tf.contrib.distributions.VectorDiffeomixture

Contents

Class `VectorDiffeomixture`

Inherits From: `Distribution`

Defined in `tensorflow/contrib/distributions/python/ops/vector_diffeomixture.py`.

VectorDiffeomixture distribution.

The VectorDiffeomixture is an approximation to a compound distribution, i.e.,

```
p(x) = int_{X} q(x | v) p(v) dv
     = lim_{Q->infty} sum{ prob[i] q(x | loc=sum_k^K lambda[k;i] loc[k],
                                          scale=sum_k^K lambda[k;i] scale[k])
                           : i=0, ..., Q-1 }
```

where `q(x | v)` is a vector version of the `distribution` argument and `p(v)` is a SoftmaxNormal parameterized by `mix_loc` and `mix_scale`. The vector-ization of `distribution` entails an affine transformation of iid samples from `distribution`. The `prob` term is from quadrature and `lambda[k] = sigmoid(mix_loc[k] + sqrt(2) mix_scale[k] grid[k])` where the `grid` points correspond to the `prob` s.

In the non-approximation case, a draw from the mixture distribution (the "prior") represents the convex weights for different affine transformations. I.e., draw a mixing vector `v` (from the `K-1` -simplex) and let the final sample be: `y = (sum_k^K v[k] scale[k]) @ x + (sum_k^K v[k] loc[k])` where `@` denotes matrix multiplication. However, the non-approximate distribution does not have an analytical probability density function (pdf). Therefore the `VectorDiffeomixture` class implements an approximation based on Gauss-Hermite quadrature. I.e., in Note: although the `VectorDiffeomixture` is approximately the `SoftmaxNormal-Distribution` compound distribution, it is itself a valid distribution. It possesses a `sample`, `log_prob`, `mean`, `covariance` which are all mutually consistent.

Intended Use

This distribution is noteworthy because it implements a mixture of `Vector` -ized distributions yet has samples differentiable in the distribution's parameters (aka "reparameterized"). It has an analytical density function with `O(dKQ)` complexity. `d` is the vector dimensionality, `K` is the number of components, and `Q` is the number of quadrature points. These properties make it well-suited for Bayesian Variational Inference, i.e., as a surrogate family for the posterior.

For large values of `mix_scale`, the `VectorDistribution` behaves increasingly like a discrete mixture. (In most cases this limit is only achievable by also increasing the quadrature polynomial degree, `Q`.)

The term `Vector` is consistent with similar named Tensorflow `Distribution` s. For more details, see the "About `Vector` distributions in Tensorflow." section.

The term `Diffeomixture` is a portmanteau of diffeomorphism and compound mixture. For more details, see the "About `Diffeomixture` s and reparametrization.`" section.

Mathematical Details

The `VectorDiffeomixture` approximates a SoftmaxNormal-mixed ("prior") compound distribution. Using variable-substitution and Gauss-Hermite quadrature we can redefine the distribution to be a parameter-less convex combination of `K` different affine combinations of a `d` iid samples from `distribution`.

That is, defined over `R**d` this distribution is parameterized by a (batch of) length-`K` `mix_loc` and `mix_scale` vectors, a length-`K` list of (a batch of) length-`d` `loc` vectors, and a length-`K` list of `scale` `LinearOperator` s each operating on a (batch of) length-`d` vector space. Finally, a `distribution` parameter specifies the underlying base distribution which is "lifted" to become multivariate ("lifting" is the same concept as in `TransformedDistribution`).

The probability density function (pdf) is,

```
pdf(y; mix_loc, mix_scale, loc, scale, phi)
  = sum{ prob[i] phi(f_inverse(x; i)) / abs(det(interp_scale[i]))
        : i=0, ..., Q-1 }
```

where, `phi` is the base distribution pdf, and,

```
f_inverse(x; i) = inv(interp_scale[i]) @ (x - interp_loc[i])
interp_loc[i]   = sum{ lambda[k; i] loc[k]   : k=0, ..., K-1 }
interp_scale[i] = sum{ lambda[k; i] scale[k] : k=0, ..., K-1 }
```

and,

```
grid, weight = np.polynomial.hermite.hermgauss(quadrature_polynomial_degree)
prob[k]   = weight[k] / sqrt(pi)
lambda[k; i] = sigmoid(mix_loc[k] + sqrt(2) mix_scale[k] grid[i])
```

The distribution corresponding to `phi` must be a scalar-batch, scalar-event distribution. Typically it is reparameterized. If not, it must be a function of non-trainable parameters.

