tf.contrib.distributions.RelaxedBernoulli

Contents

Class **`RelaxedBernoulli`**

Inherits From: `TransformedDistribution`

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

RelaxedBernoulli distribution with temperature and logits parameters.

The RelaxedBernoulli is a distribution over the unit interval (0,1), which continuously approximates a Bernoulli. The degree of approximation is controlled by a temperature: as the temperaturegoes to 0 the RelaxedBernoulli becomes discrete with a distribution described by the `logits` or `probs` parameters, as the temperature goes to infinity the RelaxedBernoulli becomes the constant distribution that is identically 0.5.

The RelaxedBernoulli distribution is a reparameterized continuous distribution that is the binary special case of the RelaxedOneHotCategorical distribution (Maddison et al., 2016; Jang et al., 2016). For details on the binary special case see the appendix of Maddison et al. (2016) where it is referred to as BinConcrete. If you use this distribution, please cite both papers.

Some care needs to be taken for loss functions that depend on the log-probability of RelaxedBernoullis, because computing log-probabilities of the RelaxedBernoulli can suffer from underflow issues. In many case loss functions such as these are invariant under invertible transformations of the random variables. The KL divergence, found in the variational autoencoder loss, is an example. Because RelaxedBernoullis are sampled by a Logistic random variable followed by a `tf.sigmoid` op, one solution is to treat the Logistic as the random variable and `tf.sigmoid` as downstream. The KL divergences of two Logistics, which are always followed by a `tf.sigmoid` op, is equivalent to evaluating KL divergences of RelaxedBernoulli samples. See Maddison et al., 2016 for more details where this distribution is called the BinConcrete.

An alternative approach is to evaluate Bernoulli log probability or KL directly on relaxed samples, as done in Jang et al., 2016. In this case, guarantees on the loss are usually violated. For instance, using a Bernoulli KL in a relaxed ELBO is no longer a lower bound on the log marginal probability of the observation. Thus care and early stopping are important.

Examples

Creates three continuous distributions, which approximate 3 Bernoullis with probabilities (0.1, 0.5, 0.4). Samples from these distributions will be in the unit interval (0,1).

```
temperature = 0.5
p = [0.1, 0.5, 0.4]
dist = RelaxedBernoulli(temperature, probs=p)
```

Creates three continuous distributions, which approximate 3 Bernoullis with logits (-2, 2, 0). Samples from these distributions will be in the unit interval (0,1).

```
temperature = 0.5
logits = [-2, 2, 0]
dist = RelaxedBernoulli(temperature, logits=logits)
```

Creates three continuous distributions, whose sigmoid approximate 3 Bernoullis with logits (-2, 2, 0).

```
temperature = 0.5
logits = [-2, 2, 0]
dist = Logistic(logits/temperature, 1./temperature)
samples = dist.sample()
sigmoid_samples = tf.sigmoid(samples)
# sigmoid_samples has the same distribution as samples from
# RelaxedBernoulli(temperature, logits=logits)
```

Creates three continuous distributions, which approximate 3 Bernoullis with logits (-2, 2, 0). Samples from these distributions will be in the unit interval (0,1). Because the temperature is very low, samples from these distributions are almost discrete, usually taking values very close to 0 or 1.

```
temperature = 1e-5
logits = [-2, 2, 0]
dist = RelaxedBernoulli(temperature, logits=logits)
```

Creates three continuous distributions, which approximate 3 Bernoullis with logits (-2, 2, 0). Samples from these distributions will be in the unit interval (0,1). Because the temperature is very high, samples from these distributions are usually close to the (0.5, 0.5, 0.5) vector.

```
temperature = 100
logits = [-2, 2, 0]
dist = RelaxedBernoulli(temperature, logits=logits)
```

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. 2016.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical Reparameterization with Gumbel-Softmax. 2016.

Properties

`allow_nan_stats`

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:

- `allow_nan_stats`: Python `bool`.

`batch_shape`

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:

- `batch_shape`: `TensorShape`, possibly unknown.

`bijector`

Function transforming x => y.

`distribution`

Base distribution, p(x).

`dtype`

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

`event_shape`

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

May be partially defined or unknown.

Returns:

- `event_shape`: `TensorShape`, possibly unknown.

`logits`

Log-odds of `1`.

`name`

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

`parameters`

Dictionary of parameters used to instantiate this `Distribution`.

`probs`

Probability of `1`.

`reparameterization_type`

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`.

`temperature`

Distribution parameter for the location.

`validate_args`

Python `bool` indicating possibly expensive checks are enabled.

Methods

`__init__`

```
__init__(
    temperature,
    logits=None,
    probs=None,
    validate_args=False,
    allow_nan_stats=True,
    name='RelaxedBernoulli'
)
```

Construct RelaxedBernoulli distributions.

Args:

- `temperature`: An 0-D `Tensor`, representing the temperature of a set of RelaxedBernoulli distributions. The temperature should be positive.
- `logits`: An N-D `Tensor` representing the log-odds of a positive event. Each entry in the `Tensor` parametrizes an independent RelaxedBernoulli distribution where the probability of an event is sigmoid(logits). Only one of `logits` or `probs` should be passed in.
- `probs`: An N-D `Tensor` representing the probability of a positive event. Each entry in the `Tensor` parameterizes an independent Bernoulli distribution. Only one of `logits` or `probs` should be passed in.
- `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 both `probs` and `logits` are passed, or if neither.

`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()`.

Last updated November 2, 2017.