
Funsor Documentation

Release 0.0

Uber AI Labs

Aug 31, 2023

FUNSOR CORE:

1	Operations	1
1.1	Operation classes	1
1.2	Builtin operations	3
1.3	Array operations	5
2	Domains	7
3	Interpretations	9
3.1	Interpreter	9
3.2	Interpretations	9
3.3	Monte Carlo	11
3.4	Preconditioning	11
3.5	Approximations	12
3.6	Evidence lower bound	13
4	Funsors	15
4.1	Basic Funsors	15
4.2	Delta	23
4.3	Tensor	24
4.4	Gaussian	28
4.5	Joint	29
4.6	Contraction	30
4.7	Integrate	31
4.8	Constant	31
5	Optimizer	33
6	Adjoint Algorithms	35
7	Sum-Product Algorithms	37
8	Affine Pattern Matching	41
9	Funsor Factory	43
10	Testing Utilites	45
11	Typing Utilites	47
12	Recipes using Funsor	51

13	Pyro-Compatible Distributions	53
13.1	FunsorDistribution Base Class	53
13.2	Hidden Markov Models	54
13.3	Conversion Utilities	58
14	Distribution Funsors	61
15	Mini-Pyro Interface	65
15.1	Mini Pyro	65
16	Einsum Interface	69
17	Compiler & Tracer	71
18	Named tensor notation with funsors (Part 1)	73
18.1	Introduction	73
18.2	Named Tensors	73
18.3	Named tensor operations	74
18.4	Advanced indexing	79
19	Example: Adam optimizer	81
20	Example: Discrete HMM	83
21	Example: Switching Linear Dynamical System EEG	85
22	Example: Forward-Backward algorithm	97
23	Example: Kalman Filter	101
24	Example: Mini Pyro	103
25	Example: PCFG	105
26	Example: Biased Kalman Filter	107
27	Example: Switching Linear Dynamical System	115
28	Example: Talbot’s method for numerical inversion of the Laplace transform	117
29	Example: VAE MNIST	121
30	Indices and tables	125
	Python Module Index	127
	Index	129

OPERATIONS

1.1 Operation classes

```
class BinaryOp(*args, **kwargs)
```

Bases: *Op*

```
    arity = 2
```

```
class FinitaryOp(*args, **kwargs)
```

Bases: *Op*

```
    arity = 1
```

```
class LogAbsDetJacobianOp(*args, **kwargs)
```

Bases: *BinaryOp*

```
    static default(x, y, fn)
```

```
    dispatcher = <dispatched log_abs_det_jacobian>
```

```
    name = 'log_abs_det_jacobian'
```

```
    signature = <Signature (x, y, fn)>
```

```
class NullaryOp(*args, **kwargs)
```

Bases: *Op*

```
    arity = 0
```

```
class Op(*args, **kwargs)
```

Bases: *object*

Abstract base class for all mathematical operations on ground terms.

Ops take `arity`-many leftmost positional args that may be functors, followed by additional non-functor args and kwargs. The additional args and kwargs must have default values.

When wrapping new backend ops, keep in mind these restrictions, which may require you to wrap backend functions before making them into ops:

- Create new ops only by decorating a default implementation with `@UnaryOp.make`, `@BinaryOp.make`, etc.
- Register backend-specific implementations via `@my_op.register(type1)`, `@my_op.register(type1, type2)` etc for arity 1, 2, etc. Patterns may include only the first `arity`-many types.
- Only the first `arity`-many arguments may be functors. Remaining args and kwargs must all be ground Python data.

Variables

`~.arity` (*int*) – The number of funsor arguments this op takes. Must be defined by subclasses.

Parameters

- `*args` –
- `**kwargs` – All extra arguments to this op, excluding the arguments up to `.arity`,

`arity = NotImplemented`

`register(*pattern)`

`classmethod subclass_register(*pattern)`

`classmethod make(fn=None, *, name=None, metaclass=None, module_name='funsor.ops')`

Factory to create a new *Op* subclass together with a new default instance of that class.

Parameters

`fn` (*callable*) – A function whose signature can be inspected.

Returns

The new default instance.

Return type

Op

`class TernaryOp(*args, **kwargs)`

Bases: *Op*

`arity = 3`

`class TransformOp(*args, **kwargs)`

Bases: *UnaryOp*

`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)`

`class UnaryOp(*args, **kwargs)`

Bases: *Op*

`arity = 1`

`class WrappedTransformOp(*args, **kwargs)`

Bases: *TransformOp*

Wrapper for a backend Transform object that provides `.inv` and `.log_abs_det_jacobian`. This additionally validates shapes on the first `__call__()`.

```
static default(x, fn, *, validate_args=True)
```

Wrapper for a backend Transform object that provides `.inv` and `.log_abs_det_jacobian`. This additionally validates shapes on the first `__call__()`.

```
dispatcher = <dispatched wrapped_transform>
```

```
property inv
```

```
property log_abs_det_jacobian
```

```
name = 'wrapped_transform'
```

```
signature = <Signature (x, fn, *, validate_args=True)>
```

```
declare_op_types(locals_, all_, name_)
```

1.2 Builtin operations

```
abs = ops.abs
```

Return the absolute value of the argument.

```
add = ops.add
```

Same as `a + b`.

```
and_ = ops.and_
```

Same as `a & b`.

```
atanh = ops.atanh
```

Return the inverse hyperbolic tangent of `x`.

```
eq = ops.eq
```

Same as `a == b`.

```
exp = ops.exp
```

Return `e` raised to the power of `x`.

```
floordiv = ops.floordiv
```

Same as `a // b`.

```
ge = ops.ge
```

Same as `a >= b`.

```
getitem = ops.getitem
```

```
getslice = ops.getslice
```

```
gt = ops.gt
```

Same as `a > b`.

```
invert = ops.invert
```

Same as `~a`.

```
le = ops.le
```

Same as `a <= b`.

lgamma = ops.lgamma

Natural logarithm of absolute value of Gamma function at x.

log = ops.log

log1p = ops.log1p

Return the natural logarithm of 1+x (base e).

The result is computed in a way which is accurate for x near zero.

lshift = ops.lshift

Same as $a \ll b$.

lt = ops.lt

Same as $a < b$.

matmul = ops.matmul

Same as $a @ b$.

max = ops.max

min = ops.min

mod = ops.mod

Same as $a \% b$.

mul = ops.mul

Same as $a * b$.

ne = ops.ne

Same as $a != b$.

neg = ops.neg

Same as $-a$.

null = ops.null

Placeholder associative op that unifies with any other op

or_ = ops.or_

Same as $a | b$.

pos = ops.pos

Same as $+a$.

pow = ops.pow

Same as $a ** b$.

reciprocal = ops.reciprocal

rshift = ops.rshift

Same as $a \gg b$.

safediv = ops.safediv

safesub = ops.safesub

sigmoid = ops.sigmoid

sqrt = **ops.sqrt**

Return the square root of x.

sub = **ops.sub**

Same as $a - b$.

tanh = **ops.tanh**

Return the hyperbolic tangent of x.

truediv = **ops.truediv**

Same as a / b .

xor = **ops.xor**

Same as $a \wedge b$.

1.3 Array operations

all = **ops.all**

amax = **ops.amax**

amin = **ops.amin**

any = **ops.any**

argmax = **ops.argmax**

argmin = **ops.argmin**

astype = **ops.astype**

cat = **ops.cat**

cholesky = **ops.cholesky**

Like `numpy.linalg.cholesky()` but uses `sqrt` for scalar matrices.

cholesky_inverse = **ops.cholesky_inverse**

Like `torch.cholesky_inverse()` but supports batching and gradients.

cholesky_solve = **ops.cholesky_solve**

clamp = **ops.clamp**

detach = **ops.detach**

diagonal = **ops.diagonal**

einsum = **ops.einsum**

expand = **ops.expand**

finfo = **ops.finfo**

flip = **ops.flip**

full_like = **ops.full_like**

```
isnan = ops.isnan
logaddexp = ops.logaddexp
logsumexp = ops.logsumexp
mean = ops.mean
new_arange = ops.new_arange
new_eye = ops.new_eye
new_full = ops.new_full
new_zeros = ops.new_zeros
permute = ops.permute
prod = ops.prod
qr = ops.qr
randn = ops.randn
sample = ops.sample
scatter = ops.scatter
scatter_add = ops.scatter_add
stack = ops.stack
std = ops.std
sum = ops.sum
transpose = ops.transpose
triangular_inv = ops.triangular_inv
triangular_solve = ops.triangular_solve
unsqueeze = ops.unsqueeze
var = ops.var
```

DOMAINS

class BintBases: `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 BintType**Bases: `ArrayType`**property size****class Dependent**(*fn*)Bases: `object`

Type hint for dependently type-decorated functions.

Examples:

```
Dependent[Real] # a constant known domain
Dependent[lambda x: Array[x.dtype, x.shape[1:]] # args are Domains
Dependent[lambda x, y: Bint[x.size + y.size]]
```

Parameters

fn (*callable*) – A lambda taking named arguments (in any order) which will be filled in with the domain of the similarly named functor argument to the decorated function. This lambda should compute a desired resulting domain given domains of arguments.

Domainalias of `type`**class Real**Bases: `object`**shape** = `()`

class RealsBases: `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 RealsType**Bases: `ArrayType`**dtype = 'real'****bint**(*size*)**find_domain**(*op*, **domains*)**find_domain**(*op*: `UnaryOp`, *domain*)**find_domain**(*op*: `AstypeOp`, *domain*)**find_domain**(*op*: `ExpOp`, *domain*)**find_domain**(*op*: `LogOp`, *domain*)**find_domain**(*op*: `ReductionOp`, *domain*)**find_domain**(*op*: `ReshapeOp`, *domain*)**find_domain**(*op*: `GetitemOp`, *lhs_domain*, *rhs_domain*)**find_domain**(*op*: `GetsliceOp`, *domain*)**find_domain**(*op*: `BinaryOp`, *lhs*, *rhs*)**find_domain**(*op*: `ComparisonOp`, *lhs*, *rhs*)**find_domain**(*op*: `FloordivOp`, *lhs*, *rhs*)**find_domain**(*op*: `ModOp`, *lhs*, *rhs*)**find_domain**(*op*: `MatmulOp`, *lhs*, *rhs*)**find_domain**(*op*: `AssociativeOp`, **domains*)**find_domain**(*op*: `WrappedTransformOp`, *domain*)**find_domain**(*op*: `LogAbsDetJacobianOp`, *domain*, *codomain*)**find_domain**(*op*: `StackOp`, *parts*)**find_domain**(*op*: `CatOp`, *parts*)**find_domain**(*op*: `EinsumOp`, *operands*)

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

reals(**args*)

INTERPRETATIONS

3.1 Interpreter

exception `PatternMissingError`

Bases: `NotImplementedError`

interpretation(*new*)

pop_interpretation()

push_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 *funson* or data structure holding *funsons*.) – An input, typically involving deferred *Funson*s.

Returns

A reinterpreted version of the input.

Raises

`ValueError`

3.2 Interpretations

class `CallableInterpretation`(*interpret*)

Bases: `Interpretation`

A simple callable interpretation.

Example usage:

```
@CallableInterpretation
def my_interpretation(cls, *args):
    return ...
```

Parameters

interpret (*callable*) – A function implementing interpretation.

`set_callable(interpret)`

Resets the callable `.interpret` attribute.

class `DispatchedInterpretation(name='dispatched')`

Bases: `Interpretation`

An interpretation based on pattern matching.

Example usage:

```
my_interpretation = DispatchedInterpretation("my_interpretation")

# Register a funsor pattern and rule.
@my_interpretation.register(...)
def my_impl(cls, *args):
    ...

# Use the new interpretation.
with my_interpretation:
    ...
```

class `Interpretation(name)`

Bases: `ContextDecorator`, `ABC`

Abstract base class for Funsor interpretations.

Instances may be used as context managers or decorators.

Parameters

name (*str*) – A name used for printing and debugging (required).

class `Memoize(base_interpretation, cache=None)`

Bases: `Interpretation`

Exploits cons-hashing to do implicit common subexpression elimination.

Parameters

- **base_interpretation** (`Interpretation`) – The interpretation to memoize.
- **cache** (*dict*) – An optional temporary cache where results will be memoized.

class `StatefulInterpretation(name='stateful')`

Bases: `Interpretation`

Base class for interpretations 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(interpretation_state, cls, *args):
    my_param = interpretation_state.my_param
    ...
```

(continues on next page)

(continued from previous page)

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

eager = eager/normalize/reflect

Eager exact naive interpretation wherever possible.

lazy = lazy/reflect

Performs substitutions eagerly, but construct lazy funsors for everything else.

memoize(cache=None)Context manager wrapping *Memoize* and yielding the cache dict.**moment_matching = moment_matching/eager/normalize/reflect**A moment matching interpretation of Reduce expressions. This falls back to *eager* in other cases.**normalize = normalize/reflect**

Normalize modulo associativity and commutativity, but do not evaluate any numerical operations.

sequential = sequential/eager/normalize/reflect

Eagerly execute ops with known implementations; additionally execute vectorized ops sequentially if no known vectorized implementation exists.

3.3 Monte Carlo

class MonteCarlo(*, rng_key=None, **sample_inputs)Bases: *StatefulInterpretation*A Monte Carlo interpretation of *Integrate* expressions. This falls back to the previous interpreter in other cases.**Parameters****rng_key** –

3.4 Preconditioning

class Precondition(aux_name='aux')Bases: *StatefulInterpretation*

Preconditioning interpretation for adjoint computations.

This interpretation is intended to be used once, followed by a call to *combine_subs()* as follows:

```
# Lazily build a factor graph.
with reflect:
    log_joint = Gaussian(...) + ... + Gaussian(...)
    log_Z = log_joint.reduce(ops.logaddexp)

# Run a backward sampling under the precondition interpretation.
with Precondition() as p:
    marginals = adjoint(
        ops.logaddexp, ops.add, log_Z, batch_vars=p.sample_vars
```

(continues on next page)

```

    )
    combine_subs = p.combine_subs()

    # Extract samples from Delta distributions.
    samples = {
        k: v(**combine_subs)
        for name, delta in marginals.items()
        for k, v in funsor.montecarlo.extract_samples(delta).items()
    }

```

See `forward_filter_backward_precondition()` for complete usage.

Parameters

aux_name (*str*) – Name of the auxiliary variable containing white noise.

combine_subs()

Method to create a combining substitution after preconditioning is complete. The returned substitution replaces per-factor auxiliary variables with slices into a single combined auxiliary variable.

Returns

A substitution indexing each factor-wise auxiliary variable into a single global auxiliary variable.

Return type

dict

3.5 Approximations

argmax_approximate = argmax_approximate

Point-approximate at the argmax of the provided guide.

compute_argmax(*model*, *approx_vars*)

compute_argmax(*model*: Tensor, *approx_vars*)

compute_argmax(*model*: Gaussian, *approx_vars*)

compute_argmax(*model*: Contraction[Union[LogaddexpOp, NullOp], AddOp, frozenset, Tuple[Union[Tensor, Number], Gaussian]], *approx_vars*)

compute_argmax(*model*: Contraction[NullOp, AddOp, frozenset, tuple], *approx_vars*)

Computes argmax of a funsor.

Parameters

- **model** (Funsor) – A function of the approximated vars.
- **approx_vars** (frozenset) – A frozenset of *Variable*s to maximize.

Returns

A dict mapping name (str) to point estimate (Funsor), for each variable name in *approx_vars*.

Return type

str

laplace_approximate = laplace_approximate

Gaussian approximate using the value and Hessian of the model, evaluated at the mode of the guide.

`mean_approximate = mean_approximate`

Point-approximate at the mean of the provided guide.

3.6 Evidence lower bound

`class Elbo(guide, approx_vars)`

Bases: *StatefulInterpretation*

Given an approximating guide funsor, approximates:

```
model.reduce(ops.logaddexp, approx_vars)
```

by the lower bound:

```
Integrate(guide, model - guide, approx_vars)
```

Parameters

- **guide** (*Funsor*) – A guide or proposal funsor.
- **approx_vars** (*frozenset*) – The variables being integrated.

4.1 Basic Funsors

class `Approximate(*args, **kwargs)`

Bases: `Funsor`

Interpretation-specific approximation wrt a set of variables.

The default eager interpretation should be exact. The user-facing interface is the `Funsor.approximate()` method.

Parameters

- **op** (`AssociativeOp`) – An associative operator.
- **model** (`Funsor`) – An exact funsor depending on `approx_vars`.
- **guide** (`Funsor`) – A proposal funsor guiding optional approximation.
- **approx_vars** (`frozenset`) – A set of variables over which to approximate.

class `Binary(*args, **kwargs)`

Bases: `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 `Cat(name, parts, part_name=None)`

Bases: `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 Funsor(*args, **kwargs)

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.

property dtype

property shape

input_vars()

quote()

pretty(*args, **kwargs)

item()

property requires_grad

reduce(op, reduced_vars=None)

Reduce along all or a subset of inputs.

Parameters

- **op** (*AssociativeOp* or *ReductionOp*) – 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.

approximate(op, guide, approx_vars=None)

Approximate wrt and all or a subset of inputs.

Parameters

- **op** (*AssociativeOp*) – A reduction operation.
- **guide** (*Funsor*) – A guide funsor (e.g. a proposal distribution).
- **approx_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 *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.

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

`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()`

`atanh()`

`sqrt()`

`exp()`

`log()`

`log1p()`

`sigmoid()`

`tanh()`

reshape(*shape*)

all(*axis=None, keepdims=False*)

any(*axis=None, keepdims=False*)

argmax(*axis=None, keepdims=False*)

argmin(*axis=None, keepdims=False*)

max(*axis=None, keepdims=False*)

min(*axis=None, keepdims=False*)

sum(*axis=None, keepdims=False*)

prod(*axis=None, keepdims=False*)

logsumexp(*axis=None, keepdims=False*)

mean(*axis=None, keepdims=False*)

std(*axis=None, ddof=0, keepdims=False*)

var(*axis=None, ddof=0, keepdims=False*)

class Independent(*args, **kwargs)

Bases: *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.add, '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*.

eager_subs(*subs*)

mean()

variance()

entropy()

class Lambda(*args, **kwargs)

Bases: *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 Number(data, dtype=None)

Bases: *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 Reduce(*args, **kwargs)

Bases: *Funsor*

Lazy reduction over multiple variables.

The user-facing interface is the *Funsor.reduce()* method.

Parameters

- **op** (*AssociativeOp*) – An associative operator.
- **arg** (*funsor*) – An argument to be reduced.
- **reduced_vars** (*frozenset*) – A set of variables over which to reduce.

class Scatter(*args, **kwargs)

Bases: *Funsor*

Transpose of structurally linear *Subs*, followed by *Reduce*.

For injective scatter operations this should satisfy the equation:

```
if destin = Scatter(op, subs, source, frozenset())
then source = Subs(destin, subs)
```

The `reduced_vars` is merely for computational efficiency, and could always be split out into a separate `.reduce()`. For example in the following equation, the left hand side uses much less memory than the right hand side:

```
Scatter(op, subs, source, reduced_vars) ==
Scatter(op, subs, source, frozenset()).reduce(op, reduced_vars)
```

Warning: This is currently implemented only for injective scatter operations. In particular, this does not allow accumulation behavior like scatter-add.

Note: `Scatter(ops.add, ...)` is the funsor analog of `numpy.add.at()` or `torch.index_put()` or `jax.lax.scatter_add()`. For injective substitutions, `Scatter(ops.add, ...)` is roughly equivalent to the tensor operation:

```
result = zeros(...) # since zero is the additive unit
result[subs] = source
```

Parameters

- **op** (*AssociativeOp*) – An op. The unit of this op will be used as default value.
- **subs** (*tuple*) – A substitution.
- **source** (*Funsor*) – A source for data to be scattered from.
- **reduced_vars** (*frozenset*) – A set of variables over which to reduce.

`eager_subs(subs)`

`class Stack(*args, **kwargs)`

Bases: *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 Slice(name, *args, **kwargs)`

Bases: *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 Subs(*arg, subs*)

Bases: *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()*.

class Unary(**args, **kwargs*)

Bases: *Funsor*

Lazy unary operation.

Parameters

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

class Variable(**args, **kwargs*)

Bases: *Funsor*

Funsor representing a single free variable.

Parameters

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

eager_subs(*subs*)

of_shape(**shape*)

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

to_data(*x: Funsor, name_to_dim=None*)

to_data(*x: Number, name_to_dim=None*)

to_data(*x: Tensor, name_to_dim=None*)

to_data(*funsor_dist: Distribution, name_to_dim=None*)

to_data(*funsor_dist: Independent[Union[Independent, Distribution], str, str, str], name_to_dim=None*)

to_data(*funsor_dist: Gaussian, name_to_dim=None*)

to_data(*funsor_dist: Contraction[Union[LogaddexpOp, NullOp], AddOp, frozenset, Tuple[Union[Tensor, Number], Gaussian]], name_to_dim=None*)

to_data(*funsor_dist: Multinomial, name_to_dim=None*)

to_data(*funsor_dist: Delta, name_to_dim=None*)

to_data(*expr: Unary[TransformOp, Union[Unary, Variable]], name_to_dim=None*)

to_data(*x: Constant, name_to_dim=None*)

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.