WARNING: If you backprop through a VectorDiffeomixture sample and the "base" distribution is both: not `FULLY_REPARAMETERIZED` and a function of trainable variables, then the gradient is not guaranteed correct!

About `Vector` distributions in TensorFlow.

The `VectorDiffeomixture` is a non-standard distribution that has properties particularly useful in variational Bayesian methods.

Conditioned on a draw from the SoftmaxNormal, `Y|v` is a vector whose components are linear combinations of affine transformations, thus is itself an affine transformation. Therefore `Y|v` lives in the vector space generated by vectors of affine-transformed distributions.

> **Note:** The marginals `Y_1|v, ..., Y_d|v` are *not* generally identical to some parameterization of `distribution`. This is due to the fact that the sum of draws from `distribution` are not generally itself the same `distribution`.

About `Diffeomixtures` and reparameterization.

The `VectorDiffeomixture` is designed to be reparameterized, i.e., its parameters are only used to transform samples from a distribution which has no trainable parameters. This property is important because backprop stops at sources of stochasticity. That is, as long as the parameters are used *after* the underlying source of stochasticity, the computed gradient is accurate.

Reparametrization means that we can use gradient-descent (via backprop) to optimize Monte-Carlo objectives. Such objectives are a finite-sample approximation of an expectation and arise throughout scientific computing.

Examples

```
ds = tf.contrib.distributions
la = tf.contrib.linalg

# Create two batches of VectorDiffeomixtures, one with mix_loc=[0.] and
# another with mix_loc=[1]. In both cases, `K=2` and the affine
# transformations involve:
# k=0: loc=zeros(dims)  scale=LinearOperatorScaledIdentity
# k=1: loc=[2.]*dims    scale=LinOpDiag
dims = 5
vdm = ds.VectorDiffeomixture(
    mix_loc=[[0.], [1]],
    mix_scale=[1.],
    distribution=ds.Normal(loc=0., scale=1.),
    loc=[
        None,  # Equivalent to `np.zeros(dims, dtype=np.float32)`.
        np.float32([2.]*dims),
    ],
    scale=[
        la.LinearOperatorScaledIdentity(
          num_rows=dims,
          multiplier=np.float32(1.1),
          is_positive_definite=True),
        la.LinearOperatorDiag(
          diag=np.linspace(2.5, 3.5, dims, dtype=np.float32),
          is_positive_definite=True),
    ],
    validate_args=True)

## Properties

<h3 id="allow_nan_stats"><code>allow_nan_stats</code></h3>

Python `bool` describing behavior when a stat is undefined.

Stats return +/- infinity when it makes sense. E.g., the variance of a
Cauchy distribution is infinity. However, sometimes the statistic is
undefined, e.g., if a distribution's pdf does not achieve a maximum within
the support of the distribution, the mode is undefined. If the mean is
undefined, then by definition the variance is undefined. E.g. the mean for
Student's T for df = 1 is undefined (no clear way to say it is either + or -
infinity), so the variance = E[(X - mean)**2] is also undefined.

#### Returns:

* <b>`allow_nan_stats`</b>: Python `bool`.

<h3 id="batch_shape"><code>batch_shape</code></h3>

Shape of a single sample from a single event index as a `TensorShape`.

May be partially defined or unknown.

The batch dimensions are indexes into independent, non-identical
parameterizations of this distribution.

#### Returns:
```

```
* <b>`batch_shape`</b>: `TensorShape`, possibly unknown.

<h3 id="distribution"><code>distribution</code></h3>

Base scalar-event, scalar-batch distribution.

<h3 id="dtype"><code>dtype</code></h3>

The `DType` of `Tensor`s handled by this `Distribution`.

<h3 id="endpoint_affine"><code>endpoint_affine</code></h3>

Affine transformation for each of `K` components.

<h3 id="event_shape"><code>event_shape</code></h3>

Shape of a single sample from a single batch as a `TensorShape`.

May be partially defined or unknown.

#### Returns:

* <b>`event_shape`</b>: `TensorShape`, possibly unknown.

<h3 id="interpolate_weight"><code>interpolate_weight</code></h3>

Grid of mixing probabilities, one for each grid point.

<h3 id="interpolated_affine"><code>interpolated_affine</code></h3>

Affine transformation for each convex combination of `K` components.

<h3 id="mixture_distribution"><code>mixture_distribution</code></h3>

Distribution used to select a convex combination of affine transforms.

<h3 id="name"><code>name</code></h3>

Name prepended to all ops created by this `Distribution`.

<h3 id="parameters"><code>parameters</code></h3>

Dictionary of parameters used to instantiate this `Distribution`.

<h3 id="quadrature_polynomial_degree"><code>quadrature_polynomial_degree</code></h3>

Polynomial largest exponent used for Gauss-Hermite quadrature.

<h3 id="reparameterization_type"><code>reparameterization_type</code></h3>

Describes how samples from the distribution are reparameterized.

Currently this is one of the static instances
`distributions.FULLY_REPARAMETERIZED`
or `distributions.NOT_REPARAMETERIZED`.

#### Returns:

An instance of `ReparameterizationType`.

<h3 id="validate_args"><code>validate_args</code></h3>

Python `bool` indicating possibly expensive checks are enabled.

## Methods
```

````
<h3 id="__init__"><code>__init__</code></h3>

