torch.distributions*), 40
 Bint (class in *funsor.domains*), 5
 bint () (in module *funsor.domains*), 6
 BintType (class in *funsor.domains*), 5
 block (class in *funsor.minipyro*), 44
 BlockMatrix (class in *funsor.gaussian*), 18
 BlockVector (class in *funsor.gaussian*), 18

C

Cat (class in *funsor.terms*), 13
 cat (in module *funsor.ops.array*), 3

Categorical (class in *funsor.torch.distributions*), 40
 CategoricalLogits (class in *funsor.torch.distributions*), 40
 check_funsor() (in module *funsor.testing*), 29
 cholesky (in module *funsor.ops.array*), 3
 cholesky_inverse (in module *funsor.ops.array*), 3
 cholesky_solve (in module *funsor.ops.array*), 3
 clamp (in module *funsor.ops.array*), 3
 clamp_finite() (Tensor method), 15
 ClippedAdam (class in *funsor.minipyro*), 44
 CondIndepStackFrame (class in *funsor.minipyro*), 44
 Contraction (class in *funsor.cnf*), 19

D

Delta (class in *funsor.delta*), 14
 Delta (class in *funsor.torch.distributions*), 40
 detach (in module *funsor.ops.array*), 3
 diagonal (in module *funsor.ops.array*), 3
 dim (CondIndepStackFrame attribute), 44
 Dirichlet (class in *funsor.torch.distributions*), 40
 DirichletMultinomial (class in *funsor.torch.distributions*), 40
 DiscreteHMM (class in *funsor.pyro.hmm*), 32
 dispatch() (*funsor.interpreter.StatefulInterpretation* class method), 8
 dispatch() (*funsor.montecarlo.MonteCarlo* class method), 8
 dispatched_interpretation() (in module *funsor.interpreter*), 7
 dist_class (BernoulliLogits attribute), 39
 dist_class (BernoulliProbs attribute), 39
 dist_class (Beta attribute), 39
 dist_class (Binomial attribute), 40
 dist_class (Categorical attribute), 40
 dist_class (CategoricalLogits attribute), 40
 dist_class (Delta attribute), 40
 dist_class (Dirichlet attribute), 40
 dist_class (DirichletMultinomial attribute), 40
 dist_class (Distribution attribute), 39
 dist_class (Gamma attribute), 40
 dist_class (GammaPoisson attribute), 40
 dist_class (Multinomial attribute), 40
 dist_class (MultivariateNormal attribute), 40
 dist_class (NonreparameterizedBeta attribute), 41
 dist_class (NonreparameterizedDirichlet attribute), 41
 dist_class (NonreparameterizedGamma attribute), 41
 dist_class (NonreparameterizedNormal attribute), 41
 dist_class (Normal attribute), 41
 dist_class (Poisson attribute), 41
 dist_class (VonMises attribute), 41

dist_to_funsor() (in module *funsor.pyro.convert*), 36
 distribute_subs_contraction() (in module *funsor.cnf*), 20
 Distribution (class in *funsor.distribution*), 39
 Distribution (class in *funsor.minipyro*), 43
 DivOp (class in *funsor.ops.builtin*), 2
 do_fresh_subs() (in module *funsor.cnf*), 20
 Domain (in module *funsor.domains*), 5
 dtype (Bint attribute), 5
 dtype (Funsor attribute), 10
 dtype (RealsType attribute), 5

E

eager() (in module *funsor.terms*), 9
 eager_contract_base() (in module *funsor.optimizer*), 21
 eager_contraction_gaussian() (in module *funsor.cnf*), 19
 eager_contraction_generic_recursive() (in module *funsor.cnf*), 19
 eager_contraction_generic_to_tuple() (in module *funsor.cnf*), 19
 eager_contraction_tensor() (in module *funsor.cnf*), 19
 eager_contraction_to_binary() (in module *funsor.cnf*), 19
 eager_contraction_to_reduce() (in module *funsor.cnf*), 19
 eager_independent_joint() (in module *funsor.joint*), 19
 eager_log_prob() (*funsor.distribution.Distribution* class method), 39
 eager_markov_product() (in module *funsor.sum_product*), 26
 eager_or_die() (in module *funsor.terms*), 9
 eager_reduce() (Delta method), 15
 eager_reduce() (Distribution method), 39
 eager_reduce() (Funsor method), 11
 eager_reduce() (Gaussian method), 19
 eager_reduce() (Stack method), 13
 eager_reduce() (Tensor method), 15
 eager_reduce_exp() (in module *funsor.joint*), 19
 eager_subs() (Cat method), 13
 eager_subs() (Delta method), 15
 eager_subs() (Funsor method), 11
 eager_subs() (Gaussian method), 19
 eager_subs() (Independent method), 14
 eager_subs() (MarkovProduct method), 26
 eager_subs() (Slice method), 13
 eager_subs() (Stack method), 13
 eager_subs() (Tensor method), 15
 eager_subs() (Variable method), 12
 eager_unary() (Funsor method), 11