```

to_funsor(x, output=None, dim_to_name=None, **kwargs)
to_funsor(x: Funsor, output=None, dim_to_name=None)
to_funsor(name: str, output=None)
to_funsor(x: Number, output=None, dim_to_name=None)
to_funsor(s: slice, output=None, dim_to_name=None)
to_funsor(args: tuple, output=None, dim_to_name=None)
to_funsor(x: generic, output=None, dim_to_name=None)
to_funsor(x: ndarray, output=None, dim_to_name=None)
to_funsor(backend_dist: Beta, output=None, dim_to_name=None)
to_funsor(backend_dist: Cauchy, output=None, dim_to_name=None)
to_funsor(backend_dist: Chi2, output=None, dim_to_name=None)
to_funsor(backend_dist: _PyroWrapper_BernoulliProbs, output=None, dim_to_name=None)
to_funsor(backend_dist: _PyroWrapper_BernoulliLogits, output=None, dim_to_name=None)
to_funsor(backend_dist: Binomial, output=None, dim_to_name=None)
to_funsor(backend_dist: Categorical, output=None, dim_to_name=None)
to_funsor(backend_dist: _PyroWrapper_CategoricalLogits, output=None, dim_to_name=None)
to_funsor(pyro_dist: Delta, output=None, dim_to_name=None)
to_funsor(backend_dist: Dirichlet, output=None, dim_to_name=None)
to_funsor(backend_dist: DirichletMultinomial, output=None, dim_to_name=None)
to_funsor(backend_dist: Exponential, output=None, dim_to_name=None)
to_funsor(backend_dist: Gamma, output=None, dim_to_name=None)
to_funsor(backend_dist: GammaPoisson, output=None, dim_to_name=None)
to_funsor(backend_dist: Geometric, output=None, dim_to_name=None)
to_funsor(backend_dist: Gumbel, output=None, dim_to_name=None)
to_funsor(backend_dist: HalfCauchy, output=None, dim_to_name=None)
to_funsor(backend_dist: HalfNormal, output=None, dim_to_name=None)
to_funsor(backend_dist: Laplace, output=None, dim_to_name=None)
to_funsor(backend_dist: Logistic, output=None, dim_to_name=None)
to_funsor(backend_dist: LowRankMultivariateNormal, output=None, dim_to_name=None)
to_funsor(backend_dist: Multinomial, output=None, dim_to_name=None)
to_funsor(backend_dist: MultivariateNormal, output=None, dim_to_name=None)
to_funsor(backend_dist: NonreparameterizedBeta, output=None, dim_to_name=None)
to_funsor(backend_dist: NonreparameterizedDirichlet, output=None, dim_to_name=None)
to_funsor(backend_dist: NonreparameterizedGamma, output=None, dim_to_name=None)
to_funsor(backend_dist: NonreparameterizedNormal, output=None, dim_to_name=None)
to_funsor(backend_dist: Normal, output=None, dim_to_name=None)
to_funsor(backend_dist: Pareto, output=None, dim_to_name=None)
to_funsor(backend_dist: Poisson, output=None, dim_to_name=None)

```

```

to_funsor(backend_dist: StudentT, output=None, dim_to_name=None)
to_funsor(backend_dist: Uniform, output=None, dim_to_name=None)
to_funsor(backend_dist: VonMises, output=None, dim_to_name=None)
to_funsor(backend_dist: ContinuousBernoulli, output=None, dim_to_name=None)
to_funsor(backend_dist: FisherSnedecor, output=None, dim_to_name=None)
to_funsor(backend_dist: NegativeBinomial, output=None, dim_to_name=None)
to_funsor(backend_dist: OneHotCategorical, output=None, dim_to_name=None)
to_funsor(backend_dist: RelaxedBernoulli, output=None, dim_to_name=None)
to_funsor(backend_dist: Weibull, output=None, dim_to_name=None)
to_funsor(tfn: Transform, output=None, dim_to_name=None, real_inputs=None)
to_funsor(tfn: ExpTransform, output=None, dim_to_name=None, real_inputs=None)
to_funsor(tfn: TanhTransform, output=None, dim_to_name=None, real_inputs=None)
to_funsor(tfn: SigmoidTransform, output=None, dim_to_name=None, real_inputs=None)
to_funsor(tfn: _InverseTransform, output=None, dim_to_name=None, real_inputs=None)
to_funsor(tfn: ComposeTransform, output=None, dim_to_name=None, real_inputs=None)
to_funsor(backend_dist: ExpandedDistribution, output=None, dim_to_name=None)
to_funsor(backend_dist: Independent, output=None, dim_to_name=None)
to_funsor(backend_dist: MaskedDistribution, output=None, dim_to_name=None)
to_funsor(backend_dist: TransformedDistribution, output=None, dim_to_name=None)
to_funsor(pyro_dist: Bernoulli, output=None, dim_to_name=None)
to_funsor(x: ProvenanceTensor, output=None, dim_to_name=None)
to_funsor(x: Tensor, output=None, dim_to_name=None)
to_funsor(pyro_dist: FunsorDistribution, output=None, dim_to_name=None)

```

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

4.2 Delta

```
class Delta(*args)
```

Bases: *Funsor*

Normalized delta distribution binding multiple variables.

There are three syntaxes supported for constructing Deltas:

```
Delta(((name1, (point1, log_density1)),
      (name2, (point2, log_density2)),
      (name3, (point3, log_density3))))
```

or for a single name:

```
Delta(name, point, log_density)
```

or for default `log_density == 0`:

```
Delta(name, point)
```

Parameters

terms (*tuple*) – A tuple of tuples of the form (name, (point, log_density)).

align(*names*)

eager_subs(*subs*)

eager_reduce(*op, reduced_vars*)

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

4.3 Tensor

Einsum(*equation, *operands*)

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.

class Function(*args, **kwargs)

Bases: *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.

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

Bases: *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, {"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** (*dict*) – 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()`

`property requires_grad`

`align(names)`

`eager_subs(subs)`

`eager_unary(op)`

`eager_reduce(op, reduced_vars)`

`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

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]:
```

(continues on next page)

(continued from previous page)

```

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.

ignore_jit_warnings()

tensor_dot(x, y, dims)

Wrapper around `torch.tensor_dot()` or `np.tensor_dot()` 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 Gaussian

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()`

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 `Gaussian`(*white_vec=None, prec_sqrt=None, inputs=None, *, mean=None, info_vec=None, precision=None, scale_tril=None, covariance=None*)

Bases: `Funsor`

Funsor representing a batched Gaussian log-density function.

Gaussians are the internal representation for joint and conditional multivariate normal distributions and multivariate normal likelihoods. Mathematically, a Gaussian represents the quadratic log density function:

$$\begin{aligned} f(\mathbf{x}) &= -0.5 * || \mathbf{x} @ \text{prec_sqrt} - \text{white_vec} ||^2 \\ &= -0.5 * < \mathbf{x} @ \text{prec_sqrt} - \text{white_vec} | \mathbf{x} @ \text{prec_sqrt} - \text{white_vec} > \\ &= -0.5 * < \mathbf{x} | \text{prec_sqrt} @ \text{prec_sqrt.T} | \mathbf{x} > \\ &\quad + < \mathbf{x} | \text{prec_sqrt} | \text{white_vec} > - 0.5 ||\text{white_vec}||^2 \end{aligned}$$

Internally Gaussians use a square root information filter (SRIF) representation consisting of a square root of the precision matrix `prec_sqrt` and a vector in the whitened space `white_vec`. This representation allows space-efficient construction of Gaussians with incomplete information, i.e. with zero eigenvalues in the precision matrix. These incomplete log densities arise when making low-dimensional observations of higher-dimensional hidden state. Sampling and marginalization are supported only for full-rank Gaussians or full-rank subsets of Gaussians. See the `rank()` and `is_full_rank()` properties.

Note: `Gaussian`s are not normalized probability distributions, rather they are canonicalized to evaluate to zero log density at their maximum: $f(\text{prec_sqrt} \setminus \text{white_vec}) = 0$. Not only are Gaussians non-normalized,

but they may be rank deficient and non-normalizable, in which case sampling and marginalization are supported only on full-rank subsets of variables.

Parameters

- **white_vec** (*torch.Tensor*) – An batched white noise vector, where `white_vec = prec_sqrt.T @ mean`. Alternatively you can specify one of the kwargs `mean` or `info_vec`, which will be converted to `white_vec`.
- **prec_sqrt** (*torch.Tensor*) – A batched square root of the positive semidefinite precision matrix. This need not be square, and typically has shape `prec_sqrt.shape == white_vec.shape[:-1] + (dim, rank)`, where `dim` is the total flattened size of real inputs and `rank = white_vec.shape[-1]`. Alternatively you can specify one of the kwargs `precision`, `covariance`, or `scale_tril`, which will be converted to `prec_sqrt`.
- **inputs** (*OrderedDict*) – Mapping from name to *Domain*.

`compression_threshold = 2`

classmethod `set_compression_threshold(threshold: float)`

Context manager to set rank compression threshold.

To save space Gaussians compress wide `prec_sqrt` matrices down to square. However compression uses a QR decomposition which can be expensive and which has unstable gradients when the resulting precision matrix is rank deficient. To balance space and time costs and numerical stability, compression is trigger only on `prec_sqrt` matrices whose width to height ratio is greater than `threshold`.

Parameters

threshold (*float*) – Defaults to 2. To optimize for space and deterministic computations, set `threshold = 1`. To optimize for fewest QR decompositions and numerical stability, set `threshold = math.inf`.

property `rank`

property `is_full_rank`

`log_normalizer()`

`align(names)`

`eager_subs(subs)`

`eager_reduce(op, reduced_vars)`

`align_gaussian(new_inputs, old, expand=False)`

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

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(*args, **kwargs)`

Bases: `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

align(*names*)

GaussianMixture

alias of `Contraction`

children_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`

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.

4.8 Constant

class `ConstantMeta`(*name, bases, dct*)

Bases: `FunsorMeta`

Wrapper to convert `const_inputs` to a tuple.

class `Constant`(*const_inputs, arg*)

Bases: `Funsor`

Funsor that is constant wrt `const_inputs`.

`Constant` can be used for provenance tracking.

Examples:

```
a = Constant(OrderedDict(x=Real, y= Bint[3]), Number(0))
a(y=1) # returns Constant(OrderedDict(x=Real), Number(0))
a(x=2, y=1) # returns Number(0)

d = Tensor(torch.tensor([1, 2, 3]))["y"]
a + d # returns Constant(OrderedDict(x=Real), d)

c = Constant(OrderedDict(x=Bint[3]), Number(1))
c.reduce(ops.add, "x") # returns Number(3)
```

Parameters

- **const_inputs** (*dict*) – A mapping from input name (*str*) to datatype (`funsor.domain.Domain`).
- **arg** (*funsor*) – A funsor that is constant wrt to `const_inputs`.

`eager_subs`(*subs*)

`eager_reduce`(*op, reduced_vars*)

`align`(*names*)

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

eager_reduce_add(*op, arg, reduced_vars*)

eager_binary_constant_constant(*op, lhs, rhs*)

eager_binary_constant_tensor(*op, lhs, rhs*)

eager_binary_tensor_constant(*op, lhs, rhs*)

eager_unary(*op, arg*)

OPTIMIZER

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

`unfold_contraction_variadic(r, b, v, *ts)`

`optimize_contraction_variadic(r, b, v, *ts)`

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

`optimize_contract_finitary_functor(red_op, bin_op, reduced_vars, terms)`

`apply_optimizer(x)`

ADJOINT ALGORITHMS

class AdjointTape

Bases: *Interpretation*

interpret(cls, *args)

adjoint(sum_op, bin_op, root, targets=None, *, batch_vars={})

forward_backward(sum_op, bin_op, expr, *, batch_vars=frozenset({}))

adjoint(sum_op, bin_op, expr)

adjoint_binary(adj_sum_op, adj_prod_op, out_adj, op, lhs, rhs)

adjoint_reduce(adj_sum_op, adj_prod_op, out_adj, op, arg, reduced_vars)

adjoint_contract_unary(adj_sum_op, adj_prod_op, out_adj, sum_op, prod_op, reduced_vars, arg)

adjoint_contract_generic(adj_sum_op, adj_prod_op, out_adj, sum_op, prod_op, reduced_vars, terms)

adjoint_contract(adj_sum_op, adj_prod_op, out_adj, sum_op, prod_op, reduced_vars, lhs, rhs)

adjoint_cat(adj_sum_op, adj_prod_op, out_adj, name, parts, part_name)

adjoint_subs(adj_sum_op, adj_prod_op, out_adj, arg, subs)

adjoint_scatter(adj_sum_op, adj_prod_op, out_adj, op, subs, source, reduced_vars)

SUM-PRODUCT ALGORITHMS

partial_unroll(*factors*, *eliminate=frozenset({})*, *plate_to_step={}*)

Performs partial unrolling of plated factor graphs to standard factor graphs. Only plates with `history={0, 1}` are supported.

For plates (`history=0`) unrolling operation appends `_i` suffix to variable names for index `i` in the plate (e.g., `"x->"x_0`" for `i=0`). For markov dimensions (`history=1`) unrolling operation renames the suffixes `var_prev` to `var_i` and `var_curr` to `var_{i+1}` for index `i` (e.g., `"x_prev->"x_0`" and `"x_curr->"x_1`" for `i=0`). Markov vars are assumed to have names that follow `var_suffix` formatting and specifically `var_0` for the initial factor (e.g., (`"x_0"`, `"x_prev"`, `"x_curr"`) for `history=1`).

Parameters

- **factors** (*tuple or list*) – A collection of functors.
- **eliminate** (*frozenset*) – A set of free variables to unroll, including both sum variables and product variable.
- **plate_to_step** (*dict*) – A dict mapping markov dimensions to step collections that contain ordered sequences of Markov variable names (e.g., `{"time": frozenset({"x_0", "x_prev", "x_curr"})}`). Plates are passed with an empty step.

Returns

a list of partially unrolled Functors, a frozenset of partially unrolled variable names, and a frozenset of remaining plates.

partial_sum_product(*sum_op*, *prod_op*, *factors*, *eliminate=frozenset({})*, *plates=frozenset({})*, *pedantic=False*, *pow_op=None*, *plate_to_scale=None*)

Performs partial sum-product contraction of a collection of factors.

Returns

a list of partially contracted Functors.

Return type

list

dynamic_partial_sum_product(*sum_op*, *prod_op*, *factors*, *eliminate=frozenset({})*, *plate_to_step={}*)

Generalization of the tensor variable elimination algorithm of `functor.sum_product.partial_sum_product()` to handle higher-order markov dimensions in addition to plate dimensions. Markov dimensions in transition factors are eliminated efficiently using the parallel-scan algorithm in `functor.sum_product.sarkka_bilmes_product()`. The resulting factors are then combined with the initial factors and final states are eliminated. Therefore, when Markov dimension is eliminated `factors` has to contain initial factors and transition factors.

Parameters

- **sum_op** (*AssociativeOp*) – A semiring sum operation.

- **prod_op** (*AssociativeOp*) – A semiring product operation.
- **factors** (*tuple or list*) – A collection of funsors.
- **eliminate** (*frozenset*) – A set of free variables to eliminate, including both sum variables and product variable.
- **plate_to_step** (*dict*) – A dict mapping markov dimensions to `step` collections that contain ordered sequences of Markov variable names (e.g., `{"time": frozenset({"x_0", "x_prev", "x_curr"})}`). Plates are passed with an empty `step`.

Returns

a list of partially contracted Funsors.

Return type

list

modified_partial_sum_product(*sum_op, prod_op, factors, eliminate=frozenset({}), plate_to_step={}*)

Generalization of the tensor variable elimination algorithm of `funsor.sum_product.partial_sum_product()` to handle markov dimensions in addition to plate dimensions. Markov dimensions in transition factors are eliminated efficiently using the parallel-scan algorithm in `funsor.sum_product.sequential_sum_product()`. The resulting factors are then combined with the initial factors and final states are eliminated. Therefore, when Markov dimension is eliminated `factors` has to contain a pairs of initial factors and transition factors.

Parameters

- **sum_op** (*AssociativeOp*) – A semiring sum operation.
- **prod_op** (*AssociativeOp*) – A semiring product operation.
- **factors** (*tuple or list*) – A collection of funsors.
- **eliminate** (*frozenset*) – A set of free variables to eliminate, including both sum variables and product variable.
- **plate_to_step** (*dict*) – A dict mapping markov dimensions to `step` collections that contain ordered sequences of Markov variable names (e.g., `{"time": frozenset({"x_0", "x_prev", "x_curr"})}`). Plates are passed with an empty `step`.

Returns

a list of partially contracted Funsors.

Return type

list

sum_product(*sum_op, prod_op, factors, eliminate=frozenset({}), plates=frozenset({}), pedantic=False, pow_op=None, plate_to_scale=None*)

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: *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=None*)

Bases: *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

`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 Ordered-Dict mapping input name to a (`coefficient`, `eqn`) pair in einsum form.

Return type

`tuple`

`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`

FUNSOR FACTORY

class Fresh(*fn*)

Bases: `object`

Type hint for `make_funsor()` decorated functions. This provides hints for fresh variables (names) and the return type.

Examples:

```
Fresh[Real] # a constant known domain
Fresh[lambda x: Array[x.dtype, x.shape[1:]] # args are Domains
Fresh[lambda x, y: Bint[x.size + y.size]]
```

Parameters

fn (*callable*) – A lambda taking named arguments (in any order) which will be filled in with the domain of the similarly named funsor argument to the decorated function. This lambda should compute a desired resulting domain given domains of arguments.

class Bound

Bases: `object`

Type hint for `make_funsor()` decorated functions. This provides hints for bound variables (names).

class Has(*bound*)

Bases: `object`

Type hint for `make_funsor()` decorated functions.

This hint asserts that a set of *Bound* variables always appear in the `.inputs` of the annotated argument.

For example, we could write a named `matmul` function that asserts that both arguments always contain the reduced input, and cannot be constant with respect to that input:

```
@make_funsor
def MatMul(
    x: Has[{"i"}],
    y: Has[{"i"}],
    i: Bound,
) -> Fresh[lambda x: x]:
    return (x * y).reduce(ops.add, i)
```

Here the string "i" in the annotations for `x` and `y` refer to the argument `i` of our `MatMul` function, which is known to be *Bound* (i.e it does not appear in the `.inputs` of evaluating `Matmul(x, y, "i")`).

Warning: This annotation is experimental and may be removed in the future.

Note that because Funsor is inherently extensional, violating a *Has* constraint only raises a `SyntaxWarning` rather than a full `TypeError` and even then only under the `reflect()` interpretation.

As such, *Has* annotations should be used sparingly, reserved for cases where the programmer has complete control over the inputs to a function and knows that an argument will always depend on a bound variable, e.g. when writing one-off Funsor terms to describe custom layers in a neural network.

Parameters

bound (*set*) – A set of strings of names of *Bound* arguments of a `make_funsor()` -decorated function.

`make_funsor(fn)`

Decorator to dynamically create a subclass of *Funsor*, together with a single default eager pattern.

This infers inputs, outputs, fresh, and bound variables from type hints follow the following convention:

- Funsor inputs are typed *Funsor*.
- Bound variable inputs (names) are typed *Bound*.
- Fresh variable inputs (names) are typed *Fresh* together with lambda to compute the dependent domain.
- Ground value inputs (e.g. Python ints) are typed `Value` together with their actual data type, e.g. `Value[int]`.
- The return value is typed *Fresh* together with a lambda to compute the dependent return domain.

For example to unflatten a single coordinate into a pair of coordinates we could define:

```
@make_funsor
def Unflatten(
    x: Funsor,
    i: Bound,
    i_over_2: Fresh[lambda i: Bint[i.size // 2]],
    i_mod_2: Fresh[lambda: Bint[2]],
) -> Fresh[lambda x: x]:
    assert i.output.size % 2 == 0
    return x(**{i.name: i_over_2 * Number(2, 3) + i_mod_2})
```

Parameters

fn (*callable*) – A type annotated function of Funsors.

Return type

subclass of *Funsor*

TESTING UTILITIES

xfail_if_not_implemented(*msg='Not implemented', *, match=None*)

xfail_if_not_found(*msg='Not implemented'*)

requires_backend(**backends, reason=None*)

excludes_backend(**backends, reason=None*)

class ActualExpected(*actual, expected*)

Bases: `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_functor(*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 functors are equivalent up to permutation of inputs.

rand(**args*)

randint(*low, high, size*)

randn(**args*)

random_scale_tril(**args*)

zeros(**args*)

ones(**args*)

empty(**args*)

random_tensor(*inputs, output=Real*)

Creates a random *functor.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*)

iter_subsets(*iterable*, *, *min_size=None*, *max_size=None*)

class DesugarGetitem

Bases: `object`

Helper to desugar `.__getitem__()` syntax.

Example:

```
>>> desugar_getitem[1:3, ..., None]
(slice(1, 3), Ellipsis, None)
```

TYPING UTILITIES

class `GenericTypeMeta`(*name, bases, dct*)

Bases: `type`

Metaclass to support subtyping with parameters for pattern matching, e.g. `Number[int, int]`.

class `Variadic`

Bases: `object`

A typing-compatible drop-in replacement for `Variadic`.

deep_isinstance(*obj, cls*)

Enhanced version of `isinstance()` that can handle basic structured `typing` types, including `Funsor` terms and other `GenericTypeMeta` instances, `Union`, `Tuple`, and `FrozenSet`.

Does not support `TypeVar`, arbitrary `Generic`, forward references, or mutable generic collection types like `List`. Will attempt to fall back to `isinstance()` when it encounters an unsupported type in `obj` or `cls`.

Usage:

```
x = (1, ("a", "b"))
assert deep_isinstance(x, typing.Tuple[int, tuple])
assert deep_isinstance(x, typing.Tuple[typing.Any, typing.Tuple[str, ...]])
```

Parameters

- **obj** – An object that may be an instance of `cls`.
- **cls** – A class that may be a parent class of `obj`.

deep_issubclass(*subcls, cls*)

Enhanced version of `issubclass()` that can handle structured types, including `Funsor` terms, `Tuple`, and `FrozenSet`.

Does not support more advanced `typing` features such as `TypeVar`, arbitrary `Generic` subtypes, forward references, or mutable collection types like `List`. Will attempt to fall back to `issubclass()` when it encounters a type in `subcls` or `cls` that it does not understand.

Usage:

```
class A: pass
class B(A): pass

assert deep_issubclass(typing.Tuple[int, B], typing.Tuple[int, A])
assert not deep_issubclass(typing.Tuple[int, A], typing.Tuple[int, B])
```

(continues on next page)

(continued from previous page)

```
assert deep_issubclass(typing.Tuple[A, A], typing.Tuple[A, ...])
assert not deep_issubclass(typing.Tuple[B], typing.Tuple[A, ...])
```

Parameters

- **subcls** – A class that may be a subclass of **cls**.
- **cls** – A class that may be a parent class of **subcls**.

deep_type(*obj*)**deep_type**(*obj: tuple*)**deep_type**(*obj: frozenset*)

An enhanced version of `type()` that reconstructs structured `typing`` types for a limited set of immutable data structures, notably `tuple` and `frozenset`. Mostly intended for internal use in Funsor interpretation pattern-matching.

Example:

```
assert deep_type((1, ("a",))) is typing.Tuple[int, typing.Tuple[str]]
assert deep_type(frozenset(["a"])) is typing.FrozenSet[str]
```

get_args(*tp*)**get_origin**(*tp*)**get_type_hints**(*obj, globalns=None, localns=None*)

Return type hints for an object.

This is often the same as `obj.__annotations__`, but it handles forward references encoded as string literals, and if necessary adds `Optional[t]` if a default value equal to `None` is set.

The argument may be a module, class, method, or function. The annotations are returned as a dictionary. For classes, annotations include also inherited members.

`TypeError` is raised if the argument is not of a type that can contain annotations, and an empty dictionary is returned if no annotations are present.

BEWARE – the behavior of `globalns` and `localns` is counterintuitive (unless you are familiar with how `eval()` and `exec()` work). The search order is `locals` first, then `globals`.

- If no dict arguments are passed, an attempt is made to use the `globals` from `obj` (or the respective module's `globals` for classes), and these are also used as the `locals`. If the object does not appear to have `globals`, an empty dictionary is used.
- If one dict argument is passed, it is used for both `globals` and `locals`.
- If two dict arguments are passed, they specify `globals` and `locals`, respectively.

register_subclasscheck(*cls*)

Decorator for registering a custom `__subclasscheck__` method for `cls` which is only ever invoked in `deep_issubclass()`.

This is primarily intended for working with the `typing` library at runtime. Prefer overriding `__subclasscheck__` in the usual way with a metaclass where possible.

class `typing_wrap`(*tp*)

Bases: `object`

Utility callable for overriding the runtime behavior of `typing` objects.

RECIPES USING FUNSOR

This module provides a number of high-level algorithms using Funsor.

forward_filter_backward_rsample(*factors*: *Dict*[*str*, *Funsor*], *eliminate*: *FrozenSet*[*str*], *plates*: *FrozenSet*[*str*], *sample_inputs*: *Dict*[*str*, *type*] = {}, *rng_key*=None)

A forward-filter backward-batched-reparametrized-sample algorithm for use in variational inference. The motivating use case is performing Gaussian tensor variable elimination over structured variational posteriors.

Parameters

- **factors** (*dict*) – A dictionary mapping sample site name to a Funsor factor created at that sample site.
- **frozenset** – A set of names of latent variables to marginalize and plates to aggregate.
- **plates** – A set of names of plates to aggregate.
- **sample_inputs** (*dict*) – An optional dict of enclosing sample indices over which samples will be drawn in batch.
- **rng_key** – A random number key for the JAX backend.

Returns

A pair *samples*:*Dict*[*str*, *Tensor*], *log_prob*: *Tensor* of samples and log density evaluated at each of those samples. If *sample_inputs* is nonempty, both outputs will be batched.

Return type

tuple

forward_filter_backward_precondition(*factors*: *Dict*[*str*, *Funsor*], *eliminate*: *FrozenSet*[*str*], *plates*: *FrozenSet*[*str*], *aux_name*: *str* = 'aux')

A forward-filter backward-precondition algorithm for use in variational inference or preconditioning in Hamiltonian Monte Carlo. The motivating use case is performing Gaussian tensor variable elimination over structured variational posteriors, and optionally using the learned posterior to determine momentum in HMC.

Parameters

- **factors** (*dict*) – A dictionary mapping sample site name to a Funsor factor created at that sample site.
- **frozenset** – A set of names of latent variables to marginalize and plates to aggregate.
- **plates** – A set of names of plates to aggregate.
- **aux_name** (*str*) – Name of the auxiliary variable containing white noise.

Returns

A pair `samples:Dict[str, Tensor]`, `log_prob: Tensor` of samples and log density evaluated at each of those samples. Both outputs depend on a vector named by `aux_name`, e.g. `aux: Reals[d]` where `d` is the total number of elements in eliminated variables.

Return type

`tuple`

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).

13.1 FunctorDistribution Base Class

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

Bases: `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 = {}
```

```
property 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)
```

13.2 Hidden Markov Models

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

Bases: *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.

property 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: *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 `Independent of 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: *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: *FunsorDistribution*

Switching Linear Dynamical System represented as a Hidden Markov Model.

This corresponds to the generative model:

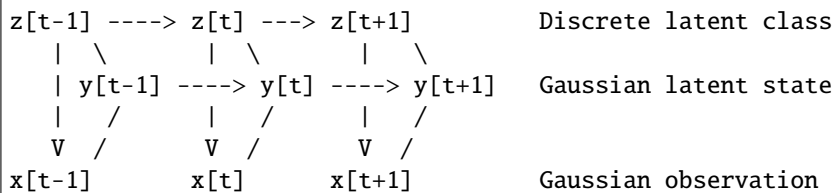
```
z = Categorical(logits=initial_logits).sample()
y = initial_mvn[z].sample()
```

(continues on next page)

(continued from previous page)

```
x = []
for t in range(num_steps):
    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:



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

13.3 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.distribution.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.

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, **kwargs)`

Bases: *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*)

property `has_enumerate_support`

classmethod `eager_log_prob(*params)`

enumerate_support(*expand=False*)

entropy()

mean()

variance()

class `Beta(*args, **kwargs)`

Bases: *Distribution*

dist_class

alias of `Beta`

class `Cauchy(*args, **kwargs)`

Bases: *Distribution*

dist_class

alias of `Cauchy`

class `Chi2(*args, **kwargs)`

Bases: *Distribution*

dist_class

alias of `Chi2`

```
class BernoulliProbs(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of _PyroWrapper_BernoulliProbs
class BernoulliLogits(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of _PyroWrapper_BernoulliLogits
class Binomial(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Binomial
class Categorical(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Categorical
class CategoricalLogits(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of _PyroWrapper_CategoricalLogits
class Delta(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Delta
class Dirichlet(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Dirichlet
class DirichletMultinomial(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of DirichletMultinomial
class Exponential(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Exponential
class Gamma(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Gamma
```

```
class GammaPoisson(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of GammaPoisson
class Geometric(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Geometric
class Gumbel(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Gumbel
class HalfCauchy(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of HalfCauchy
class HalfNormal(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of HalfNormal
class Laplace(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Laplace
class Logistic(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Logistic
class LowRankMultivariateNormal(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of LowRankMultivariateNormal
class Multinomial(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Multinomial
class MultivariateNormal(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of MultivariateNormal
```

```
class NonreparameterizedBeta(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of NonreparameterizedBeta
class NonreparameterizedDirichlet(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of NonreparameterizedDirichlet
class NonreparameterizedGamma(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of NonreparameterizedGamma
class NonreparameterizedNormal(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of NonreparameterizedNormal
class Normal(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Normal
class Pareto(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Pareto
class Poisson(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Poisson
class StudentT(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of StudentT
class Uniform(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of Uniform
class VonMises(*args, **kwargs)
    Bases: Distribution
    dist_class
        alias of 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.

15.1 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: Messenger
    postprocess_message(msg)
    get_trace(*args, **kwargs)

class replay(fn, guide_trace)
    Bases: Messenger
    process_message(msg)

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

process_message(*msg*)

class seed(*fn=None, rng_seed=None*)

Bases: *Messenger*

class CondIndepStackFrame(*name, size, dim*)

Bases: *tuple*

property dim

Alias for field number 2

property name

Alias for field number 0

property size

Alias for field number 1

class PlateMessenger(*fn, name, size, dim*)

Bases: *Messenger*

process_message(*msg*)

tensor_to_funsor(*value, cond_indep_stack, output*)

class log_joint(*fn=None*)

Bases: *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: *PyroOptim*

TorchOptimizer

alias of *Adam*

class ClippedAdam(*optim_args*)

Bases: *PyroOptim*

TorchOptimizer

alias of *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: [ELBO](#)

class TraceMeanField_ELBO(***kwargs*)

Bases: [ELBO](#)

class TraceEnum_ELBO(***kwargs*)

Bases: [ELBO](#)

class Jit(*fn, **kwargs*)

Bases: [object](#)

class Jit_ELBO(*elbo, **kwargs*)

Bases: [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.

COMPILER & TRACER

lower(*expr*: Funsor) → *Funsor*

Lower a funsor expression: - eliminate bound variables - convert Contraction to Binary

Parameters

expr (Funsor) – An arbitrary funsor expression.

Returns

A lowered funsor expression.

Return type

Funsor

trace_function(*fn*, *kwargs*: dict, *, *allow_constants=False*)

Traces function to an *OpProgram* that runs on backend values.

Example:

```
# Create a function involving ops.
def fn(a, b, x):
    return ops.add(ops.matmul(a, x), b)

# Evaluate via Funsor substitution.
data = dict(a=randn(3, 3), b=randn(3), x=randn(3))
expected = fn(**data)

# Alternatively evaluate via a program.
program = trace_function(expr, data)
actual = program(**data)
assert (actual == expected).all()
```

Parameters

expr (Funsor) – A funsor expression to evaluate.

Returns

An op program.

Return type

OpProgram

class OpProgram(*constants, inputs, operations*)

Bases: *object*

Backend program for evaluating a symbolic funsor expression.

Programs depend on the funsor library only via `funsor.ops` and op registrations; program evaluation does not involve funsor interpretation or rewriting. Programs can be pickled and unpickled.

Parameters

- **expr** (*iterable*) – A list of built-in constants (leaves).
- **inputs** (*iterable*) – A list of string names of program inputs (leaves).
- **operations** (*iterable*) – A list of program operations defining non-leaf nodes in the program dag. Each operations is a tuple (`op`, `arg_ids`) where `op` is a funsor op and `arg_ids` is a tuple of positions of values, starting from zero and counting: constants, inputs, and operation outputs.

`as_code(name='program')`

Returns Python code text defining a straight-line function equivalent to this program.

Parameters

name (*str*) – Optional name for the function, defaults to “program”.

Returns

A string defining a python function equivalent to this program.

Return type

`str`

NAMED TENSOR NOTATION WITH FUNSORS (PART 1)

18.1 Introduction

Mathematical notation with *named axes* introduced in [Named Tensor Notation \(Chiang, Rush, Barak 2021\)](#) improves the readability of mathematical formulas involving multidimensional arrays. This includes tensor operations such as elementwise operations, reductions, contractions, renaming, indexing, and broadcasting. In this tutorial we translate examples from [Named Tensor Notation](#) into [funsors](#) to demonstrate the implementation of these operations in [funsor](#) library and familiarize readers with [funsor](#) syntax. Part 1 covers examples from [2 Informal Overview](#), [3.4.2 Advanced Indexing](#), and [5 Formal Definitions](#).

First, let's import some dependencies.

```
[ ]: !pip install funsor[torch]@git+https://github.com/pyro-ppl/funsor
```

```
[1]: from torch import tensor

import funsor
import funsor.ops as ops
from funsor import Number, Tensor, Variable
from funsor.domains import Bint

funsor.set_backend("torch")
```

18.2 Named Tensors

Each tensor axis is given a name:

$$A \in \mathbb{R}^{\text{height}[3] \times \text{width}[3]} = \mathbb{R}^{\text{width}[3] \times \text{height}[3]}$$
$$A = \text{height} \begin{array}{c} \text{width} \\ \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \end{array} = \text{width} \begin{array}{c} \text{height} \\ \begin{bmatrix} 3 & 1 & 2 \\ 1 & 5 & 6 \\ 4 & 9 & 5 \end{bmatrix} \end{array}.$$

```
[2]: A = Tensor(tensor([[3, 1, 4], [1, 5, 9], [2, 6, 5]]))["height", "width"]
```

Access elements of A using named indices:

$$A_{\text{height}(1), \text{width}(3)} = A_{\text{width}(3), \text{height}(1)} = 4$$

```
[3]: # A(height=0, width=2) =
      A(width=2, height=0)
```

```
[3]: Tensor(tensor(4))
```

Partial indexing:

$$A_{\text{height}(1)} = \begin{matrix} & \text{width} \\ & [3 \quad 1 \quad 4] \end{matrix} \quad A_{\text{width}(3)} = \begin{matrix} & \text{height} \\ [4 \quad 9 \quad 5] & \end{matrix}.$$

```
[4]: A(height=0)
```

```
[4]: Tensor(tensor([3, 1, 4]), {'width': Bint[3]})
```

```
[5]: A(width=2)
```

```
[5]: Tensor(tensor([4, 9, 5]), {'height': Bint[3]})
```

18.3 Named tensor operations

18.3.1 Elementwise operations and broadcasting

Elementwise operations:

$$\frac{1}{1 + \exp(-A)} = \text{height} \begin{matrix} & \text{width} \\ \begin{bmatrix} \frac{1}{1+\exp(-3)} & \frac{1}{1+\exp(-1)} & \frac{1}{1+\exp(-4)} \\ \frac{1}{1+\exp(-1)} & \frac{1}{1+\exp(-5)} & \frac{1}{1+\exp(-9)} \\ \frac{1}{1+\exp(-2)} & \frac{1}{1+\exp(-6)} & \frac{1}{1+\exp(-5)} \end{bmatrix} \end{matrix}.$$

```
[6]: # A.sigmoid() =
      # ops.sigmoid(A) =
      # 1 / (1 + ops.exp(-A)) =
      1 / (1 + (-A).exp())
```

```
[6]: Tensor(tensor([[0.9526, 0.7311, 0.9820],
                   [0.7311, 0.9933, 0.9999],
                   [0.8808, 0.9975, 0.9933]]), {'height': Bint[3], 'width': Bint[3]})
```

Tensors with different shapes are automatically broadcasted against each other before an operation is applied. Let

$$x \in \mathbb{R}^{\text{height}[3]} \quad y \in \mathbb{R}^{\text{width}[3]}$$

$$x = \text{height} \begin{bmatrix} 2 \\ 7 \\ 1 \end{bmatrix} \quad y = \begin{matrix} \text{width} \\ [1 \quad 4 \quad 1] \end{matrix}.$$

```
[7]: x = Tensor(tensor([2, 7, 1]))["height"]
```

```
y = Tensor(tensor([1, 4, 1]))["width"]
```

Binary addition operation:

$$A + x = \text{height} \begin{matrix} & \text{width} \\ \begin{bmatrix} 3+2 & 1+2 & 4+2 \\ 1+7 & 5+7 & 9+7 \\ 2+1 & 6+1 & 5+1 \end{bmatrix} \end{matrix} \quad A + y = \text{height} \begin{matrix} & \text{width} \\ \begin{bmatrix} 3+1 & 1+4 & 4+1 \\ 1+1 & 5+4 & 9+1 \\ 2+1 & 6+4 & 5+1 \end{bmatrix} \end{matrix}.$$

```
[8]: # ops.add(A, x) =
      A + x
```

```
[8]: Tensor(tensor([[ 5,  3,  6],
                   [ 8, 12, 16],
                   [ 3,  7,  6]]), {'height': Bint[3], 'width': Bint[3]})
```

```
[9]: # ops.add(A, y) =
      A + y
```

```
[9]: Tensor(tensor([[ 4,  5,  5],
                   [ 2,  9, 10],
                   [ 3, 10,  6]]), {'height': Bint[3], 'width': Bint[3]})
```

Binary multiplication operation:

$$A \odot x = \text{height} \begin{matrix} & \text{width} \\ \begin{bmatrix} 3 \cdot 2 & 1 \cdot 2 & 4 \cdot 2 \\ 1 \cdot 7 & 5 \cdot 7 & 9 \cdot 7 \\ 2 \cdot 1 & 6 \cdot 1 & 5 \cdot 1 \end{bmatrix} \end{matrix}$$

```
[10]: # ops.mul(A, x) =
       A * x
```

```
[10]: Tensor(tensor([[ 6,  2,  8],
                   [ 7, 35, 63],
                   [ 2,  6,  5]]), {'height': Bint[3], 'width': Bint[3]})
```

Binary maximum operation:

$$\max(A, y) = \text{height} \begin{matrix} & \text{width} \\ \begin{bmatrix} \max(3,1) & \max(1,4) & \max(4,1) \\ \max(1,1) & \max(5,4) & \max(9,1) \\ \max(2,1) & \max(6,4) & \max(5,1) \end{bmatrix} \end{matrix}.$$

```
[11]: ops.max(A, y)
```

```
[11]: Tensor(tensor([[3, 4, 4],
                   [1, 5, 9],
                   [2, 6, 5]]), {'height': Bint[3], 'width': Bint[3]})
```

18.3.2 Reductions

Named axes can be reduced over by calling the `.reduce` method and specifying the `reduction operator` and names of reduced axes. Note that reduction is defined only for operators that are associative and commutative.