``` python
__init__(
    mix_loc,
    mix_scale,
    distribution,
    loc=None,
    scale=None,
    quadrature_polynomial_degree=8,
    validate_args=False,
    allow_nan_stats=True,
    name='VectorDiffeomixture'
)
````

Constructs the VectorDiffeomixture on `R**k`.

Args:

- `mix_loc`: `float`-like `Tensor`. Represents the `location` parameter of the SoftmaxNormal used for selecting one of the `K` affine transformations.
- `mix_scale`: `float`-like `Tensor`. Represents the `scale` parameter of the SoftmaxNormal used for selecting one of the `K` affine transformations.
- `distribution`: `tf.Distribution`-like instance. Distribution from which `d` iid samples are used as input to the selected affine transformation. Must be a scalar-batch, scalar-event distribution. Typically `distribution.reparameterization_type = FULLY_REPARAMETERIZED` or it is a function of non-trainable parameters. WARNING: If you backprop through a VectorDiffeomixture sample and the `distribution` is not `FULLY_REPARAMETERIZED` yet is a function of trainable variables, then the gradient will be incorrect!
- `loc`: Length-`K` list of `float`-type `Tensor`s. The `k`-th element represents the `shift` used for the `k`-th affine transformation. If the `k`-th item is `None`, `loc` is implicitly `0`. When specified, must have shape `[B1, ..., Bb, d]` where `b >= 0` and `d` is the event size.
- `scale`: Length-`K` list of `LinearOperator`s. Each should be positive-definite and operate on a `d`-dimensional vector space. The `k`-th element represents the `scale` used for the `k`-th affine transformation. `LinearOperator`s must have shape `[B1, ..., Bb, d, d]`, `b >= 0`, i.e., characterizes `b`-batches of `d x d` matrices
- `quadrature_polynomial_degree`: Python `int`-like scalar.
- `validate_args`: Python `bool`, default `False`. When `True` distribution parameters are checked for validity despite possibly degrading runtime performance. When `False` invalid inputs may silently render incorrect outputs.
- `allow_nan_stats`: Python `bool`, default `True`. When `True`, statistics (e.g., mean, mode, variance) use the value "`NaN`" to indicate the result is undefined. When `False`, an exception is raised if one or more of the statistic's batch members are undefined.
- `name`: Python `str` name prefixed to Ops created by this class.

Raises:

- `ValueError`: if `not scale or len(scale) < 2`.
- `ValueError`: if `len(loc) != len(scale)`
- `ValueError`: if `quadrature_polynomial_degree < 1`.
- `ValueError`: if `validate_args` and any not scale.is_positive_definite.
- `TypeError`: if any scale.dtype != scale[0].dtype.
- `TypeError`: if any loc.dtype != scale[0].dtype.
````

* `NotImplementedError`: if `len(scale) != 2`.
* `ValueError`: if `not distribution.is_scalar_batch`.
* `ValueError`: if `not distribution.is_scalar_event`.

`batch_shape_tensor`

```
batch_shape_tensor(name='batch_shape_tensor')
```

Shape of a single sample from a single event index as a 1-D `Tensor`.

The batch dimensions are indexes into independent, non-identical parameterizations of this distribution.