eager_unary() (*Number method*), 13
 eager_unary() (*Tensor method*), 15
 Einsum (*class in funsor.tensor*), 17
 einsum (*in module funsor.ops.array*), 3
 einsum() (*in module funsor.einsum*), 47
 ELBO (*class in funsor.minipyro*), 45
 elbo() (*in module funsor.minipyro*), 45
 empty() (*in module funsor.testing*), 29
 enumerate_support() (*Distribution method*), 39
 eq (*in module funsor.ops.builtin*), 1
 exp (*in module funsor.ops.builtin*), 1
 exp() (*Funsor method*), 11
 expand (*in module funsor.ops.array*), 3
 expand() (*DiscreteHMM method*), 32
 expand() (*FunsorDistribution method*), 31
 expand() (*SwitchingLinearHMM method*), 35
 expand_inputs() (*Distribution method*), 43
 Expectation() (*in module funsor.minipyro*), 44
 ExpOp (*class in funsor.ops.builtin*), 3
 extract_affine() (*in module funsor.affine*), 27

F

filter() (*SwitchingLinearHMM method*), 35
 find_domain() (*in module funsor.domains*), 6
 finfo (*in module funsor.ops.array*), 3
 full_like (*in module funsor.ops.array*), 3
 Function (*class in funsor.tensor*), 16
 function() (*in module funsor.tensor*), 16
 Funsor (*class in funsor.terms*), 9
 funsor.adjoint (*module*), 23
 funsor.affine (*module*), 27
 funsor.cnf (*module*), 19
 funsor.delta (*module*), 14
 funsor.domains (*module*), 5
 funsor.einsum (*module*), 47
 funsor.gaussian (*module*), 18
 funsor.integrate (*module*), 20
 funsor.interpreter (*module*), 7
 funsor.joint (*module*), 19
 funsor.memoize (*module*), 8
 funsor.minipyro (*module*), 43
 funsor.montecarlo (*module*), 8
 funsor.ops.array (*module*), 3
 funsor.ops.builtin (*module*), 1
 funsor.ops.op (*module*), 1
 funsor.optimizer (*module*), 21
 funsor.pyro.convert (*module*), 36
 funsor.pyro.distribution (*module*), 31
 funsor.pyro.hmm (*module*), 32
 funsor.sum_product (*module*), 25
 funsor.tensor (*module*), 15
 funsor.terms (*module*), 9
 funsor.testing (*module*), 29
 funsor.torch.distributions (*module*), 39

funsor_to_cat_and_mvn() (*in module funsor.pyro.convert*), 37
 funsor_to_mvn() (*in module funsor.pyro.convert*), 37
 funsor_to_tensor() (*in module funsor.pyro.convert*), 36
 FunsorDistribution (*class in funsor.pyro.distribution*), 31
 funsordistribution_to_funsor() (*in module funsor.pyro.distribution*), 31

G

Gamma (*class in funsor.torch.distributions*), 40
 GammaPoisson (*class in funsor.torch.distributions*), 40
 Gaussian (*class in funsor.gaussian*), 18
 GaussianHMM (*class in funsor.pyro.hmm*), 32
 GaussianMixture (*in module funsor.cnf*), 19
 GaussianMRF (*class in funsor.pyro.hmm*), 33
 ge (*in module funsor.ops.builtin*), 1
 get_param_store() (*in module funsor.minipyro*), 43
 get_trace() (*trace method*), 43
 getitem (*in module funsor.ops.builtin*), 1
 GetitemOp (*class in funsor.ops.builtin*), 3
 gt (*in module funsor.ops.builtin*), 2