$$\sum_{\text{height}} A = \sum_i A_{\text{height}(i)} = \begin{bmatrix} 3 + 1 + 2 & 1 + 5 + 6 & 4 + 9 + 5 \end{bmatrix}^{\text{width}}$$

```
[12]: A.reduce(ops.add, "height")
```

```
[12]: Tensor(tensor([ 6, 12, 18]), {'width': Bint[3]})
```

$$\sum_{\text{width}} A = \sum_j A_{\text{width}(j)} = \begin{bmatrix} 3 + 1 + 4 & 1 + 5 + 9 & 2 + 6 + 5 \end{bmatrix}^{\text{height}}$$

```
[13]: A.reduce(ops.add, "width")
```

```
[13]: Tensor(tensor([ 8, 15, 13]), {'height': Bint[3]})
```

Reduction over multiple axes:

$$\sum_{\substack{\text{height} \\ \text{width}}} A = \sum_i \sum_j A_{\text{height}(i), \text{width}(j)} = 3 + 1 + 4 + 1 + 5 + 9 + 2 + 6 + 5.$$

```
[14]: A.reduce(ops.add, {"height", "width"})
```

```
[14]: Tensor(tensor(36))
```

Multiplication reduction:

$$\prod_{\text{height}} A = \prod_i A_{\text{height}(i)} = \begin{bmatrix} 3 \cdot 1 \cdot 2 & 1 \cdot 5 \cdot 6 & 4 \cdot 9 \cdot 5 \end{bmatrix}^{\text{width}}$$

```
[15]: A.reduce(ops.mul, "height")
```

```
[15]: Tensor(tensor([ 6, 30, 180]), {'width': Bint[3]})
```

Max reduction:

$$\max_{\text{height}} A = \max\{A_{\text{height}(i)} \mid 1 \leq i \leq n\} = \begin{bmatrix} \max(3, 1, 2) & \max(1, 5, 6) & \max(4, 9, 5) \end{bmatrix}^{\text{width}}$$

```
[16]: A.reduce(ops.max, "height")
```

```
[16]: Tensor(tensor([3, 6, 9]), {'width': Bint[3]})
```


18.3.3 Contraction

Contraction operation can be written as elementwise multiplication followed by summation over an axis:

$$A \underset{\text{width}}{\odot} y = \sum_j A_{\text{width}(j)} y_{\text{width}(j)} = \text{height} \begin{bmatrix} 3 \cdot 1 + 1 \cdot 4 + 4 \cdot 1 \\ 1 \cdot 1 + 5 \cdot 4 + 9 \cdot 1 \\ 2 \cdot 1 + 6 \cdot 4 + 5 \cdot 1 \end{bmatrix}.$$

```
[17]: (A * y).reduce(ops.add, "width")
```

```
[17]: Tensor(tensor([[11, 30, 31]]), {'height': Bint[3]})
```

Some other operations from linear algebra:

$$x \underset{\text{height}}{\odot} x = \sum_i x_{\text{height}(i)} x_{\text{height}(i)} \quad \text{inner product}$$

```
[18]: (x * x).reduce(ops.add, "height")
```

```
[18]: Tensor(tensor(54))
```

$$[x \odot y]_{\text{height}(i), \text{width}(j)} = x_{\text{height}(i)} y_{\text{width}(j)} \quad \text{outer product}$$

```
[19]: x * y
```

```
[19]: Tensor(tensor([[ 2,  8,  2],
                  [ 7, 28,  7],
                  [ 1,  4,  1]]), {'height': Bint[3], 'width': Bint[3]})
```

$$A \underset{\text{width}}{\odot} y = \sum_i A_{\text{width}(i)} y_{\text{width}(i)} \quad \text{matrix-vector product}$$

```
[20]: (A * y).reduce(ops.add, "width")
```

```
[20]: Tensor(tensor([[11, 30, 31]]), {'height': Bint[3]})
```

$$x \underset{\text{height}}{\odot} A = \sum_i x_{\text{height}(i)} A_{\text{height}(i)} \quad \text{vector-matrix product}$$

```
[21]: (x * A).reduce(ops.add, "height")
```

```
[21]: Tensor(tensor([[15, 43, 76]]), {'width': Bint[3]})
```

$$A \underset{\text{width}}{\odot} B = \sum_i A_{\text{width}(i)} \odot B_{\text{width}(i)} \quad \text{matrix-matrix product } (B \in \mathbb{R}^{\text{width} \times \text{width}2})$$

```
[22]: B = Tensor(
    tensor([[3, 2, 5], [5, 4, 0], [8, 3, 6]]),
    ["width", "width2"]
```

```
(A * B).reduce(ops.add, "width")
```

```
[22]: Tensor(tensor([[ 46,  22,  39],
                    [100,  49,  59],
                    [ 76,  43,  40]]), {'height': Bint[3], 'width2': Bint[3]})
```

Contraction can be generalized to other binary and reduction operations:

$$\max_{\text{width}}(A + y) = \text{height} \begin{bmatrix} \max(3 + 1, 1 + 4, 4 + 1) \\ \max(1 + 1, 5 + 4, 9 + 1) \\ \max(2 + 1, 6 + 4, 5 + 1) \end{bmatrix}.$$

```
[23]: (A + y).reduce(ops.max, "width")
```

```
[23]: Tensor(tensor([ 5, 10, 10]), {'height': Bint[3]})
```

18.3.4 Renaming and reshaping

Renaming funsors is simple:

$$A_{\text{height} \rightarrow \text{height}2} = \text{height}2 \begin{matrix} \text{width} \\ \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \end{matrix}.$$

```
[24]: # A(height=Variable("height2", Bint[3]))
A(height="height2")
```

```
[24]: Tensor(tensor([[3, 1, 4],
                    [1, 5, 9],
                    [2, 6, 5]]), {'height2': Bint[3], 'width': Bint[3]})
```

$$A_{(\text{height}, \text{width}) \rightarrow \text{layer}} = \begin{matrix} \text{layer} \\ [3 & 1 & 4 & 1 & 5 & 9 & 2 & 6 & 5] \end{matrix}$$

```
[25]: layer = Variable("layer", Bint[9])
```

```
A_layer = A(height=layer // Number(3, 4), width=layer % Number(3, 4))
A_layer
```

```
[25]: Tensor(tensor([[3, 1, 4, 1, 5, 9, 2, 6, 5]]), {'layer': Bint[9]})
```

$$A_{\text{layer} \rightarrow (\text{height}, \text{width})} = \text{height} \begin{matrix} & \text{width} \\ \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \end{matrix}.$$

```
[26]: height = Variable("height", Bint[3])
width = Variable("width", Bint[3])

A_layer(layer=height * Number(3, 4) + width % Number(3, 4))
```

```
[26]: Tensor(tensor([[3, 1, 4],
                    [1, 5, 9],
                    [2, 6, 5]]), {'height': Bint[3], 'width': Bint[3]})
```

18.4 Advanced indexing

All of advanced indexing can be achieved through name substitutions in funsors.

$$\text{index}_{\text{ax}}: \mathbb{R}^{\text{ax}[n]} \times [n] \rightarrow \mathbb{R}$$

$$\text{index}_{\text{ax}}(A, i) = A_{\text{ax}(i)}.$$

$$E \in \mathbb{R}^{\text{vocab}[n] \times \text{emb}}$$

$$i \in [n]$$

$$I \in [n]^{\text{seq}}$$

$$P \in \mathbb{R}^{\text{seq} \times \text{vocab}[n]}$$

Partial indexing $\text{index}_{\text{vocab}}(E, i)$:

```
[27]: E = Tensor(
    tensor([[2, 1, 5], [3, 4, 2], [1, 3, 7], [1, 4, 3], [5, 9, 2]]),
    ["vocab", "emb"])
```

```
E(vocab=2)
```

```
[27]: Tensor(tensor([1, 3, 7]), {'emb': Bint[3]})
```

Integer array indexing $\text{index}_{\text{vocab}}(E, I)$:

```
[28]: I = Tensor(tensor([3, 2, 4, 0]), dtype=5)["seq"]
```

```
E(vocab=I)
```

```
[28]: Tensor(tensor([[1, 4, 3],
                    [1, 3, 7],
                    [5, 9, 2],
                    [2, 1, 5]]), {'seq': Bint[4], 'emb': Bint[3]})
```

Gather operation $\text{index}_{\text{vocab}}(P, I)$:

```
[29]: P = Tensor(
    tensor([[6, 2, 4, 2], [8, 2, 1, 3], [5, 5, 7, 0], [1, 3, 8, 2], [5, 9, 2, 3]]),
    ["vocab", "seq"]
```

P(vocab=I)

```
[29]: Tensor(tensor([1, 5, 2, 2]), {'seq': Bint[4]})
```

Indexing with two integer arrays:

$$\begin{aligned}
 |\text{seq}| &= m \\
 I_1 &= [m]^{\text{subseq}} \\
 I_2 &= [n]^{\text{subseq}} \\
 S &= \underset{\text{vocab}}{\text{index}}(\underset{\text{seq}}{\text{index}}(P, I_1), I_2) \in \mathbb{R}^{\text{subseq}} \\
 S_{\text{subseq}(i)} &= P_{\text{seq}(I_{\text{subseq}(i)}), \text{vocab}(I_{\text{subseq}(i)})}.
 \end{aligned}$$

```
[30]: I1 = Tensor(tensor([1, 2, 0]), dtype=4)["subseq"]
      I2 = Tensor(tensor([3, 0, 4]), dtype=5)["subseq"]
```

P(seq=I1, vocab=I2)

```
[30]: Tensor(tensor([3, 4, 5]), {'subseq': Bint[3]})
```

EXAMPLE: ADAM OPTIMIZER

```
import argparse

import torch

import funsor
import funsor.ops as ops
from funsor.adam import Adam
from funsor.domains import Real, Reals
from funsor.tensor import Tensor
from funsor.terms import Variable

def main(args):
    funsor.set_backend("torch")

    # Problem definition.
    N = 100
    P = 10
    data = Tensor(torch.randn(N, P))["n"]
    true_weight = Tensor(torch.randn(P))
    true_bias = Tensor(torch.randn(1))
    truth = true_bias + true_weight @ data

    # Model.
    weight = Variable("weight", Reals[P])
    bias = Variable("bias", Real)
    pred = bias + weight @ data
    loss = (pred - truth).abs().reduce(ops.add, "n")

    # Inference.
    with Adam(args.num_steps, lr=args.learning_rate, log_every=args.log_every) as optim:
        loss.reduce(ops.min, {"weight", "bias"})

    print(f"True bias\n{true_bias}")
    print("Learned bias\n{}".format(optim.param("bias")))
    print(f"True weight\n{true_weight}")
    print("Learned weight\n{}".format(optim.param("weight")))

if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
parser = argparse.ArgumentParser(  
    description="Linear regression example using Adam interpretation"  
)  
parser.add_argument("-P", "--num-features", type=int, default=10)  
parser.add_argument("-N", "--num-data", type=int, default=100)  
parser.add_argument("-n", "--num-steps", type=int, default=201)  
parser.add_argument("-lr", "--learning-rate", type=float, default=0.05)  
parser.add_argument("--log-every", type=int, default=20)  
args = parser.parse_args()  
main(args)
```

EXAMPLE: DISCRETE HMM

```
import argparse
from collections import OrderedDict

import torch

import funsor
import funsor.ops as ops
import funsor.torch.distributions as dist
from funsor.interpreter import reinterpret
from funsor.optimizer import apply_optimizer

def main(args):
    funsor.set_backend("torch")

    # Declare parameters.
    trans_probs = torch.tensor([[0.2, 0.8], [0.7, 0.3]], requires_grad=True)
    emit_probs = torch.tensor([[0.4, 0.6], [0.1, 0.9]], requires_grad=True)
    params = [trans_probs, emit_probs]

    # A discrete HMM model.
    def model(data):
        log_prob = funsor.to_funsor(0.0)

        trans = dist.Categorical(
            probs=funsor.Tensor(
                trans_probs,
                inputs=OrderedDict([("prev", funsor.Bint[args.hidden_dim])]),
            )
        )

        emit = dist.Categorical(
            probs=funsor.Tensor(
                emit_probs,
                inputs=OrderedDict([("latent", funsor.Bint[args.hidden_dim])]),
            )
        )

        x_curr = funsor.Number(0, args.hidden_dim)
        for t, y in enumerate(data):
```

(continues on next page)

```

x_prev = x_curr

# A delayed sample statement.
x_curr = funsor.Variable("x_{}".format(t), funsor.Bint[args.hidden_dim])
log_prob += trans(prev=x_prev, value=x_curr)

if not args.lazy and isinstance(x_prev, funsor.Variable):
    log_prob = log_prob.reduce(ops.logaddexp, x_prev.name)

log_prob += emit(latent=x_curr, value=funsor.Tensor(y, dtype=2))

log_prob = log_prob.reduce(ops.logaddexp)
return log_prob

# Train model parameters.
data = torch.ones(args.time_steps, dtype=torch.long)
optim = torch.optim.Adam(params, lr=args.learning_rate)
for step in range(args.train_steps):
    optim.zero_grad()
    if args.lazy:
        with funsor.interpretations.lazy:
            log_prob = apply_optimizer(model(data))
            log_prob = reinterpret(log_prob)
    else:
        log_prob = model(data)
    assert not log_prob.inputs, "free variables remain"
    loss = -log_prob.data
    loss.backward()
    optim.step()

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Kalman filter example")
    parser.add_argument("-t", "--time-steps", default=10, type=int)
    parser.add_argument("-n", "--train-steps", default=101, type=int)
    parser.add_argument("-lr", "--learning-rate", default=0.05, type=float)
    parser.add_argument("-d", "--hidden-dim", default=2, type=int)
    parser.add_argument("--lazy", action="store_true")
    parser.add_argument("--filter", action="store_true")
    parser.add_argument("--xfail-if-not-implemented", action="store_true")
    args = parser.parse_args()

    if args.xfail_if_not_implemented:
        try:
            main(args)
        except NotImplementedError:
            print("XFAIL")
    else:
        main(args)

```


EXAMPLE: SWITCHING LINEAR DYNAMICAL SYSTEM EEG

We use a switching linear dynamical system [1] to model a EEG time series dataset. For inference we use a moment-matching approximation enabled by *funsor.interpretations.moment_matching*.

References

[1] Anderson, B., and J. Moore. "Optimal filtering. Prentice-Hall, Englewood Cliffs." New Jersey (1979).

```
import argparse
import time
from collections import OrderedDict
from os.path import exists
from urllib.request import urlopen

import numpy as np
import pyro
import torch
import torch.nn as nn

import funsor
import funsor.ops as ops
import funsor.torch.distributions as dist
from funsor.pyro.convert import (
    funsor_to_cat_and_mvn,
    funsor_to_mvn,
    matrix_and_mvn_to_funsor,
    mvn_to_funsor,
)

# download dataset from UCI archive
def download_data():
    if not exists("eeg.dat"):
        url = "http://archive.ics.uci.edu/ml/machine-learning-databases/00264/EEG%20Eye
↳%20State.arff"
        with open("eeg.dat", "wb") as f:
            f.write(urlopen(url).read())

class SLDS(nn.Module):
    def __init__(
        self,
```

(continues on next page)

(continued from previous page)

```

num_components, # the number of switching states K
hidden_dim, # the dimension of the continuous latent space
obs_dim, # the dimension of the continuous outputs
fine_transition_matrix=True, # controls whether the transition matrix depends_
↪on s_t
fine_transition_noise=False, # controls whether the transition noise depends on_
↪s_t
fine_observation_matrix=False, # controls whether the observation matrix_
↪depends on s_t
fine_observation_noise=False, # controls whether the observation noise depends_
↪on s_t
moment_matching_lag=1,
): # controls the expense of the moment matching approximation
self.num_components = num_components
self.hidden_dim = hidden_dim
self.obs_dim = obs_dim
self.moment_matching_lag = moment_matching_lag
self.fine_transition_noise = fine_transition_noise
self.fine_observation_matrix = fine_observation_matrix
self.fine_observation_noise = fine_observation_noise
self.fine_transition_matrix = fine_transition_matrix

assert moment_matching_lag > 0
assert (
    fine_transition_noise
    or fine_observation_matrix
    or fine_observation_noise
    or fine_transition_matrix
), (
    "The continuous dynamics need to be coupled to the discrete dynamics in at_
↪least one way [use at "
    + "least one of the arguments --ftn --ftm --fon --fom]"
)

super(SLDS, self).__init__()

# initialize the various parameters of the model
self.transition_logits = nn.Parameter(
    0.1 * torch.randn(num_components, num_components)
)
if fine_transition_matrix:
    transition_matrix = torch.eye(hidden_dim) + 0.05 * torch.randn(
        num_components, hidden_dim, hidden_dim
    )
else:
    transition_matrix = torch.eye(hidden_dim) + 0.05 * torch.randn(
        hidden_dim, hidden_dim
    )
self.transition_matrix = nn.Parameter(transition_matrix)
if fine_transition_noise:
    self.log_transition_noise = nn.Parameter(
        0.1 * torch.randn(num_components, hidden_dim)

```

(continues on next page)

(continued from previous page)

```

    )
    else:
        self.log_transition_noise = nn.Parameter(0.1 * torch.randn(hidden_dim))
    if fine_observation_matrix:
        self.observation_matrix = nn.Parameter(
            0.3 * torch.randn(num_components, hidden_dim, obs_dim)
        )
    else:
        self.observation_matrix = nn.Parameter(
            0.3 * torch.randn(hidden_dim, obs_dim)
        )
    if fine_observation_noise:
        self.log_obs_noise = nn.Parameter(
            0.1 * torch.randn(num_components, obs_dim)
        )
    else:
        self.log_obs_noise = nn.Parameter(0.1 * torch.randn(obs_dim))

    # define the prior distribution p(x_0) over the continuous latent at the initial_
    ↪time step t=0
    x_init_mvn = pyro.distributions.MultivariateNormal(
        torch.zeros(self.hidden_dim), torch.eye(self.hidden_dim)
    )
    self.x_init_mvn = mvn_to_funsor(
        x_init_mvn,
        real_inputs=OrderedDict([("x_0", funsor.Reals[self.hidden_dim])]),
    )

    # we construct the various funsors used to compute the marginal log probability and_
    ↪other model quantities.
    # these funsors depend on the various model parameters.
    def get_tensors_and_dists(self):
        # normalize the transition probabilities
        trans_logits = self.transition_logits - self.transition_logits.logsumexp(
            dim=-1, keepdim=True
        )
        trans_probs = funsor.Tensor(
            trans_logits, OrderedDict([("s", funsor.Bint[self.num_components])])
        )

        trans_mvn = pyro.distributions.MultivariateNormal(
            torch.zeros(self.hidden_dim), self.log_transition_noise.exp().diag_embed()
        )
        obs_mvn = pyro.distributions.MultivariateNormal(
            torch.zeros(self.obs_dim), self.log_obs_noise.exp().diag_embed()
        )

        event_dims = (
            ("s",) if self.fine_transition_matrix or self.fine_transition_noise else ()
        )
        x_trans_dist = matrix_and_mvn_to_funsor(
            self.transition_matrix, trans_mvn, event_dims, "x", "y"

```

(continues on next page)

(continued from previous page)

```

)
event_dims = (
    ("s",)
    if self.fine_observation_matrix or self.fine_observation_noise
    else ()
)
y_dist = matrix_and_mvn_to_funsor(
    self.observation_matrix, obs_mvn, event_dims, "x", "y"
)

return trans_logits, trans_probs, trans_mvn, obs_mvn, x_trans_dist, y_dist

# compute the marginal log probability of the observed data using a moment-matching_
↪approximation
@funsor.interpretations.moment_matching
def log_prob(self, data):
    (
        trans_logits,
        trans_probs,
        trans_mvn,
        obs_mvn,
        x_trans_dist,
        y_dist,
    ) = self.get_tensors_and_dists()

    log_prob = funsor.Number(0.0)

    s_vars = {-1: funsor.Tensor(torch.tensor(0), dtype=self.num_components)}
    x_vars = {}

    for t, y in enumerate(data):
        # construct free variables for s_t and x_t
        s_vars[t] = funsor.Variable(f"s_{t}", funsor.Bint[self.num_components])
        x_vars[t] = funsor.Variable(f"x_{t}", funsor.Reals[self.hidden_dim])

        # incorporate the discrete switching dynamics
        log_prob += dist.Categorical(trans_probs(s=s_vars[t - 1]), value=s_vars[t])

        # incorporate the prior term p(x_t | x_{t-1})
        if t == 0:
            log_prob += self.x_init_mvn(value=x_vars[t])
        else:
            log_prob += x_trans_dist(s=s_vars[t], x=x_vars[t - 1], y=x_vars[t])

        # do a moment-matching reduction. at this point log_prob depends on (moment_
↪matching_lag + 1)-many
        # pairs of free variables.
        if t > self.moment_matching_lag - 1:
            log_prob = log_prob.reduce(
                ops.logaddexp,
                {
                    s_vars[t - self.moment_matching_lag],