Args:

* `name`: name to give to the op

Returns:

* `batch_shape`: `Tensor`.

`cdf`

```
cdf(
    value,
    name='cdf'
)
```

Cumulative distribution function.

Given random variable `X`, the cumulative distribution function `cdf` is:

```
cdf(x) := P[X <= x]
```

Args:

* `value`: `float` or `double` `Tensor`.
* `name`: The name to give this op.

Returns:

* `cdf`: a `Tensor` of shape `sample_shape(x) + self.batch_shape` with values of type `self.dtype`.

`copy`

```
copy(**override_parameters_kwargs)
```

Creates a deep copy of the distribution.

Note: the copy distribution may continue to depend on the original initialization arguments.

Args:

- `**override_parameters_kwargs`: String/value dictionary of initialization arguments to override with new values.

Returns:

- `distribution`: A new instance of `type(self)` initialized from the union of self.parameters and override_parameters_kwargs, i.e., `dict(self.parameters, **override_parameters_kwargs)`.

`covariance`

```
covariance(name='covariance')
```

Covariance.

Covariance is (possibly) defined only for non-scalar-event distributions.

For example, for a length-`k`, vector-valued distribution, it is calculated as,

```
Cov[i, j] = Covariance(X_i, X_j) = E[(X_i - E[X_i]) (X_j - E[X_j])]
```

where `Cov` is a (batch of) `k x k` matrix, `0 <= (i, j) < k`, and `E` denotes expectation.

Alternatively, for non-vector, multivariate distributions (e.g., matrix-valued, Wishart), `Covariance` shall return a (batch of) matrices under some vectorization of the events, i.e.,

```
Cov[i, j] = Covariance(Vec(X)_i, Vec(X)_j) = [as above]
```

where `Cov` is a (batch of) `k' x k'` matrices, `0 <= (i, j) < k' = reduce_prod(event_shape)`, and `Vec` is some function mapping indices of this distribution's event dimensions to indices of a length-`k'` vector.

Args:

- `name`: The name to give this op.

Returns:

- `covariance`: Floating-point `Tensor` with shape `[B1, ..., Bn, k', k']` where the first `n` dimensions are batch coordinates and `k' = reduce_prod(self.event_shape)`.

`entropy`

```
entropy(name='entropy')
```

Shannon entropy in nats.

`event_shape_tensor`

```
event_shape_tensor(name='event_shape_tensor')
```

Shape of a single sample from a single batch as a 1-D int32 `Tensor`.

Args:

- `name`: name to give to the op

Returns:

- `event_shape`: `Tensor`.

`is_scalar_batch`

```
is_scalar_batch(name='is_scalar_batch')
```

Indicates that `batch_shape == []`.

Args:

- `name`: The name to give this op.

Returns:

- `is_scalar_batch`: `bool` scalar `Tensor`.

`is_scalar_event`

```
is_scalar_event(name='is_scalar_event')
```

Indicates that `event_shape == []`.

Args:

- `name`: The name to give this op.

Returns:

- `is_scalar_event`: `bool` scalar `Tensor`.

`log_cdf`

```
log_cdf(
    value,
    name='log_cdf'
)
```

Log cumulative distribution function.

Given random variable `X`, the cumulative distribution function `cdf` is:

```
log_cdf(x) := Log[ P[X <= x] ]
```

Often, a numerical approximation can be used for `log_cdf(x)` that yields a more accurate answer than simply taking the logarithm of the `cdf` when `x << -1`.

Args:

- `value`: `float` or `double` `Tensor`.
- `name`: The name to give this op.

Returns:

- `logcdf`: a `Tensor` of shape `sample_shape(x) + self.batch_shape` with values of type `self.dtype`.

`log_prob`

```
log_prob(
    value,
    name='log_prob'
)
```

Log probability density/mass function.

Args:

- `value`: `float` or `double` `Tensor`.
- `name`: The name to give this op.

Returns:

- `log_prob`: a `Tensor` of shape `sample_shape(x) + self.batch_shape` with values of type `self.dtype`.

`log_survival_function`

```
log_survival_function(
    value,
    name='log_survival_function'
)
```

Log survival function.

Given random variable `X`, the survival function is defined:

```
log_survival_function(x) = Log[ P[X > x] ]
                         = Log[ 1 - P[X <= x] ]
                         = Log[ 1 - cdf(x) ]
```

Typically, different numerical approximations can be used for the log survival function, which are more accurate than `1 - cdf(x)` when `x >> 1`.