H

has_enumerate_support (*Distribution attribute*), 39
 has_rsampl (DiscreteHMM attribute), 32
 has_rsampl (Distribution attribute), 39
 has_rsampl (GaussianHMM attribute), 33
 has_rsampl (GaussianMRF attribute), 34
 has_rsampl (SwitchingLinearHMM attribute), 35

I

id_from_inputs() (*in module funsor.testing*), 29
 ignore_jit_warnings() (*in module funsor.tensor*), 15
 Independent (*class in funsor.terms*), 14
 Integrate (*class in funsor.integrate*), 20
 interpretation() (*in module funsor.interpreter*), 7
 inv() (*TransformOp static method*), 1
 invert (*in module funsor.ops.builtin*), 2
 is_affine() (*in module funsor.affine*), 27
 is_array() (*in module funsor.testing*), 29
 is_numeric_array (*in module funsor.ops.array*), 3
 item() (*Funsor method*), 10
 item() (*Number method*), 13
 item() (*Tensor method*), 15

J

Jit (*class in funsor.minipyro*), 45
 Jit_ELBO (*class in funsor.minipyro*), 45

JitTrace_ELBO() (in module *funsor.minipyro*), 45
 JitTraceEnum_ELBO() (in module *funsor.minipyro*), 45
 JitTraceMeanField_ELBO() (in module *funsor.minipyro*), 45

L

Lambda (class in *funsor.terms*), 13
 lazy() (in module *funsor.terms*), 9
 le (in module *funsor.ops.builtin*), 2
 log (in module *funsor.ops.builtin*), 2
 log() (Funsor method), 11
 loglp (in module *funsor.ops.builtin*), 2
 loglp() (Funsor method), 11
 log_abs_det_jacobian() (TransformOp static method), 1
 log_joint (class in *funsor.minipyro*), 44
 log_normalizer (Gaussian attribute), 19
 log_prob() (DiscreteHMM method), 32
 log_prob() (Distribution method), 43
 log_prob() (FunsorDistribution method), 31
 log_prob() (SwitchingLinearHMM method), 35
 logaddexp (in module *funsor.ops.array*), 3
 LogAddExpOp (class in *funsor.ops.array*), 4
 LogOp (class in *funsor.ops.builtin*), 3
 logsumexp (in module *funsor.ops.array*), 3
 logsumexp() (Funsor method), 11
 lt (in module *funsor.ops.builtin*), 2

M

make_chain_einsum() (in module *funsor.testing*), 30
 make_einsum_example() (in module *funsor.testing*), 29
 make_hmm_einsum() (in module *funsor.testing*), 30
 make_plated_hmm_einsum() (in module *funsor.testing*), 30
 MarkovProduct (class in *funsor.sum_product*), 26
 MarkovProductMeta (class in *funsor.sum_product*), 26
 materialize() (Tensor method), 16
 matmul (in module *funsor.ops.builtin*), 2
 MatmulOp (class in *funsor.ops.builtin*), 2
 matrix_and_mvn_to_funsor() (in module *funsor.pyro.convert*), 38
 max (in module *funsor.ops.builtin*), 2
 max() (Funsor method), 11
 memoize() (in module *funsor.memoize*), 8
 Messenger (class in *funsor.minipyro*), 43
 min (in module *funsor.ops.builtin*), 2
 min() (Funsor method), 11
 mixed_sequential_sum_product() (in module *funsor.sum_product*), 26
 moment_matching() (in module *funsor.terms*), 9

moment_matching_contract_default() (in module *funsor.joint*), 19
 moment_matching_contract_joint() (in module *funsor.joint*), 19
 moment_matching_reduce() (Funsor method), 11
 MonteCarlo (class in *funsor.montecarlo*), 8
 mul (in module *funsor.ops.builtin*), 2
 MulOp (class in *funsor.ops.builtin*), 2
 Multinomial (class in *funsor.torch.distributions*), 40
 MultivariateNormal (class in *funsor.torch.distributions*), 40
 mvn_to_funsor() (in module *funsor.pyro.convert*), 36