```

(continues on next page)

(continued from previous page)

```

        x_vars[t - self.moment_matching_lag],
    },
)

# incorporate the observation p(y_t | x_t, s_t)
log_prob += y_dist(s=s_vars[t], x=x_vars[t], y=y)

T = data.shape[0]
# reduce any remaining free variables
for t in range(self.moment_matching_lag):
    log_prob = log_prob.reduce(
        ops.logaddexp,
        {
            s_vars[T - self.moment_matching_lag + t],
            x_vars[T - self.moment_matching_lag + t],
        },
    )

# assert that we've reduced all the free variables in log_prob
assert not log_prob.inputs, "unexpected free variables remain"

# return the PyTorch tensor behind log_prob (which we can directly differentiate)
return log_prob.data

# do filtering, prediction, and smoothing using a moment-matching approximation.
# here we implicitly use a moment matching lag of L = 1. the general logic follows
# the logic in the log_prob method.
@torch.no_grad()
@funsor.interpretations.moment_matching
def filter_and_predict(self, data, smoothing=False):
    (
        trans_logits,
        trans_probs,
        trans_mvn,
        obs_mvn,
        x_trans_dist,
        y_dist,
    ) = self.get_tensors_and_dists()

    log_prob = funsor.Number(0.0)

    s_vars = {-1: funsor.Tensor(torch.tensor(0), dtype=self.num_components)}
    x_vars = {-1: None}

    predictive_x_dists, predictive_y_dists, filtering_dists = [], [], []
    test_LLs = []

    for t, y in enumerate(data):
        s_vars[t] = funsor.Variable(f"s_{t}", funsor.Bint[self.num_components])
        x_vars[t] = funsor.Variable(f"x_{t}", funsor.Reals[self.hidden_dim])

        log_prob += dist.Categorical(trans_probs(s=s_vars[t - 1]), value=s_vars[t])

```

(continues on next page)

```

if t == 0:
    log_prob += self.x_init_mvn(value=x_vars[t])
else:
    log_prob += x_trans_dist(s=s_vars[t], x=x_vars[t - 1], y=x_vars[t])

if t > 0:
    log_prob = log_prob.reduce(
        ops.logaddexp, {s_vars[t - 1], x_vars[t - 1]}
    )

# do 1-step prediction and compute test LL
if t > 0:
    predictive_x_dists.append(log_prob)
    _log_prob = log_prob - log_prob.reduce(ops.logaddexp)
    predictive_y_dist = y_dist(s=s_vars[t], x=x_vars[t]) + _log_prob
    test_LLs.append(
        predictive_y_dist(y=y).reduce(ops.logaddexp).data.item()
    )
    predictive_y_dist = predictive_y_dist.reduce(
        ops.logaddexp, {f"x_{t}", f"s_{t}"}
    )
    predictive_y_dists.append(funsor_to_mvn(predictive_y_dist, 0, ()))

log_prob += y_dist(s=s_vars[t], x=x_vars[t], y=y)

# save filtering dists for forward-backward smoothing
if smoothing:
    filtering_dists.append(log_prob)

# do the backward recursion using previously computed ingredients
if smoothing:
    # seed the backward recursion with the filtering distribution at t=T
    smoothing_dists = [filtering_dists[-1]]
    T = data.size(0)

    s_vars = {
        t: funsor.Variable(f"s_{t}", funsor.Bint[self.num_components])
        for t in range(T)
    }
    x_vars = {
        t: funsor.Variable(f"x_{t}", funsor.Reals[self.hidden_dim])
        for t in range(T)
    }

    # do the backward recursion.
    # let p[t|t-1] be the predictive distribution at time step t.
    # let p[t|t] be the filtering distribution at time step t.
    # let f[t] denote the prior (transition) density at time step t.
    # then the smoothing distribution p[t|T] at time step t is
    # given by the following recursion.
    #  $p[t-1|T] = p[t-1|t-1] \langle p[t|T] f[t] / p[t|t-1] \rangle$ 

```

(continues on next page)

(continued from previous page)

```

# where <...> denotes integration of the latent variables at time step t.
for t in reversed(range(T - 1)):
    integral = smoothing_dists[-1] - predictive_x_dists[t]
    integral += dist.Categorical(
        trans_probs(s=s_vars[t]), value=s_vars[t + 1]
    )
    integral += x_trans_dist(s=s_vars[t], x=x_vars[t], y=x_vars[t + 1])
    integral = integral.reduce(
        ops.logaddexp, {s_vars[t + 1], x_vars[t + 1]}
    )
    smoothing_dists.append(filtering_dists[t] + integral)

# compute predictive test MSE and predictive variances
predictive_means = torch.stack([d.mean for d in predictive_y_dists]) # T-1 ydim
predictive_vars = torch.stack(
    [d.covariance_matrix.diagonal(dim1=-1, dim2=-2) for d in predictive_y_dists]
)
predictive_mse = (predictive_means - data[1:, :]).pow(2.0).mean(-1)

if smoothing:
    # compute smoothed mean function
    smoothing_dists = [
        funsor_to_cat_and_mvn(d, 0, (f"s_{t}",))
        for t, d in enumerate(reversed(smoothing_dists))
    ]
    means = torch.stack([d[1].mean for d in smoothing_dists]) # T 2 xdim
    means = torch.matmul(means.unsqueeze(-2), self.observation_matrix).squeeze(-2)
    ) # T 2 ydim

    probs = torch.stack([d[0].logits for d in smoothing_dists]).exp()
    probs = probs / probs.sum(-1, keepdim=True) # T 2

    smoothing_means = (probs.unsqueeze(-1) * means).sum(-2) # T ydim
    smoothing_probs = probs[:, 1]

    return (
        predictive_mse,
        torch.tensor(np.array(test_LLs)),
        predictive_means,
        predictive_vars,
        smoothing_means,
        smoothing_probs,
    )
else:
    return predictive_mse, torch.tensor(np.array(test_LLs))

def main(args):
    funsor.set_backend("torch")

    # download and pre-process EEG data if not in test mode

```

(continues on next page)

```

if not args.test:
    download_data()
    N_val, N_test = 149, 200
    data = np.loadtxt("eeg.dat", delimiter=",", skiprows=19)
    print(f"[raw data shape] {data.shape}")
    data = data[:, :20, :]
    print(f"[data shape after thinning] {data.shape}")
    eye_state = [int(d) for d in data[:, -1].tolist()]
    data = torch.tensor(data[:, :-1]).float()
# in test mode (for continuous integration on github) so create fake data
else:
    data = torch.randn(10, 3)
    N_val, N_test = 2, 2

T, obs_dim = data.shape
N_train = T - N_test - N_val

np.random.seed(0)
rand_perm = np.random.permutation(N_val + N_test)
val_indices = rand_perm[0:N_val]
test_indices = rand_perm[N_val:]

data_mean = data[0:N_train, :].mean(0)
data -= data_mean
data_std = data[0:N_train, :].std(0)
data /= data_std

print(f"Length of time series T: {T}   Observation dimension: {obs_dim}")
print(f"N_train: {N_train}   N_val: {N_val}   N_test: {N_test}")

torch.manual_seed(args.seed)

# set up model
slds = SLDS(
    num_components=args.num_components,
    hidden_dim=args.hidden_dim,
    obs_dim=obs_dim,
    fine_observation_noise=args.fon,
    fine_transition_noise=args.ftn,
    fine_observation_matrix=args.fom,
    fine_transition_matrix=args.ftm,
    moment_matching_lag=args.moment_matching_lag,
)

# set up optimizer
adam = torch.optim.Adam(
    slds.parameters(),
    lr=args.learning_rate,
    betas=(args.beta1, 0.999),
    amsgrad=True,
)
scheduler = torch.optim.lr_scheduler.ExponentialLR(adam, gamma=args.gamma)

```

(continues on next page)

(continued from previous page)

```

ts = [time.time()]

report_frequency = 1

# training loop
for step in range(args.num_steps):
    nll = -slds.log_prob(data[0:N_train, :]) / N_train
    nll.backward()

    if step == 5:
        scheduler.base_lrs[0] *= 0.20

    adam.step()
    scheduler.step()
    adam.zero_grad()

    if step % report_frequency == 0 or step == args.num_steps - 1:
        step_dt = ts[-1] - ts[-2] if step > 0 else 0.0
        pred_mse, pred_LLs = slds.filter_and_predict(
            data[0 : N_train + N_val + N_test, :]
        )
        val_mse = pred_mse[val_indices].mean().item()
        test_mse = pred_mse[test_indices].mean().item()
        val_ll = pred_LLs[val_indices].mean().item()
        test_ll = pred_LLs[test_indices].mean().item()

        stats = "[step %03d] train_nll: %.5f val_mse: %.5f val_ll: %.5f test_mse: %.
↪5f test_ll: %.5f\t(dt: %.2f)"
        print(
            stats % (step, nll.item(), val_mse, val_ll, test_mse, test_ll, step_dt)
        )

    ts.append(time.time())

# plot predictions and smoothed means
if args.plot:
    assert not args.test
    (
        predicted_mse,
        LLs,
        pred_means,
        pred_vars,
        smooth_means,
        smooth_probs,
    ) = slds.filter_and_predict(data, smoothing=True)

    pred_means = pred_means.data.numpy()
    pred_stds = pred_vars.sqrt().data.numpy()
    smooth_means = smooth_means.data.numpy()
    smooth_probs = smooth_probs.data.numpy()

import matplotlib

```

(continues on next page)

(continued from previous page)

```

matplotlib.use("Agg") # noqa: E402
import matplotlib.pyplot as plt

f, axes = plt.subplots(4, 1, figsize=(12, 8), sharex=True)
T = data.size(0)
N_valtest = N_val + N_test
to_seconds = 117.0 / T

for k, ax in enumerate(axes[:-1]):
    which = [0, 4, 10][k]
    ax.plot(to_seconds * np.arange(T), data[:, which], "ko", markersize=2)
    ax.plot(
        to_seconds * np.arange(N_train),
        smooth_means[:N_train, which],
        ls="solid",
        color="r",
    )
    ax.plot(
        to_seconds * (N_train + np.arange(N_valtest)),
        pred_means[-N_valtest:, which],
        ls="solid",
        color="b",
    )
    ax.fill_between(
        to_seconds * (N_train + np.arange(N_valtest)),
        pred_means[-N_valtest:, which] - 1.645 * pred_stds[-N_valtest:, which],
        pred_means[-N_valtest:, which] + 1.645 * pred_stds[-N_valtest:, which],
        color="lightblue",
    )
    ax.set_ylabel(f"$y_{which + 1}$", fontsize=20)
    ax.tick_params(axis="both", which="major", labelsize=14)

axes[-1].plot(to_seconds * np.arange(T), eye_state, "k", ls="solid")
axes[-1].plot(to_seconds * np.arange(T), smooth_probs, "r", ls="solid")
axes[-1].set_xlabel("Time (s)", fontsize=20)
axes[-1].set_ylabel("Eye state", fontsize=20)
axes[-1].tick_params(axis="both", which="major", labelsize=14)

plt.tight_layout(pad=0.7)
plt.savefig("eeg.pdf")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Switching linear dynamical system")
    parser.add_argument("-n", "--num-steps", default=3, type=int)
    parser.add_argument("-s", "--seed", default=15, type=int)
    parser.add_argument("-hd", "--hidden-dim", default=5, type=int)
    parser.add_argument("-k", "--num-components", default=2, type=int)
    parser.add_argument("-lr", "--learning-rate", default=0.5, type=float)
    parser.add_argument("-b1", "--beta1", default=0.75, type=float)

```

(continues on next page)

(continued from previous page)

```
parser.add_argument("-g", "--gamma", default=0.99, type=float)
parser.add_argument("-mml", "--moment-matching-lag", default=1, type=int)
parser.add_argument("--plot", action="store_true")
parser.add_argument("--fon", action="store_true")
parser.add_argument("--ftm", action="store_true")
parser.add_argument("--fom", action="store_true")
parser.add_argument("--ftn", action="store_true")
parser.add_argument("--test", action="store_true")
args = parser.parse_args()

main(args)
```


EXAMPLE: FORWARD-BACKWARD ALGORITHM

```
import argparse
from collections import OrderedDict
from typing import Dict, List, Tuple

import funsor.ops as ops
from funsor import Funsor, Tensor
from funsor.adjoint import AdjointTape
from funsor.domains import Bint
from funsor.testing import assert_close, random_tensor

def forward_algorithm(
    factors: List[Funsor],
    step: Dict[str, str],
) -> Tuple[Funsor, List[Funsor]]:
    """
    Calculate log marginal probability using the forward algorithm:
     $Z = p(y[0:T])$ 

    Transition and emission probabilities are given by init and trans factors:
     $init = p(y[0], x[0])$ 
     $trans[t] = p(y[t], x[t] | x[t-1])$ 

    Forward probabilities are computed inductively:
     $alpha[t] = p(y[0:t], x[t])$ 
     $alpha[0] = init$ 
     $alpha[t+1] = alpha[t] @ trans[t+1]$ 
    """
    step = OrderedDict(sorted(step.items()))
    drop = tuple("_drop_{}".format(i) for i in range(len(step)))
    prev_to_drop = dict(zip(step.keys(), drop))
    curr_to_drop = dict(zip(step.values(), drop))
    reduce_vars = frozenset(drop)

    # base case
    alpha = factors[0]
    alphas = [alpha]
    # inductive steps
    for trans in factors[1:]:
        alpha = (alpha(**curr_to_drop) + trans(**prev_to_drop)).reduce(
```

(continues on next page)

```

        ops.logaddexp, reduce_vars
    )
    alphas.append(alpha)
else:
    Z = alpha(**curr_to_drop).reduce(ops.logaddexp, reduce_vars)
return Z

def forward_backward_algorithm(
    factors: List[Funsor],
    step: Dict[str, str],
) -> List[Tensor]:
    """
    Calculate marginal probabilities:
     $p(x[t], x[t-1] \mid Y)$ 
    """
    step = OrderedDict(sorted(step.items()))
    drop = tuple("_drop_{}".format(i) for i in range(len(step)))
    prev_to_drop = dict(zip(step.keys(), drop))
    curr_to_drop = dict(zip(step.values(), drop))
    reduce_vars = frozenset(drop)

    # Base cases
    alpha = factors[0] #  $\alpha[0] = p(y[0], x[0])$ 
    beta = Tensor(
        ops.full_like(alpha.data, 0.0), alpha(x_curr="x_prev").inputs
    ) #  $\beta[T] = 1$ 

    # Backward algorithm
    #  $\beta[t] = p(y[t+1:T] \mid x[t])$ 
    #  $\beta[t] = \text{trans}[t+1] @ \beta[t+1]$ 
    betas = [beta]
    for trans in factors[:0:-1]:
        beta = (trans(**curr_to_drop) + beta(**prev_to_drop)).reduce(
            ops.logaddexp, reduce_vars
        )
        betas.append(beta)
    else:
        init = factors[0]
        Z = (init(**curr_to_drop) + beta(**prev_to_drop)).reduce(
            ops.logaddexp, reduce_vars
        )
    betas.reverse()

    # Forward algorithm & Marginal computations
    marginal = alpha + betas[0](**{"x_prev": "x_curr"}) - Z #  $p(x[0] \mid Y)$ 
    marginals = [marginal]
    # inductive steps
    for trans, beta in zip(factors[1:], betas[1:]):
        #  $\alpha[t-1] * \text{trans}[t] = p(y[0:t], x[t-1], x[t])$ 
        alpha_trans = alpha(**{"x_curr": "x_prev"}) + trans
        #  $\alpha[t] = p(y[0:t], x[t])$ 

```

(continues on next page)

(continued from previous page)

```

alpha = alpha_trans.reduce(ops.logaddexp, "x_prev")
# alpha[t-1] * trans[t] * beta[t] / Z = p(x[t-1], x[t] | Y)
marginal = alpha_trans + beta(**{"x_prev": "x_curr"}) - Z
marginals.append(marginal)

return marginals

def main(args):
    """
    Compute marginal probabilities  $p(x[t], x[t-1] | Y)$  for an HMM:

    x[0] -> ... -> x[t-1] -> x[t] -> ... -> x[T]
    |           |           |           |
    v           v           v           v
    y[0]       y[t-1]     y[t]       y[T]

    Z = p(y[0:T])
    alpha[t] = p(y[0:t], x[t])
    beta[t] = p(y[t+1:T] | x[t])
    trans[t] = p(y[t], x[t] | x[t-1])

     $p(x[t], x[t-1] | Y) = \alpha[t-1] * \beta[t] * \text{trans}[t] / Z$ 

     $d Z / d \text{trans}[t] = \alpha[t-1] * \beta[t]$ 

    **References:**

    [1] Jason Eisner (2016)
        "Inside-Outside and Forward-Backward Algorithms Are Just Backprop
        (Tutorial Paper)"
        https://www.cs.jhu.edu/~jason/papers/eisner.spnlp16.pdf
    [2] Zhifei Li and Jason Eisner (2009)
        "First- and Second-Order Expectation Semirings
        with Applications to Minimum-Risk Training on Translation Forests"
        http://www.cs.jhu.edu/~zfli/pubs/semiring\_translation\_zhifei\_emnlp09.pdf
    """

    # transition and emission probabilities
    init = random_tensor(OrderedDict([("x_curr", Bint[args.hidden_dim])]))
    factors = [init]
    for t in range(args.time_steps - 1):
        factors.append(
            random_tensor(
                OrderedDict(x_prev=Bint[args.hidden_dim], x_curr=Bint[args.hidden_dim])
            )
        )

    # Compute marginal probabilities using the forward-backward algorithm
    marginals = forward_backward_algorithm(factors, {"x_prev": "x_curr"})
    # Compute marginal probabilities using backpropagation
    with AdjointTape() as tape:

```

(continues on next page)

(continued from previous page)

```
Z = forward_algorithm(factors, {"x_prev": "x_curr"})
result = tape.adjoint(ops.logaddexp, ops.add, Z, factors)
adjoint_terms = list(result.values())

t = 0
for expected, adj, trans in zip(marginals, adjoint_terms, factors):
    # adjoint returns dZ / dtrans = alpha[t-1] * beta[t]
    # marginal = alpha[t-1] * beta[t] * trans / Z
    actual = adj + trans - Z
    assert_close(expected, actual.align(tuple(expected.inputs)), rtol=1e-4)
    print("")
    print(f"Marginal term: p(x[{t}], x[{t-1}] | Y)")
    print("Forward-backward algorithm:\n", expected.data)
    print("Differentiating forward algorithm:\n", actual.data)
    t += 1

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Forward-Backward Algorithm Is Just Backprop"
    )
    parser.add_argument("-t", "--time-steps", default=10, type=int)
    parser.add_argument("-d", "--hidden-dim", default=3, type=int)
    args = parser.parse_args()

    main(args)
```


EXAMPLE: KALMAN FILTER

```
import argparse

import torch

import funsor
import funsor.ops as ops
import funsor.torch.distributions as dist
from funsor.interpreter import reinterpret
from funsor.optimizer import apply_optimizer

def main(args):
    funsor.set_backend("torch")

    # Declare parameters.
    trans_noise = torch.tensor(0.1, requires_grad=True)
    emit_noise = torch.tensor(0.5, requires_grad=True)
    params = [trans_noise, emit_noise]

    # A Gaussian HMM model.
    def model(data):
        log_prob = funsor.to_funsor(0.0)

        x_curr = funsor.Tensor(torch.tensor(0.0))
        for t, y in enumerate(data):
            x_prev = x_curr

            # A delayed sample statement.
            x_curr = funsor.Variable("x_{}".format(t), funsor.Real)
            log_prob += dist.Normal(1 + x_prev / 2.0, trans_noise, value=x_curr)

            # Optionally marginalize out the previous state.
            if t > 0 and not args.lazy:
                log_prob = log_prob.reduce(ops.logaddexp, x_prev.name)

            # An observe statement.
            log_prob += dist.Normal(0.5 + 3 * x_curr, emit_noise, value=y)

        # Marginalize out all remaining delayed variables.
        log_prob = log_prob.reduce(ops.logaddexp)
```