Args:

- `value`: `float` or `double` `Tensor`.
- `name`: The name to give this op.

Returns:

`Tensor` of shape `sample_shape(x) + self.batch_shape` with values of type `self.dtype`.

`mean`

```
mean(name='mean')
```

Mean.

`mode`

```
mode(name='mode')
```

Mode.

`param_shapes`

```
param_shapes(
    cls,
    sample_shape,
    name='DistributionParamShapes'
)
```

Shapes of parameters given the desired shape of a call to `sample()`.

This is a class method that describes what key/value arguments are required to instantiate the given `Distribution` so that a particular shape is returned for that instance's call to `sample()`.

Subclasses should override class method `_param_shapes`.

Args:

- `sample_shape`: `Tensor` or python list/tuple. Desired shape of a call to `sample()`.
- `name`: name to prepend ops with.

Returns:

`dict` of parameter name to `Tensor` shapes.

`param_static_shapes`

```
param_static_shapes(
    cls,
    sample_shape
)
```

param_shapes with static (i.e. `TensorShape`) shapes.

This is a class method that describes what key/value arguments are required to instantiate the given `Distribution` so that a particular shape is returned for that instance's call to `sample()`. Assumes that the sample's shape is known statically.

Subclasses should override class method `_param_shapes` to return constant-valued tensors when constant values are fed.

Args:

- `sample_shape` : `TensorShape` or python list/tuple. Desired shape of a call to `sample()` .

Returns:

`dict` of parameter name to `TensorShape` .

Raises:

- `ValueError` : if `sample_shape` is a `TensorShape` and is not fully defined.

`prob`

```
prob(
    value,
    name='prob'
)
```

Probability density/mass function.

Args:

- `value` : `float` or `double` `Tensor` .
- `name` : The name to give this op.

Returns:

- `prob` : a `Tensor` of shape `sample_shape(x) + self.batch_shape` with values of type `self.dtype` .

`quantile`

```
quantile(
    value,
    name='quantile'
)
```

Quantile function. Aka "inverse cdf" or "percent point function".

Given random variable `X` and `p in [0, 1]` , the `quantile` is:

```
quantile(p) := x such that P[X <= x] == p
```

Args:

- `value` : `float` or `double` `Tensor` .
- `name` : The name to give this op.

Returns:

- `quantile` : a `Tensor` of shape `sample_shape(x) + self.batch_shape` with values of type `self.dtype` .

`sample`

```
sample(
    sample_shape=(),
    seed=None,
    name='sample'
)
```

Generate samples of the specified shape.

Note that a call to `sample()` without arguments will generate a single sample.

Args:

- `sample_shape`: 0D or 1D `int32` `Tensor`. Shape of the generated samples.
- `seed`: Python integer seed for RNG
- `name`: name to give to the op.

Returns:

- `samples`: a `Tensor` with prepended dimensions `sample_shape`.

`stddev`

```
stddev(name='stddev')
```

Standard deviation.

Standard deviation is defined as,

```
stddev = E[(X - E[X])**2]**0.5
```

where `X` is the random variable associated with this distribution, `E` denotes expectation, and `stddev.shape = batch_shape + event_shape`.

Args:

- `name`: The name to give this op.

Returns:

- `stddev`: Floating-point `Tensor` with shape identical to `batch_shape + event_shape`, i.e., the same shape as `self.mean()`.

`survival_function`

```
survival_function(
    value,
    name='survival_function'
)
```

Survival function.

Given random variable `X`, the survival function is defined:

```
survival_function(x) = P[X > x]
                     = 1 - P[X <= x]
                     = 1 - cdf(x).
```

Args:

- `value`: `float` or `double` `Tensor`.
- `name`: The name to give this op.

Returns:

`Tensor` of shape `sample_shape(x) + self.batch_shape` with values of type `self.dtype`.

`variance`

```
variance(name='variance')
```

Variance.

Variance is defined as,

```
Var = E[(X - E[X])**2]
```

where `X` is the random variable associated with this distribution, `E` denotes expectation, and `Var.shape = batch_shape + event_shape`.

Args:

- `name`: The name to give this op.

Returns:

- `variance`: Floating-point `Tensor` with shape identical to `batch_shape + event_shape`, i.e., the same shape as `self.mean()`.