N

naive_contract_einsum() (in module *funsor.einsum*), 47
 naive_einsum() (in module *funsor.einsum*), 47
 naive_plated_einsum() (in module *funsor.einsum*), 47
 naive_sarkka_bilmes_product() (in module *funsor.sum_product*), 26
 naive_sequential_sum_product() (in module *funsor.sum_product*), 25
 name (CondIndepStackFrame attribute), 44
 ne (in module *funsor.ops.builtin*), 2
 neg (in module *funsor.ops.builtin*), 2
 NegOp (class in *funsor.ops.builtin*), 2
 new_arange (in module *funsor.ops.array*), 3
 new_arange() (Tensor method), 15
 new_eye (in module *funsor.ops.array*), 3
 new_zeros (in module *funsor.ops.array*), 3
 NonreparameterizedBeta (class in *funsor.torch.distributions*), 40
 NonreparameterizedDirichlet (class in *funsor.torch.distributions*), 41
 NonreparameterizedGamma (class in *funsor.torch.distributions*), 41
 NonreparameterizedNormal (class in *funsor.torch.distributions*), 41
 Normal (class in *funsor.torch.distributions*), 41
 normalize_contraction_commutative_canonical_order() (in module *funsor.cnf*), 19
 normalize_contraction_commute_joint() (in module *funsor.cnf*), 20
 normalize_contraction_generic_args() (in module *funsor.cnf*), 20
 normalize_contraction_generic_tuple() (in module *funsor.cnf*), 20
 normalize_fuse_subs() (in module *funsor.cnf*), 20
 normalize_trivial() (in module *funsor.cnf*), 20
 NullOp (class in *funsor.ops.builtin*), 3
 nullop (in module *funsor.ops.builtin*), 2