(continues on next page)

```
    return log_prob

    # Train model parameters.
    torch.manual_seed(0)
    data = torch.randn(args.time_steps)
    optim = torch.optim.Adam(params, lr=args.learning_rate)
    for step in range(args.train_steps):
        optim.zero_grad()
        if args.lazy:
            with funsor.interpretations.lazy:
                log_prob = apply_optimizer(model(data))
            log_prob = reinterpret(log_prob)
        else:
            log_prob = model(data)
        assert not log_prob.inputs, "free variables remain"
        loss = -log_prob.data
        loss.backward()
        optim.step()
        if args.verbose and step % 10 == 0:
            print("step {} loss = {}".format(step, loss.item()))

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Kalman filter example")
    parser.add_argument("-t", "--time-steps", default=10, type=int)
    parser.add_argument("-n", "--train-steps", default=101, type=int)
    parser.add_argument("-lr", "--learning-rate", default=0.05, type=float)
    parser.add_argument("--lazy", action="store_true")
    parser.add_argument("--filter", action="store_true")
    parser.add_argument("-v", "--verbose", action="store_true")
    args = parser.parse_args()
    main(args)
```

EXAMPLE: MINI PYRO

```
import argparse

import torch
from pyroapi import distributions as dist
from pyroapi import infer, optim, pyro, pyro_backend
from torch.distributions import constraints

import funsor
from funsor.montecarlo import MonteCarlo

def main(args):
    funsor.set_backend("torch")

    # Define a basic model with a single Normal latent random variable `loc`
    # and a batch of Normally distributed observations.
    def model(data):
        loc = pyro.sample("loc", dist.Normal(0.0, 1.0))
        with pyro.plate("data", len(data), dim=-1):
            pyro.sample("obs", dist.Normal(loc, 1.0), obs=data)

    # Define a guide (i.e. variational distribution) with a Normal
    # distribution over the latent random variable `loc`.
    def guide(data):
        guide_loc = pyro.param("guide_loc", torch.tensor(0.0))
        guide_scale = pyro.param(
            "guide_scale", torch.tensor(1.0), constraint=constraints.positive
        )
        pyro.sample("loc", dist.Normal(guide_loc, guide_scale))

    # Generate some data.
    torch.manual_seed(0)
    data = torch.randn(100) + 3.0

    # Because the API in minipyro matches that of Pyro proper,
    # training code works with generic Pyro implementations.
    with pyro_backend(args.backend), MonteCarlo():
        # Construct an SVI object so we can do variational inference on our
        # model/guide pair.
        Elbo = infer.JitTrace_ELBO if args.jit else infer.Trace_ELBO
```

(continues on next page)

```
elbo = Elbo()
adam = optim.Adam({"lr": args.learning_rate})
svi = infer.SVI(model, guide, adam, elbo)

# Basic training loop
pyro.get_param_store().clear()
for step in range(args.num_steps):
    loss = svi.step(data)
    if args.verbose and step % 100 == 0:
        print(f"step {step} loss = {loss}")

# Report the final values of the variational parameters
# in the guide after training.
if args.verbose:
    for name in pyro.get_param_store():
        value = pyro.param(name).data
        print(f"{name} = {value.detach().cpu().numpy()}")

# For this simple (conjugate) model we know the exact posterior. In
# particular we know that the variational distribution should be
# centered near 3.0. So let's check this explicitly.
assert (pyro.param("guide_loc") - 3.0).abs() < 0.1

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Minipyro demo")
    parser.add_argument("-b", "--backend", default="funsor")
    parser.add_argument("-n", "--num-steps", default=1001, type=int)
    parser.add_argument("-lr", "--learning-rate", default=0.02, type=float)
    parser.add_argument("--jit", action="store_true")
    parser.add_argument("-v", "--verbose", action="store_true")
    args = parser.parse_args()
    main(args)
```

EXAMPLE: PCFG

```
import argparse
import math
from collections import OrderedDict

import torch

import funsor
import funsor.ops as ops
from funsor.delta import Delta
from funsor.domains import Bint
from funsor.tensor import Tensor
from funsor.terms import Number, Stack, Variable

def Uniform(components):
    components = tuple(components)
    size = len(components)
    if size == 1:
        return components[0]
    var = Variable("v", Bint[size])
    return Stack(var.name, components).reduce(ops.logaddexp, var.name) - math.log(size)

# @of_shape(*([Bint[2]] * size))
def model(size, position=0):
    if size == 1:
        name = str(position)
        return Uniform((Delta(name, Number(0, 2)), Delta(name, Number(1, 2))))
    return Uniform(
        model(t, position) + model(size - t, t + position) for t in range(1, size)
    )

def main(args):
    funsor.set_backend("torch")
    torch.manual_seed(args.seed)

    print_ = print if args.verbose else lambda msg: None
    print_("Data:")
    data = torch.distributions.Categorical(torch.ones(2)).sample((args.size,))
```

(continues on next page)

(continued from previous page)

```
assert data.shape == (args.size,)
data = Tensor(data, OrderedDict(i=Bint[args.size]), dtype=2)
print_(data)

print_("Model:")
m = model(args.size)
print_(m.pretty())

print_("Eager log_prob:")
obs = {str(i): data(i) for i in range(args.size)}
log_prob = m(**obs)
print_(log_prob)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="PCFG example")
    parser.add_argument("-s", "--size", default=3, type=int)
    parser.add_argument("--seed", default=0, type=int)
    parser.add_argument("-v", "--verbose", action="store_true")
    args = parser.parse_args()
    main(args)
```

EXAMPLE: BIASED KALMAN FILTER

```
import argparse
import itertools
import math
import os

import pyro.distributions as dist
import torch
import torch.nn as nn
from torch.optim import Adam

import funsor
import funsor.ops as ops
import funsor.torch.distributions as f_dist
from funsor.domains import Reals
from funsor.pyro.convert import dist_to_funsor, funsor_to_mvn
from funsor.tensor import Tensor, Variable

# We use a 2D continuous-time NCV dynamics model throughout.
# See http://webee.technion.ac.il/people/shimkin/Estimation09/ch8\_target.pdf
TIME_STEP = 1.0
NCV_PROCESS_NOISE = torch.tensor(
    [
        [1 / 3, 0.0, 1 / 2, 0.0],
        [0.0, 1 / 3, 0.0, 1 / 2],
        [1 / 2, 0.0, 1.0, 0.0],
        [0.0, 1 / 2, 0.0, 1.0],
    ]
)
NCV_TRANSITION_MATRIX = torch.tensor(
    [
        [1.0, 0.0, 0.0, 0.0],
        [0.0, 1.0, 0.0, 0.0],
        [1.0, 0.0, 1.0, 0.0],
        [0.0, 1.0, 0.0, 1.0],
    ]
)

@torch.no_grad()
def generate_data(num_frames, num_sensors):
```

(continues on next page)

(continued from previous page)

```

"""
Generate data from a damped NCV dynamics model
"""
dt = TIME_STEP
bias_scale = 4.0
obs_noise = 1.0
trans_noise = 0.3

# define dynamics
z = torch.cat([10.0 * torch.randn(2), torch.rand(2)]) # position # velocity
damp = 0.1 # damp the velocities
f = torch.tensor(
    [
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [dt * math.exp(-damp * dt), 0, math.exp(-damp * dt), 0],
        [0, dt * math.exp(-damp * dt), 0, math.exp(-damp * dt)],
    ]
)
trans_dist = dist.MultivariateNormal(
    torch.zeros(4),
    scale_tril=trans_noise * torch.linalg.cholesky(NCV_PROCESS_NOISE),
)

# define biased sensors
sensor_bias = bias_scale * torch.randn(2 * num_sensors)
h = torch.eye(4, 2).unsqueeze(-1).expand(-1, -1, num_sensors).reshape(4, -1)
obs_dist = dist.MultivariateNormal(
    sensor_bias, scale_tril=obs_noise * torch.eye(2 * num_sensors)
)

states = []
observations = []
for t in range(num_frames):
    z = z @ f + trans_dist.sample()
    states.append(z)

    x = z @ h + obs_dist.sample()
    observations.append(x)

states = torch.stack(states)
observations = torch.stack(observations)
assert observations.shape == (num_frames, num_sensors * 2)
return observations, states, sensor_bias

class Model(nn.Module):
    def __init__(self, num_sensors):
        super(Model, self).__init__()
        self.num_sensors = num_sensors

        # learnable params

```

(continues on next page)

(continued from previous page)

```

self.log_bias_scale = nn.Parameter(torch.tensor(0.0))
self.log_obs_noise = nn.Parameter(torch.tensor(0.0))
self.log_trans_noise = nn.Parameter(torch.tensor(0.0))

def forward(self, observations, add_bias=True):
    obs_dim = 2 * self.num_sensors
    bias_scale = self.log_bias_scale.exp()
    obs_noise = self.log_obs_noise.exp()
    trans_noise = self.log_trans_noise.exp()

    # bias distribution
    bias = Variable("bias", Reals[obs_dim])
    assert not torch.isnan(bias_scale), "bias scales was nan"
    bias_dist = dist_to_funsor(
        dist.MultivariateNormal(
            torch.zeros(obs_dim),
            scale_tril=bias_scale * torch.eye(2 * self.num_sensors),
        )
    )(value=bias)

    init_dist = dist.MultivariateNormal(
        torch.zeros(4), scale_tril=100.0 * torch.eye(4)
    )
    self.init = dist_to_funsor(init_dist)(value="state")

    # hidden states
    prev = Variable("prev", Reals[4])
    curr = Variable("curr", Reals[4])
    self.trans_dist = f_dist.MultivariateNormal(
        loc=prev @ NCV_TRANSITION_MATRIX,
        scale_tril=trans_noise * torch.linalg.cholesky(NCV_PROCESS_NOISE),
        value=curr,
    )

    state = Variable("state", Reals[4])
    obs = Variable("obs", Reals[obs_dim])
    observation_matrix = Tensor(
        torch.eye(4, 2)
        .unsqueeze(-1)
        .expand(-1, -1, self.num_sensors)
        .reshape(4, -1)
    )
    assert observation_matrix.output.shape == (
        4,
        obs_dim,
    ), observation_matrix.output.shape
    obs_loc = state @ observation_matrix
    if add_bias:
        obs_loc += bias
    self.observation_dist = f_dist.MultivariateNormal(
        loc=obs_loc, scale_tril=obs_noise * torch.eye(obs_dim), value=obs
    )

```

(continues on next page)

```

logp = bias_dist
curr = "state_init"
logp += self.init(state=curr)
for t, x in enumerate(observations):
    prev, curr = curr, "state_{}".format(t)
    logp += self.trans_dist(prev=prev, curr=curr)
    logp += self.observation_dist(state=curr, obs=x)
    # marginalize out previous state
    logp = logp.reduce(ops.logaddexp, prev)
# marginalize out bias variable
logp = logp.reduce(ops.logaddexp, "bias")

# save posterior over the final state
assert set(logp.inputs) == {"state_{}".format(len(observations) - 1)}
posterior = funsor_to_mvn(logp, ndims=0)

# marginalize out remaining variables
logp = logp.reduce(ops.logaddexp)
assert isinstance(logp, Tensor) and logp.shape == (), logp.pretty()
return logp.data, posterior

def track(args):
    results = {} # keyed on (seed, bias, num_frames)
    for seed in args.seed:
        torch.manual_seed(seed)
        observations, states, sensor_bias = generate_data(
            max(args.num_frames), args.num_sensors
        )
        for bias, num_frames in itertools.product(args.bias, args.num_frames):
            print(
                "tracking with seed={}, bias={}, num_frames={}".format(
                    seed, bias, num_frames
                )
            )
            model = Model(args.num_sensors)
            optim = Adam(model.parameters(), lr=args.lr, betas=(0.5, 0.8))
            losses = []
            for i in range(args.num_epochs):
                optim.zero_grad()
                log_prob, posterior = model(observations[:num_frames], add_bias=bias)
                loss = -log_prob
                loss.backward()
                losses.append(loss.item())
                if i % 10 == 0:
                    print(loss.item())
                optim.step()

            # Collect evaluation metrics.
            final_state_true = states[num_frames - 1]
            assert final_state_true.shape == (4,)

```

(continues on next page)

(continued from previous page)

```

final_pos_true = final_state_true[:2]
final_vel_true = final_state_true[2:]

final_state_est = posterior.loc
assert final_state_est.shape == (4,)
final_pos_est = final_state_est[:2]
final_vel_est = final_state_est[2:]
final_pos_error = float(torch.norm(final_pos_true - final_pos_est))
final_vel_error = float(torch.norm(final_vel_true - final_vel_est))
print("final_pos_error = {}".format(final_pos_error))

results[seed, bias, num_frames] = {
    "args": args,
    "observations": observations[:num_frames],
    "states": states[:num_frames],
    "sensor_bias": sensor_bias,
    "losses": losses,
    "bias_scale": float(model.log_bias_scale.exp()),
    "obs_noise": float(model.log_obs_noise.exp()),
    "trans_noise": float(model.log_trans_noise.exp()),
    "final_state_estimate": posterior,
    "final_pos_error": final_pos_error,
    "final_vel_error": final_vel_error,
}
if args.metrics_filename:
    print("saving output to: {}".format(args.metrics_filename))
    torch.save(results, args.metrics_filename)
return results

def main(args):
    funsor.set_backend("torch")
    if (
        args.force
        or not args.metrics_filename
        or not os.path.exists(args.metrics_filename)
    ):
        # Increase compression threshold for numerical stability.
        with funsor.gaussian.Gaussian.set_compression_threshold(3):
            results = track(args)
    else:
        results = torch.load(args.metrics_filename)

    if args.plot_filename:
        import matplotlib

        matplotlib.use("Agg")
        import numpy as np
        from matplotlib import pyplot

        seeds = set(seed for seed, _, _ in results)
        X = args.num_frames

```

(continues on next page)

(continued from previous page)

```

pyplot.figure(figsize=(5, 1.4), dpi=300)

pos_error = np.array(
    [
        [results[s, 0, f]["final_pos_error"] for s in seeds]
        for f in args.num_frames
    ]
)
mse = (pos_error**2).mean(axis=1)
std = (pos_error**2).std(axis=1) / len(seeds) ** 0.5
pyplot.plot(X, mse**0.5, "k--")
pyplot.fill_between(
    X, (mse - std) ** 0.5, (mse + std) ** 0.5, color="black", alpha=0.15, lw=0
)

pos_error = np.array(
    [
        [results[s, 1, f]["final_pos_error"] for s in seeds]
        for f in args.num_frames
    ]
)
mse = (pos_error**2).mean(axis=1)
std = (pos_error**2).std(axis=1) / len(seeds) ** 0.5
pyplot.plot(X, mse**0.5, "r-")
pyplot.fill_between(
    X, (mse - std) ** 0.5, (mse + std) ** 0.5, color="red", alpha=0.15, lw=0
)

pyplot.ylabel("Position RMSE")
pyplot.xlabel("Track Length")
pyplot.xticks((5, 10, 15, 20, 25, 30))
pyplot.xlim(5, 30)
pyplot.tight_layout(0)
pyplot.savefig(args.plot_filename)

def int_list(args):
    result = []
    for arg in args.split(","):
        if "-" in arg:
            beg, end = map(int, arg.split("-"))
            result.extend(range(beg, 1 + end))
        else:
            result.append(int(arg))
    return result

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Biased Kalman filter")
    parser.add_argument(
        "--seed",
        default="0",

```

(continues on next page)

(continued from previous page)

```
    type=int_list,
    help="random seed, comma delimited for multiple runs",
)
parser.add_argument(
    "--bias",
    default="0,1",
    type=int_list,
    help="whether to model bias, comma delimited for multiple runs",
)
parser.add_argument(
    "-f",
    "--num-frames",
    default="5,10,15,20,25,30",
    type=int_list,
    help="number of sensor frames, comma delimited for multiple runs",
)
parser.add_argument("--num-sensors", default=5, type=int)
parser.add_argument("-n", "--num-epochs", default=50, type=int)
parser.add_argument("--lr", default=0.1, type=float)
parser.add_argument("--metrics-filename", default="", type=str)
parser.add_argument("--plot-filename", default="", type=str)
parser.add_argument("--force", action="store_true")
args = parser.parse_args()
main(args)
```


EXAMPLE: SWITCHING LINEAR DYNAMICAL SYSTEM

```
import argparse

import torch

import funsor
import funsor.ops as ops
import funsor.torch.distributions as dist

def main(args):
    funsor.set_backend("torch")

    # Declare parameters.
    trans_probs = funsor.Tensor(
        torch.tensor([[0.9, 0.1], [0.1, 0.9]], requires_grad=True)
    )
    trans_noise = funsor.Tensor(
        torch.tensor(
            [0.1, 1.0], # low noise component # high noisy component
            requires_grad=True,
        )
    )
    emit_noise = funsor.Tensor(torch.tensor(0.5, requires_grad=True))
    params = [trans_probs.data, trans_noise.data, emit_noise.data]

    # A Gaussian HMM model.
    @funsor.interpretations.moment_matching
    def model(data):
        log_prob = funsor.Number(0.0)

        # s is the discrete latent state,
        # x is the continuous latent state,
        # y is the observed state.
        s_curr = funsor.Tensor(torch.tensor(0), dtype=2)
        x_curr = funsor.Tensor(torch.tensor(0.0))
        for t, y in enumerate(data):
            s_prev = s_curr
            x_prev = x_curr

            # A delayed sample statement.
```

(continues on next page)

(continued from previous page)

```

s_curr = funsor.Variable(f"s_{t}", funsor.Bint[2])
log_prob += dist.Categorical(trans_probs[s_prev], value=s_curr)

# A delayed sample statement.
x_curr = funsor.Variable(f"x_{t}", funsor.Real)
log_prob += dist.Normal(x_prev, trans_noise[s_curr], value=x_curr)

# Marginalize out previous delayed sample statements.
if t > 0:
    log_prob = log_prob.reduce(ops.logaddexp, {s_prev.name, x_prev.name})

# An observe statement.
log_prob += dist.Normal(x_curr, emit_noise, value=y)

log_prob = log_prob.reduce(ops.logaddexp)
return log_prob

# Train model parameters.
torch.manual_seed(0)
data = torch.randn(args.time_steps)
optim = torch.optim.Adam(params, lr=args.learning_rate)
for step in range(args.train_steps):
    optim.zero_grad()
    log_prob = model(data)
    assert not log_prob.inputs, "free variables remain"
    loss = -log_prob.data
    loss.backward()
    optim.step()
    if args.verbose and step % 10 == 0:
        print(f"step {step} loss = {loss.item()}")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Switching linear dynamical system")
    parser.add_argument("-t", "--time-steps", default=10, type=int)
    parser.add_argument("-n", "--train-steps", default=101, type=int)
    parser.add_argument("-lr", "--learning-rate", default=0.01, type=float)
    parser.add_argument("--filter", action="store_true")
    parser.add_argument("-v", "--verbose", action="store_true")
    args = parser.parse_args()
    main(args)

```


EXAMPLE: TALBOT'S METHOD FOR NUMERICAL INVERSION OF THE LAPLACE TRANSFORM

```
import argparse
import math

import torch

import funsor
import funsor.ops as ops
from funsor.adam import Adam
from funsor.domains import Real
from funsor.factory import Bound, Fresh, Has, make_funsor
from funsor.interpretations import StatefulInterpretation
from funsor.tensor import Tensor
from funsor.terms import Funsor, Variable
from funsor.util import get_backend

@make_funsor
def InverseLaplace(
    F: Has[{"s"}], t: Funsor, s: Bound # noqa: F821
) -> Fresh[lambd F: F]:
    """
    Inverse Laplace transform of function  $F(s)$ .

    There is no closed-form solution for arbitrary  $F(s)$ . However, we can
    resort to numerical approximations which we store in new interpretations.
    For example, see Talbot's method below.

    :param F: function of  $s$ .
    :param t: times at which to evaluate the inverse Laplace transformation of  $F$ .
    :param s:  $s$  Variable.
    """
    return None

class Talbot(StatefulInterpretation):
    """
    Talbot's method for numerical inversion of the Laplace transform.
    """
```

(continues on next page)

(continued from previous page)

Reference

Abate, Joseph, and Ward Whitt. "A Unified Framework for Numerically Inverting Laplace Transforms." *INFORMS Journal of Computing*, vol. 18.4 (2006): 408-421. Print. (<http://www.columbia.edu/~ww2040/allpapers.html>)

Implementation here is adapted from the MATLAB implementation of the algorithm by Tucker McClure (2021). Numerical Inverse Laplace Transform (<https://www.mathworks.com/matlabcentral/fileexchange/39035-numerical-inverse-laplace-transform>),
 MATLAB Central File Exchange. Retrieved April 4, 2021.

```
:param num_steps: number of terms to sum for each t.
"""
```

```
def __init__(self, num_steps):
    super().__init__("talbot")
    self.num_steps = num_steps
```

```
@Talbot.register(InverseLaplace, Funsor, Funsor, Variable)
```

```
def talbot(self, F, t, s):
    if get_backend() == "torch":
        import torch

        k = torch.arange(1, self.num_steps)
        delta = torch.zeros(self.num_steps, dtype=torch.complex64)
        delta[0] = 2 * self.num_steps / 5
        delta[1:] = (
            2 * math.pi / 5 * k * (1 / (math.pi / self.num_steps * k).tan() + 1j)
        )

        gamma = torch.zeros(self.num_steps, dtype=torch.complex64)
        gamma[0] = 0.5 * delta[0].exp()
        gamma[1:] = (
            1
            + 1j
            * math.pi
            / self.num_steps
            * k
            * (1 + 1 / (math.pi / self.num_steps * k).tan() ** 2)
            - 1j / (math.pi / self.num_steps * k).tan()
        ) * delta[1:].exp()

        delta = Tensor(delta)["num_steps"]
        gamma = Tensor(gamma)["num_steps"]
        ilt = 0.4 / t * (gamma * F(**{s.name: delta / t})).reduce(ops.add, "num_steps")

        return Tensor(ilt.data.real, ilt.inputs)
    else:
        raise NotImplementedError(f"Unsupported backend {get_backend()}")
```

(continues on next page)

(continued from previous page)

```

def main(args):
    """
    Reference for the n-step sequential model used here:

    Aaron L. Lucius et al (2003).
    "General Methods for Analysis of Sequential "n-step" Kinetic Mechanisms:
    Application to Single Turnover Kinetics of Helicase-Catalyzed DNA Unwinding"
    https://www.sciencedirect.com/science/article/pii/S0006349503746487
    """

    funsor.set_backend("torch")

    # Problem definition.
    true_rate = 20
    true_nsteps = 4
    rate = Variable("rate", Real)
    nsteps = Variable("nsteps", Real)
    s = Variable("s", Real)
    time = Tensor(torch.arange(0.04, 1.04, 0.04))["timepoint"]
    noise = Tensor(torch.randn(time.inputs["timepoint"].size))["timepoint"] / 500
    data = (
        Tensor(1 - torch.igammac(torch.tensor(true_nsteps), true_rate * time.data))["timepoint"]
    )
    + noise
)
F = rate**nsteps / (s * (rate + s) ** nsteps)
# Inverse Laplace.
pred = InverseLaplace(F, time, "s")

# Loss function.
loss = (pred - data).abs().reduce(ops.add, "timepoint")
init_params = {
    "rate": Tensor(torch.tensor(5.0, requires_grad=True)),
    "nsteps": Tensor(torch.tensor(2.0, requires_grad=True)),
}

with Talbot(num_steps=args.talbot_num_steps):
    # Fit the data
    with Adam(
        args.num_steps,
        lr=args.learning_rate,
        log_every=args.log_every,
        params=init_params,
    ) as optim:
        loss.reduce(ops.min, {"rate", "nsteps"})
    # Fitted curve.
    fitted_curve = pred(rate=optim.param("rate"), nsteps=optim.param("nsteps"))

print(f>Data\n{data}")
print(f"Fit curve\n{fitted_curve}")
print(f"True rate\n{true_rate}")

```

(continues on next page)

(continued from previous page)

```
print("Learned rate\n{}".format(optim.param("rate").item()))
print(f"True number of steps\n{true_nsteps}")
print("Learned number of steps\n{}".format(optim.param("nsteps").item()))

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Numerical inversion of the Laplace transform using Talbot's method"
    )
    parser.add_argument("-N", "--talbot-num-steps", type=int, default=32)
    parser.add_argument("-n", "--num-steps", type=int, default=501)
    parser.add_argument("-lr", "--learning-rate", type=float, default=0.1)
    parser.add_argument("--log-every", type=int, default=20)
    args = parser.parse_args()
    main(args)
```

EXAMPLE: VAE MNIST

```
import argparse
import os
import typing
from collections import OrderedDict

import torch
import torch.utils.data
from torch import nn, optim
from torch.nn import functional as F
from torchvision import transforms
from torchvision.datasets import MNIST

import funsor
import funsor.ops as ops
import funsor.torch.distributions as dist
from funsor.domains import Bint, Reals

REPO_PATH = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
DATA_PATH = os.path.join(REPO_PATH, "data")

class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)

    def forward(self, image: Reals[28, 28]) -> typing.Tuple[Reals[20], Reals[20]]:
        image = image.reshape(image.shape[:-2] + (-1,))
        h1 = F.relu(self.fc1(image))
        loc = self.fc21(h1)
        scale = self.fc22(h1).exp()
        return loc, scale

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.fc3 = nn.Linear(20, 400)
```

(continues on next page)

(continued from previous page)

```

        self.fc4 = nn.Linear(400, 784)

    def forward(self, z: Reals[20]) -> Reals[28, 28]:
        h3 = F.relu(self.fc3(z))
        out = torch.sigmoid(self.fc4(h3))
        return out.reshape(out.shape[:-1] + (28, 28))

def main(args):
    funsor.set_backend("torch")

    # XXX Temporary fix after https://github.com/pyro-ppl/pyro/pull/2701
    import pyro

    pyro.enable_validation(False)

    encoder = Encoder()
    decoder = Decoder()

    # These rely on type hints on the .forward() methods.
    encode = funsor.function(encoder)
    decode = funsor.function(decoder)

    @funsor.montecarlo.MonteCarlo()
    def loss_function(data, subsample_scale):
        # Lazily sample from the guide.
        loc, scale = encode(data)
        q = funsor.Independent(
            dist.Normal(loc["i"], scale["i"], value="z_i"), "z", "i", "z_i"
        )

        # Evaluate the model likelihood at the lazy value z.
        probs = decode("z")
        p = dist.Bernoulli(probs["x", "y"], value=data["x", "y"])
        p = p.reduce(ops.add, {"x", "y"})

        # Construct an elbo. This is where sampling happens.
        elbo = funsor.Integrate(q, p - q, "z")
        elbo = elbo.reduce(ops.add, "batch") * subsample_scale
        loss = -elbo
        return loss

    train_loader = torch.utils.data.DataLoader(
        MNIST(DATA_PATH, train=True, download=True, transform=transforms.ToTensor()),
        batch_size=args.batch_size,
        shuffle=True,
    )

    encoder.train()
    decoder.train()
    optimizer = optim.Adam(
        list(encoder.parameters()) + list(decoder.parameters()), lr=1e-3

```

(continues on next page)

(continued from previous page)

```
)
for epoch in range(args.num_epochs):
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        subsample_scale = float(len(train_loader.dataset) / len(data))
        data = data[:, 0, :, :]
        data = funsor.Tensor(data, OrderedDict(batch= Bint[len(data)]))

        optimizer.zero_grad()
        loss = loss_function(data, subsample_scale)
        assert isinstance(loss, funsor.Tensor), loss.pretty()
        loss.data.backward()
        train_loss += loss.item()
        optimizer.step()
        if batch_idx % 50 == 0:
            print(f" loss = {loss.item()}")
            if batch_idx and args.smoke_test:
                return
    print(f"epoch {epoch} train_loss = {train_loss}")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="VAE MNIST Example")
    parser.add_argument("-n", "--num-epochs", type=int, default=10)
    parser.add_argument("--batch-size", type=int, default=8)
    parser.add_argument("--smoke-test", action="store_true")
    args = parser.parse_args()
    main(args)
```


INDICES AND TABLES

- [genindex](#)
- [modindex](#)
- [search](#)

PYTHON MODULE INDEX

f

- `functor.adjoint`, 35
- `functor.affine`, 41
- `functor.approximations`, 12
- `functor.cnf`, 30
- `functor.compiler`, 71
- `functor.constant`, 31
- `functor.delta`, 23
- `functor.domains`, 7
- `functor.einsum`, 69
- `functor.elbo`, 13
- `functor.factory`, 43
- `functor.gaussian`, 28
- `functor.integrate`, 31
- `functor.interpretations`, 9
- `functor.interpreter`, 9
- `functor.joint`, 29
- `functor.minipyro`, 65
- `functor.montecarlo`, 11
- `functor.ops.array`, 5
- `functor.ops.builtin`, 3
- `functor.ops.op`, 1
- `functor.ops.program`, 71
- `functor.ops.tracer`, 71
- `functor.optimizer`, 33
- `functor.precondition`, 11
- `functor.pyro.convert`, 58
- `functor.pyro.distribution`, 53
- `functor.pyro.hmm`, 54
- `functor.recipes`, 49
- `functor.sum_product`, 37
- `functor.tensor`, 24
- `functor.terms`, 15
- `functor.testing`, 45
- `functor.torch.distributions`, 61
- `functor.typing`, 47

A

abs (in module *funsor.ops.builtin*), 3
 abs() (*Funsor* method), 17
 ActualExpected (class in *funsor.testing*), 45
 Adam (class in *funsor.minipyro*), 66
 add (in module *funsor.ops.builtin*), 3
 adjoint() (*AdjointTape* method), 35
 adjoint() (in module *funsor.adjoint*), 35
 adjoint_binary() (in module *funsor.adjoint*), 35
 adjoint_cat() (in module *funsor.adjoint*), 35
 adjoint_contract() (in module *funsor.adjoint*), 35
 adjoint_contract_generic() (in module *funsor.adjoint*), 35
 adjoint_contract_unary() (in module *funsor.adjoint*), 35
 adjoint_reduce() (in module *funsor.adjoint*), 35
 adjoint_scatter() (in module *funsor.adjoint*), 35
 adjoint_subs() (in module *funsor.adjoint*), 35
 AdjointTape (class in *funsor.adjoint*), 35
 affine_inputs() (in module *funsor.affine*), 41
 align() (*Constant* method), 31
 align() (*Contraction* method), 30
 align() (*Delta* method), 24
 align() (*Funsor* method), 17
 align() (*Gaussian* method), 29
 align() (*Tensor* method), 25
 align_gaussian() (in module *funsor.gaussian*), 29
 align_tensor() (in module *funsor.tensor*), 26
 align_tensors() (in module *funsor.tensor*), 26
 all (in module *funsor.ops.array*), 5
 all() (*Funsor* method), 18
 amax (in module *funsor.ops.array*), 5
 amin (in module *funsor.ops.array*), 5
 and_ (in module *funsor.ops.builtin*), 3
 any (in module *funsor.ops.array*), 5
 any() (*Funsor* method), 18
 apply_optimizer() (in module *funsor.optimizer*), 33
 apply_stack() (in module *funsor.minipyro*), 66
 Approximate (class in *funsor.terms*), 15
 approximate() (*Funsor* method), 16
 arg_constraints (*FunsorDistribution* attribute), 53
 arg_constraints (*GaussianHMM* attribute), 55

arg_constraints (*SwitchingLinearHMM* attribute), 57
 argmax (in module *funsor.ops.array*), 5
 argmax() (*Funsor* method), 18
 argmax_approximate (in module *funsor.approximations*), 12
 argmin (in module *funsor.ops.array*), 5
 argmin() (*Funsor* method), 18
 arity (*BinaryOp* attribute), 1
 arity (*FinitaryOp* attribute), 1
 arity (*NullaryOp* attribute), 1
 arity (*Op* attribute), 2
 arity (*TernaryOp* attribute), 2
 arity (*UnaryOp* attribute), 2
 as_code() (*OpProgram* method), 72
 as_tensor() (*BlockMatrix* method), 28
 as_tensor() (*BlockVector* method), 28
 assert_close() (in module *funsor.testing*), 45
 assert_equiv() (in module *funsor.testing*), 45
 astype (in module *funsor.ops.array*), 5
 atanh (in module *funsor.ops.builtin*), 3
 atanh() (*Funsor* method), 17

B

BernoulliLogits (class in *funsor.torch.distributions*), 62
 BernoulliProbs (class in *funsor.torch.distributions*), 61
 Beta (class in *funsor.torch.distributions*), 61
 Binary (class in *funsor.terms*), 15
 binary_divide() (in module *funsor.cnf*), 30
 binary_subtract() (in module *funsor.cnf*), 30
 binary_to_contract() (in module *funsor.cnf*), 30
 BinaryOp (class in *funsor.ops.op*), 1
 Binomial (class in *funsor.torch.distributions*), 62
 Bint (class in *funsor.domains*), 7
 bint() (in module *funsor.domains*), 8
 BintType (class in *funsor.domains*), 7
 block (class in *funsor.minipyro*), 65
 BlockMatrix (class in *funsor.gaussian*), 28
 BlockVector (class in *funsor.gaussian*), 28
 Bound (class in *funsor.factory*), 43

C

CallableInterpretation (class in *funsor.interpretations*), 9
 Cat (class in *funsor.terms*), 15
 cat (in module *funsor.ops.array*), 5
 Categorical (class in *funsor.torch.distributions*), 62
 CategoricalLogits (class in *funsor.torch.distributions*), 62
 Cauchy (class in *funsor.torch.distributions*), 61
 check_funsor() (in module *funsor.testing*), 45
 Chi2 (class in *funsor.torch.distributions*), 61
 children_contraction() (in module *funsor.cnf*), 30
 cholesky (in module *funsor.ops.array*), 5
 cholesky_inverse (in module *funsor.ops.array*), 5
 cholesky_solve (in module *funsor.ops.array*), 5
 clamp (in module *funsor.ops.array*), 5
 clamp_finite() (Tensor method), 25
 ClippedAdam (class in *funsor.minipyro*), 66
 combine_subs() (Precondition method), 12
 compression_threshold (Gaussian attribute), 29
 compute_argmax() (in module *funsor.approximations*), 12
 CondIndepStackFrame (class in *funsor.minipyro*), 66
 Constant (class in *funsor.constant*), 31
 ConstantMeta (class in *funsor.constant*), 31
 Contraction (class in *funsor.cnf*), 30

D

declare_op_types() (in module *funsor.ops.op*), 3
 deep_isinstance() (in module *funsor.typing*), 47
 deep_issubclass() (in module *funsor.typing*), 47
 deep_type() (in module *funsor.typing*), 48
 default() (LogAbsDetJacobianOp static method), 1
 default() (WrappedTransformOp static method), 2
 Delta (class in *funsor.delta*), 23
 Delta (class in *funsor.torch.distributions*), 62
 Dependent (class in *funsor.domains*), 7
 DesugarGetitem (class in *funsor.testing*), 46
 detach (in module *funsor.ops.array*), 5
 diagonal (in module *funsor.ops.array*), 5
 dim (CondIndepStackFrame property), 66
 Dirichlet (class in *funsor.torch.distributions*), 62
 DirichletMultinomial (class in *funsor.torch.distributions*), 62
 DiscreteHMM (class in *funsor.pyro.hmm*), 54
 DispatchedInterpretation (class in *funsor.interpretations*), 10
 dispatcher (LogAbsDetJacobianOp attribute), 1
 dispatcher (WrappedTransformOp attribute), 3
 dist_class (BernoulliLogits attribute), 62
 dist_class (BernoulliProbs attribute), 62
 dist_class (Beta attribute), 61
 dist_class (Binomial attribute), 62
 dist_class (Categorical attribute), 62

dist_class (CategoricalLogits attribute), 62
 dist_class (Cauchy attribute), 61
 dist_class (Chi2 attribute), 61
 dist_class (Delta attribute), 62
 dist_class (Dirichlet attribute), 62
 dist_class (DirichletMultinomial attribute), 62
 dist_class (Distribution attribute), 61
 dist_class (Exponential attribute), 62
 dist_class (Gamma attribute), 62
 dist_class (GammaPoisson attribute), 63
 dist_class (Geometric attribute), 63
 dist_class (Gumbel attribute), 63
 dist_class (HalfCauchy attribute), 63
 dist_class (HalfNormal attribute), 63
 dist_class (Laplace attribute), 63
 dist_class (Logistic attribute), 63
 dist_class (LowRankMultivariateNormal attribute), 63
 dist_class (Multinomial attribute), 63
 dist_class (MultivariateNormal attribute), 63
 dist_class (NonreparameterizedBeta attribute), 64
 dist_class (NonreparameterizedDirichlet attribute), 64
 dist_class (NonreparameterizedGamma attribute), 64
 dist_class (NonreparameterizedNormal attribute), 64
 dist_class (Normal attribute), 64
 dist_class (Pareto attribute), 64
 dist_class (Poisson attribute), 64
 dist_class (StudentT attribute), 64
 dist_class (Uniform attribute), 64
 dist_class (VonMises attribute), 64
 dist_to_funsor() (in module *funsor.pyro.convert*), 58
 distribute_subs_contraction() (in module *funsor.cnf*), 30
 Distribution (class in *funsor.distribution*), 61
 Distribution (class in *funsor.minipyro*), 65
 do_fresh_subs() (in module *funsor.cnf*), 30
 Domain (in module *funsor.domains*), 7
 dtype (Bint attribute), 7
 dtype (Funsor property), 16
 dtype (RealsType attribute), 8
 dynamic_partial_sum_product() (in module *funsor.sum_product*), 37

E

eager (in module *funsor.interpretations*), 11
 eager_binary_constant_constant() (in module *funsor.constant*), 32
 eager_binary_constant_tensor() (in module *funsor.constant*), 32
 eager_binary_tensor_constant() (in module *funsor.constant*), 32
 eager_contract_base() (in module *funsor.optimizer*), 33
 eager_contraction_gaussian() (in module *funsor.cnf*), 30

- eager_contraction_generic_recursive() (in module *funsor.cnf*), 30
- eager_contraction_generic_to_tuple() (in module *funsor.cnf*), 30
- eager_contraction_tensor() (in module *funsor.cnf*), 30
- eager_contraction_to_binary() (in module *funsor.cnf*), 30
- eager_contraction_to_reduce() (in module *funsor.cnf*), 30
- eager_independent_joint() (in module *funsor.joint*), 29
- eager_log_prob() (Distribution class method), 61
- eager_markov_product() (in module *funsor.sum_product*), 40
- eager_reduce() (Constant method), 31
- eager_reduce() (Delta method), 24
- eager_reduce() (Distribution method), 61
- eager_reduce() (Funsor method), 17
- eager_reduce() (Gaussian method), 29
- eager_reduce() (Stack method), 20
- eager_reduce() (Tensor method), 25
- eager_reduce_add() (in module *funsor.constant*), 32
- eager_reduce_exp() (in module *funsor.joint*), 29
- eager_subs() (Cat method), 15
- eager_subs() (Constant method), 31
- eager_subs() (Delta method), 24
- eager_subs() (Funsor method), 17
- eager_subs() (Gaussian method), 29
- eager_subs() (Independent method), 18
- eager_subs() (MarkovProduct method), 40
- eager_subs() (Scatter method), 20
- eager_subs() (Slice method), 20
- eager_subs() (Stack method), 20
- eager_subs() (Tensor method), 25
- eager_subs() (Variable method), 21
- eager_unary() (Funsor method), 17
- eager_unary() (in module *funsor.constant*), 32
- eager_unary() (Number method), 19
- eager_unary() (Tensor method), 25
- einsum (in module *funsor.ops.array*), 5
- einsum() (in module *funsor.einsum*), 69
- Einsum() (in module *funsor.tensor*), 24
- Elbo (class in *funsor.elbo*), 13
- ELBO (class in *funsor.minipyro*), 67
- elbo() (in module *funsor.minipyro*), 67
- empty() (in module *funsor.testing*), 45
- entropy() (Distribution method), 61
- entropy() (Independent method), 19
- enumerate_support() (Distribution method), 61
- eq (in module *funsor.ops.builtin*), 3
- excludes_backend() (in module *funsor.testing*), 45
- exp (in module *funsor.ops.builtin*), 3
- exp() (Funsor method), 17
- expand (in module *funsor.ops.array*), 5
- expand() (*DiscreteHMM* method), 54
- expand() (*FunsorDistribution* method), 53
- expand() (*SwitchingLinearHMM* method), 57
- expand_inputs() (Distribution method), 65
- Expectation() (in module *funsor.minipyro*), 66
- Exponential (class in *funsor.torch.distributions*), 62
- extract_affine() (in module *funsor.affine*), 41
- ## F
- filter() (*SwitchingLinearHMM* method), 57
- find_domain() (in module *funsor.domains*), 8
- finfo (in module *funsor.ops.array*), 5
- FinitaryOp (class in *funsor.ops.op*), 1
- flip (in module *funsor.ops.array*), 5
- floordiv (in module *funsor.ops.builtin*), 3
- forward_backward() (in module *funsor.adjoint*), 35
- forward_filter_backward_precondition() (in module *funsor.recipes*), 51
- forward_filter_backward_rsampling() (in module *funsor.recipes*), 51
- Fresh (class in *funsor.factory*), 43
- full_like (in module *funsor.ops.array*), 5
- Function (class in *funsor.tensor*), 24
- function() (in module *funsor.tensor*), 26
- Funsor (class in *funsor.terms*), 15
- funsor.adjoint
module, 35
- funsor.affine
module, 41
- funsor.approximations
module, 12
- funsor.cnf
module, 30
- funsor.compiler
module, 71
- funsor.constant
module, 31
- funsor.delta
module, 23
- funsor.domains
module, 7
- funsor.einsum
module, 69
- funsor.elbo
module, 13
- funsor.factory
module, 43
- funsor.gaussian
module, 28
- funsor.integrate
module, 31
- funsor.interpretations
module, 9