Number (class in *funsor.terms*), 12

O

of_shape() (in module *funsor.terms*), 14

ones() (in module *funsor.testing*), 29

Op (class in *funsor.ops.op*), 1

optimize() (in module *funsor.optimizer*), 21

optimize_contract_finitary_funsor() (in module *funsor.optimizer*), 21

or_ (in module *funsor.ops.builtin*), 2

P

param() (in module *funsor.minipyro*), 44

partial_sum_product() (in module *funsor.sum_product*), 25

PatternMissingError, 8

permute (in module *funsor.ops.array*), 4

plate() (in module *funsor.minipyro*), 44

PlateMessenger (class in *funsor.minipyro*), 44

Poisson (class in *funsor.torch.distributions*), 41

postprocess_message() (log_joint method), 44

postprocess_message() (Messenger method), 43

postprocess_message() (trace method), 43

pow (in module *funsor.ops.builtin*), 2

pretty() (Funsor method), 10

process_message() (block method), 44

process_message() (log_joint method), 44

process_message() (Messenger method), 43

process_message() (PlateMessenger method), 44

process_message() (replay method), 44

prod (in module *funsor.ops.array*), 4

prod() (Funsor method), 11

PyroOptim (class in *funsor.minipyro*), 44

Q

quote() (Funsor method), 10

R

rand() (in module *funsor.testing*), 29

randint() (in module *funsor.testing*), 29

randn() (in module *funsor.testing*), 29

random_gaussian() (in module *funsor.testing*), 29

random_mvn() (in module *funsor.testing*), 29

random_tensor() (in module *funsor.testing*), 29

Real (class in *funsor.domains*), 5

Reals (class in *funsor.domains*), 5

reals() (in module *funsor.domains*), 6

RealsType (class in *funsor.domains*), 5

reciprocal (in module *funsor.ops.builtin*), 2

ReciprocalOp (class in *funsor.ops.builtin*), 3

recursion_reinterpret_contraction() (in module *funsor.cnf*), 19

Reduce (class in *funsor.terms*), 12

reduce() (Funsor method), 10

reduce_funsor() (in module *funsor.cnf*), 20

reflect() (in module *funsor.terms*), 9

register() (funsor.interpreter.StatefulInterpretation class method), 8

registry (MonteCarlo attribute), 8

registry (StatefulInterpretation attribute), 8

reinterpret() (in module *funsor.interpreter*), 7

replay (class in *funsor.minipyro*), 43

requires_grad (Funsor attribute), 10

requires_grad (Tensor attribute), 15

reshape() (Funsor method), 11

ReshapeOp (class in *funsor.ops.array*), 4

rsample() (FunsorDistribution method), 31

S

safediv (in module *funsor.ops.builtin*), 2

safesub (in module *funsor.ops.builtin*), 2

sample (in module *funsor.ops.array*), 4

sample() (Funsor method), 10

sample() (FunsorDistribution method), 31

sample() (in module *funsor.minipyro*), 44

SampleOp (class in *funsor.ops.array*), 4

sarkka_bilmes_product() (in module *funsor.sum_product*), 26

seed (class in *funsor.minipyro*), 44

sequential() (in module *funsor.terms*), 9

sequential_reduce() (Funsor method), 11

sequential_sum_product() (in module *funsor.sum_product*), 25

set_interpretation() (in module *funsor.interpreter*), 7

set_inv() (TransformOp method), 1

set_log_abs_det_jacobian() (TransformOp method), 1

shape (Bint attribute), 5

shape (Funsor attribute), 10

shape (Real attribute), 5

shape (Reals attribute), 5

sigmoid (in module *funsor.ops.builtin*), 2

sigmoid() (Funsor method), 11

size (BintType attribute), 5

size (CondIndepStackFrame attribute), 44

Slice (class in *funsor.terms*), 13

solve() (in module *funsor.delta*), 14

sqrt (in module *funsor.ops.builtin*), 2

sqrt() (Funsor method), 11

Stack (class in *funsor.terms*), 13

stack (in module *funsor.ops.array*), 4

stack() (in module *funsor.tensor*), 17

StatefulInterpretation (class in *funsor.interpreter*), 7

step() (SVI method), 44

sub (in module *funsor.ops.builtin*), 2

SubOp (*class in funsor.ops.builtin*), 2
 Subs (*class in funsor.terms*), 12
 sum (*in module funsor.ops.array*), 4
 sum() (*Funsor method*), 11
 sum_product() (*in module funsor.sum_product*), 25
 support (*FunsorDistribution attribute*), 31
 SVI (*class in funsor.minipyro*), 44
 SwitchingLinearHMM (*class in funsor.pyro.hmm*), 34

T

Tensor (*class in funsor.tensor*), 15
 tensor_to_funsor() (*in module funsor.minipyro*),
 44
 tensor_to_funsor() (*in module fun-*
sor.pyro.convert), 36
 tensordot() (*in module funsor.tensor*), 17
 to_data() (*in module funsor.terms*), 11
 to_funsor() (*in module funsor.terms*), 11
 TorchOptimizer (*Adam attribute*), 44
 TorchOptimizer (*ClippedAdam attribute*), 44
 trace (*class in funsor.minipyro*), 43
 Trace_ELBO (*class in funsor.minipyro*), 45
 TraceEnum_ELBO (*class in funsor.minipyro*), 45
 TraceMeanField_ELBO (*class in funsor.minipyro*),
 45
 TransformOp (*class in funsor.ops.op*), 1
 transpose (*in module funsor.ops.array*), 4
 triangular_solve (*in module funsor.ops.array*), 4
 truediv (*in module funsor.ops.builtin*), 2

U

Unary (*class in funsor.terms*), 12
 unary_contract() (*in module funsor.cnf*), 20
 unary_log_exp() (*in module funsor.cnf*), 20
 unary_neg_variable() (*in module funsor.cnf*), 20
 unfold() (*in module funsor.optimizer*), 21
 unfold_contraction_generic_tuple() (*in*
module funsor.optimizer), 21
 unscaled_sample() (*Contraction method*), 19
 unscaled_sample() (*Delta method*), 15
 unscaled_sample() (*Distribution method*), 39
 unscaled_sample() (*Funsor method*), 10
 unscaled_sample() (*Gaussian method*), 19
 unscaled_sample() (*Independent method*), 14
 unscaled_sample() (*Subs method*), 12
 unscaled_sample() (*Tensor method*), 15
 unsqueeze (*in module funsor.ops.array*), 4

V

Variable (*class in funsor.terms*), 12
 VonMises (*class in funsor.torch.distributions*), 41

X

xfail_if_not_implemented() (*in module fun-*
sor.testing), 29
 xfail_param() (*in module funsor.testing*), 29
 xor (*in module funsor.ops.builtin*), 2

Z

zeros() (*in module funsor.testing*), 29