- funsor.interpreter
 - module, 9
 - funsor.joint
 - module, 29
 - funsor.minipyro
 - module, 65
 - funsor.montecarlo
 - module, 11
 - funsor.ops.array
 - module, 5
 - funsor.ops.builtin
 - module, 3
 - funsor.ops.op
 - module, 1
 - funsor.ops.program
 - module, 71
 - funsor.ops.tracer
 - module, 71
 - funsor.optimizer
 - module, 33
 - funsor.precondition
 - module, 11
 - funsor.pyro.convert
 - module, 58
 - funsor.pyro.distribution
 - module, 53
 - funsor.pyro.hmm
 - module, 54
 - funsor.recipes
 - module, 49
 - funsor.sum_product
 - module, 37
 - funsor.tensor
 - module, 24
 - funsor.terms
 - module, 15
 - funsor.testing
 - module, 45
 - funsor.torch.distributions
 - module, 61
 - funsor.typing
 - module, 47
 - funsor_to_cat_and_mvn() (in module *funsor.pyro.convert*), 59
 - funsor_to_mvn() (in module *funsor.pyro.convert*), 59
 - funsor_to_tensor() (in module *funsor.pyro.convert*), 58
 - FunsorDistribution (class in *funsor.pyro.distribution*), 53
 - funsordistribution_to_funsor() (in module *funsor.pyro.distribution*), 53
- G**
- Gamma (class in *funsor.torch.distributions*), 62
 - GammaPoisson (class in *funsor.torch.distributions*), 62
 - Gaussian (class in *funsor.gaussian*), 28
 - GaussianHMM (class in *funsor.pyro.hmm*), 54
 - GaussianMixture (in module *funsor.cnf*), 30
 - GaussianMRF (class in *funsor.pyro.hmm*), 55
 - ge (in module *funsor.ops.builtin*), 3
 - GenericTypeMeta (class in *funsor.typing*), 47
 - Geometric (class in *funsor.torch.distributions*), 63
 - get_args() (in module *funsor.typing*), 48
 - get_origin() (in module *funsor.typing*), 48
 - get_param_store() (in module *funsor.minipyro*), 65
 - get_trace() (trace method), 65
 - get_type_hints() (in module *funsor.typing*), 48
 - getitem (in module *funsor.ops.builtin*), 3
 - getslice (in module *funsor.ops.builtin*), 3
 - gt (in module *funsor.ops.builtin*), 3
 - Gumbel (class in *funsor.torch.distributions*), 63
- H**
- HalfCauchy (class in *funsor.torch.distributions*), 63
 - HalfNormal (class in *funsor.torch.distributions*), 63
 - Has (class in *funsor.factory*), 43
 - has_enumerate_support (Distribution property), 61
 - has_rsampl (DiscreteHMM property), 54
 - has_rsampl (GaussianHMM attribute), 55
 - has_rsampl (GaussianMRF attribute), 56
 - has_rsampl (SwitchingLinearHMM attribute), 57
- I**
- id_from_inputs() (in module *funsor.testing*), 45
 - ignore_jit_warnings() (in module *funsor.tensor*), 27
 - Independent (class in *funsor.terms*), 18
 - input_vars() (Funsor method), 16
 - Integrate (class in *funsor.integrate*), 31
 - interpret() (AdjointTape method), 35
 - Interpretation (class in *funsor.interpretations*), 10
 - interpretation() (in module *funsor.interpreter*), 9
 - inv (WrappedTransformOp property), 3
 - inv() (TransformOp static method), 2
 - invert (in module *funsor.ops.builtin*), 3
 - is_affine() (in module *funsor.affine*), 41
 - is_array() (in module *funsor.testing*), 45
 - is_full_rank (Gaussian property), 29
 - isnan (in module *funsor.ops.array*), 5
 - item() (Funsor method), 16
 - item() (Number method), 19
 - item() (Tensor method), 25
 - iter_subsets() (in module *funsor.testing*), 46
- J**
- Jit (class in *funsor.minipyro*), 67
 - Jit_ELBO (class in *funsor.minipyro*), 67
 - JitTrace_ELBO() (in module *funsor.minipyro*), 67

- JitTraceEnum_ELBO() (in module *funsor.minipyro*), 67
- JitTraceMeanField_ELBO() (in module *funsor.minipyro*), 67
- ## L
- Lambda (class in *funsor.terms*), 19
- Laplace (class in *funsor.torch.distributions*), 63
- laplace_approximate (in module *funsor.approximations*), 12
- lazy (in module *funsor.interpretations*), 11
- le (in module *funsor.ops.builtin*), 3
- lgamma (in module *funsor.ops.builtin*), 3
- log (in module *funsor.ops.builtin*), 4
- log() (Funsor method), 17
- log1p (in module *funsor.ops.builtin*), 4
- log1p() (Funsor method), 17
- log_abs_det_jacobian (WrappedTransformOp property), 3
- log_abs_det_jacobian() (TransformOp static method), 2
- log_joint (class in *funsor.minipyro*), 66
- log_normalizer() (Gaussian method), 29
- log_prob() (DiscreteHMM method), 54
- log_prob() (Distribution method), 65
- log_prob() (FunsorDistribution method), 53
- log_prob() (SwitchingLinearHMM method), 57
- LogAbsDetJacobianOp (class in *funsor.ops.op*), 1
- logaddexp (in module *funsor.ops.array*), 6
- Logistic (class in *funsor.torch.distributions*), 63
- logsumexp (in module *funsor.ops.array*), 6
- logsumexp() (Funsor method), 18
- lower() (in module *funsor.compiler*), 71
- LowRankMultivariateNormal (class in *funsor.torch.distributions*), 63
- lshift (in module *funsor.ops.builtin*), 4
- lt (in module *funsor.ops.builtin*), 4
- ## M
- make() (Op class method), 2
- make_chain_einsum() (in module *funsor.testing*), 46
- make_einsum_example() (in module *funsor.testing*), 45
- make_funsor() (in module *funsor.factory*), 44
- make_hmm_einsum() (in module *funsor.testing*), 46
- make_plated_hmm_einsum() (in module *funsor.testing*), 46
- MarkovProduct (class in *funsor.sum_product*), 39
- MarkovProductMeta (class in *funsor.sum_product*), 39
- materialize() (Constant method), 31
- materialize() (Tensor method), 26
- matmul (in module *funsor.ops.builtin*), 4
- matrix_and_mvn_to_funsor() (in module *funsor.pyro.convert*), 59
- max (in module *funsor.ops.builtin*), 4
- max() (Funsor method), 18
- mean (in module *funsor.ops.array*), 6
- mean() (Distribution method), 61
- mean() (Funsor method), 18
- mean() (Independent method), 18
- mean_approximate (in module *funsor.approximations*), 12
- Memoize (class in *funsor.interpretations*), 10
- memoize() (in module *funsor.interpretations*), 11
- Messenger (class in *funsor.minipyro*), 65
- min (in module *funsor.ops.builtin*), 4
- min() (Funsor method), 18
- mixed_sequential_sum_product() (in module *funsor.sum_product*), 39
- mod (in module *funsor.ops.builtin*), 4
- modified_partial_sum_product() (in module *funsor.sum_product*), 38
- module
- funsor.adjoint*, 35
 - funsor.affine*, 41
 - funsor.approximations*, 12
 - funsor.cnf*, 30
 - funsor.compiler*, 71
 - funsor.constant*, 31
 - funsor.delta*, 23
 - funsor.domains*, 7
 - funsor.einsum*, 69
 - funsor.elbo*, 13
 - funsor.factory*, 43
 - funsor.gaussian*, 28
 - funsor.integrate*, 31
 - funsor.interpretations*, 9
 - funsor.interpreter*, 9
 - funsor.joint*, 29
 - funsor.minipyro*, 65
 - funsor.montecarlo*, 11
 - funsor.ops.array*, 5
 - funsor.ops.builtin*, 3
 - funsor.ops.op*, 1
 - funsor.ops.program*, 71
 - funsor.ops.tracer*, 71
 - funsor.optimizer*, 33
 - funsor.precondition*, 11
 - funsor.pyro.convert*, 58
 - funsor.pyro.distribution*, 53
 - funsor.pyro.hmm*, 54
 - funsor.recipes*, 49
 - funsor.sum_product*, 37
 - funsor.tensor*, 24
 - funsor.terms*, 15
 - funsor.testing*, 45
 - funsor.torch.distributions*, 61
 - funsor.typing*, 47
- moment_matching (in module *funsor.interpretations*), 11

moment_matching_contract_default() (in module *funsor.joint*), 29
moment_matching_contract_joint() (in module *funsor.joint*), 29
moment_matching_reduce() (Funsor method), 17
MonteCarlo (class in *funsor.montecarlo*), 11
mul (in module *funsor.ops.builtin*), 4
Multinomial (class in *funsor.torch.distributions*), 63
MultivariateNormal (class in *funsor.torch.distributions*), 63
mvn_to_funsor() (in module *funsor.pyro.convert*), 59

N

naive_contract_einsum() (in module *funsor.einsum*), 69
naive_einsum() (in module *funsor.einsum*), 69
naive_plated_einsum() (in module *funsor.einsum*), 69
naive_sarkka_bilmes_product() (in module *funsor.sum_product*), 39
naive_sequential_sum_product() (in module *funsor.sum_product*), 38
name (CondIndepStackFrame property), 66
name (LogAbsDet.JacobianOp attribute), 1
name (WrappedTransformOp attribute), 3
ne (in module *funsor.ops.builtin*), 4
neg (in module *funsor.ops.builtin*), 4
new_arange (in module *funsor.ops.array*), 6
new_arange() (Tensor method), 25
new_eye (in module *funsor.ops.array*), 6
new_full (in module *funsor.ops.array*), 6
new_zeros (in module *funsor.ops.array*), 6
NonreparameterizedBeta (class in *funsor.torch.distributions*), 63
NonreparameterizedDirichlet (class in *funsor.torch.distributions*), 64
NonreparameterizedGamma (class in *funsor.torch.distributions*), 64
NonreparameterizedNormal (class in *funsor.torch.distributions*), 64
Normal (class in *funsor.torch.distributions*), 64
normalize (in module *funsor.interpretations*), 11
normalize_contraction_commutative_canonical_order() (in module *funsor.cnf*), 30
normalize_contraction_commute_joint() (in module *funsor.cnf*), 30
normalize_contraction_generic_args() (in module *funsor.cnf*), 30
normalize_contraction_generic_tuple() (in module *funsor.cnf*), 30
normalize_fuse_subs() (in module *funsor.cnf*), 30
normalize_trivial() (in module *funsor.cnf*), 30
null (in module *funsor.ops.builtin*), 4
NullaryOp (class in *funsor.ops.op*), 1
Number (class in *funsor.terms*), 19

O

of_shape() (in module *funsor.terms*), 21
ones() (in module *funsor.testing*), 45
Op (class in *funsor.ops.op*), 1
OpProgram (class in *funsor.ops.program*), 71
optimize_contract_finitary_funsor() (in module *funsor.optimizer*), 33
optimize_contraction_variadic() (in module *funsor.optimizer*), 33
or_ (in module *funsor.ops.builtin*), 4

P

param() (in module *funsor.minipyro*), 66
Pareto (class in *funsor.torch.distributions*), 64
partial_sum_product() (in module *funsor.sum_product*), 37
partial_unroll() (in module *funsor.sum_product*), 37
PatternMissingError, 9
permute (in module *funsor.ops.array*), 6
plate() (in module *funsor.minipyro*), 66
PlateMessenger (class in *funsor.minipyro*), 66
Poisson (class in *funsor.torch.distributions*), 64
pop_interpretation() (in module *funsor.interpreter*), 9
pos (in module *funsor.ops.builtin*), 4
postprocess_message() (log_joint method), 66
postprocess_message() (Messenger method), 65
postprocess_message() (trace method), 65
pow (in module *funsor.ops.builtin*), 4
Precondition (class in *funsor.precondition*), 11
pretty() (Funsor method), 16
process_message() (block method), 65
process_message() (log_joint method), 66
process_message() (Messenger method), 65
process_message() (PlateMessenger method), 66
process_message() (replay method), 65
prod (in module *funsor.ops.array*), 6
prod() (Funsor method), 18
push_interpretation() (in module *funsor.interpreter*), 9

PyroOptim (class in *funsor.minipyro*), 66

Q

qr (in module *funsor.ops.array*), 6
quote() (Funsor method), 16

R

rand() (in module *funsor.testing*), 45
randint() (in module *funsor.testing*), 45
randn (in module *funsor.ops.array*), 6
randn() (in module *funsor.testing*), 45
random_gaussian() (in module *funsor.testing*), 45
random_mvn() (in module *funsor.testing*), 46

- random_scale_tril() (in module *funsor.testing*), 45
 random_tensor() (in module *funsor.testing*), 45
 rank (Gaussian property), 29
 Real (class in *funsor.domains*), 7
 Reals (class in *funsor.domains*), 7
 reals() (in module *funsor.domains*), 8
 RealsType (class in *funsor.domains*), 8
 reciprocal (in module *funsor.ops.builtin*), 4
 Reduce (class in *funsor.terms*), 19
 reduce() (Funsor method), 16
 reduce_funsor() (in module *funsor.cnf*), 30
 register() (Op method), 2
 register_subclasscheck() (in module *funsor.typing*), 48
 reinterpret() (in module *funsor.interpreter*), 9
 replay (class in *funsor.minipyro*), 65
 requires_backend() (in module *funsor.testing*), 45
 requires_grad (Funsor property), 16
 requires_grad (Tensor property), 25
 reshape() (Funsor method), 17
 rsample() (FunsorDistribution method), 53
 rshift (in module *funsor.ops.builtin*), 4
- ## S
- safediv (in module *funsor.ops.builtin*), 4
 safesub (in module *funsor.ops.builtin*), 4
 sample (in module *funsor.ops.array*), 6
 sample() (Funsor method), 16
 sample() (FunsorDistribution method), 53
 sample() (in module *funsor.minipyro*), 66
 sarkka_bilmes_product() (in module *funsor.sum_product*), 39
 Scatter (class in *funsor.terms*), 19
 scatter (in module *funsor.ops.array*), 6
 scatter_add (in module *funsor.ops.array*), 6
 seed (class in *funsor.minipyro*), 66
 sequential (in module *funsor.interpretations*), 11
 sequential_reduce() (Funsor method), 17
 sequential_sum_product() (in module *funsor.sum_product*), 38
 set_callable() (CallableInterpretation method), 9
 set_compression_threshold() (Gaussian class method), 29
 set_inv() (TransformOp method), 2
 set_log_abs_det_jacobian() (TransformOp method), 2
 shape (Bint attribute), 7
 shape (Funsor property), 16
 shape (Real attribute), 7
 shape (Reals attribute), 8
 sigmoid (in module *funsor.ops.builtin*), 4
 sigmoid() (Funsor method), 17
 signature (LogAbsDetJacobianOp attribute), 1
 signature (WrappedTransformOp attribute), 3
 size (BintType property), 7
 size (CondIndepStackFrame property), 66
 Slice (class in *funsor.terms*), 20
 solve() (in module *funsor.delta*), 24
 sqrt (in module *funsor.ops.builtin*), 4
 sqrt() (Funsor method), 17
 Stack (class in *funsor.terms*), 20
 stack (in module *funsor.ops.array*), 6
 StatefulInterpretation (class in *funsor.interpretations*), 10
 std (in module *funsor.ops.array*), 6
 std() (Funsor method), 18
 step() (SVI method), 66
 StudentT (class in *funsor.torch.distributions*), 64
 sub (in module *funsor.ops.builtin*), 5
 subclass_register() (Op class method), 2
 Subs (class in *funsor.terms*), 20
 sum (in module *funsor.ops.array*), 6
 sum() (Funsor method), 18
 sum_product() (in module *funsor.sum_product*), 38
 support (FunsorDistribution property), 53
 SVI (class in *funsor.minipyro*), 66
 SwitchingLinearHMM (class in *funsor.pyro.hmm*), 56
- ## T
- tanh (in module *funsor.ops.builtin*), 5
 tanh() (Funsor method), 17
 Tensor (class in *funsor.tensor*), 25
 tensor_to_funsor() (in module *funsor.minipyro*), 66
 tensor_to_funsor() (in module *funsor.pyro.convert*), 58
 tensordot() (in module *funsor.tensor*), 27
 TernaryOp (class in *funsor.ops.op*), 2
 to_data() (in module *funsor.terms*), 21
 to_funsor() (in module *funsor.terms*), 22
 TorchOptimizer (Adam attribute), 66
 TorchOptimizer (ClippedAdam attribute), 66
 trace (class in *funsor.minipyro*), 65
 Trace_ELBO (class in *funsor.minipyro*), 67
 trace_function() (in module *funsor.ops.tracer*), 71
 TraceEnum_ELBO (class in *funsor.minipyro*), 67
 TraceMeanField_ELBO (class in *funsor.minipyro*), 67
 TransformOp (class in *funsor.ops.op*), 2
 transpose (in module *funsor.ops.array*), 6
 triangular_inv (in module *funsor.ops.array*), 6
 triangular_solve (in module *funsor.ops.array*), 6
 truediv (in module *funsor.ops.builtin*), 5
 typing_wrap (class in *funsor.typing*), 48
- ## U
- Unary (class in *funsor.terms*), 21
 unary_contract() (in module *funsor.cnf*), 30
 unary_log_exp() (in module *funsor.cnf*), 30
 unary_neg_variable() (in module *funsor.cnf*), 30

UnaryOp (class in *funsor.ops.op*), 2
unfold_contraction_generic_tuple() (in module *funsor.optimizer*), 33
unfold_contraction_variadic() (in module *funsor.optimizer*), 33
Uniform (class in *funsor.torch.distributions*), 64
unsqueeze (in module *funsor.ops.array*), 6

V

var (in module *funsor.ops.array*), 6
var() (*Funsor* method), 18
Variable (class in *funsor.terms*), 21
Variadic (class in *funsor.typing*), 47
variance() (*Distribution* method), 61
variance() (*Independent* method), 18
VonMises (class in *funsor.torch.distributions*), 64

W

WrappedTransformOp (class in *funsor.ops.op*), 2

X

xfail_if_not_found() (in module *funsor.testing*), 45
xfail_if_not_implemented() (in module *funsor.testing*), 45
xfail_param() (in module *funsor.testing*), 45
xor (in module *funsor.ops.builtin*), 5

Z

zeros() (in module *funsor.testing*), 45