tf.distributions.bijectors.Bijector

Contents

Class `Bijector`

Aliases:

- Class `tf.contrib.distributions.bijectors.Bijector`
- Class `tf.distributions.bijectors.Bijector`

Defined in `tensorflow/python/ops/distributions/bijector_impl.py`.

See the guide: Random variable transformations (contrib) > Bijectors

Interface for transformations of a `Distribution` sample.

Bijectors can be used to represent any differentiable and injective (one to one) function defined on an open subset of `R^n`. Some non-injective transformations are also supported (see "Non Injective Transforms" below).

Mathematical Details

A `Bijector` implements a diffeomorphism, i.e., a bijective, differentiable function. A `Bijector` is used by `TransformedDistribution` but can be generally used for transforming a `Distribution` generated `Tensor`. A `Bijector` is characterized by three operations:

1. Forward Evaluation

Useful for turning one random outcome into another random outcome from a different distribution.

1. Inverse Evaluation

Useful for "reversing" a transformation to compute one probability in terms of another.

1. (log o det o Jacobian o inverse)(x)

"The log of the determinant of the matrix of all first-order partial derivatives of the inverse function." Useful for inverting a transformation to compute one probability in terms of another. Geometrically, the det(Jacobian) is the volume of the transformation and is used to scale the probability.

By convention, transformations of random variables are named in terms of the forward transformation. The forward transformation creates samples, the inverse is useful for computing probabilities.

Example Uses

- Basic properties:

```
x = ...  # A tensor.
# Evaluate forward transformation.
fwd_x = my_bijector.forward(x)
x == my_bijector.inverse(fwd_x)
x != my_bijector.forward(fwd_x)  # Not equal because g(x) != g(g(x)).
```

- Computing a log-likelihood:

```
def transformed_log_prob(bijector, log_prob, x):
  return (bijector.inverse_log_det_jacobian(x) +
          log_prob(bijector.inverse(x)))
```

- Transforming a random outcome:

```
def transformed_sample(bijector, x):
  return bijector.forward(x)
```

Example Bijectors

- "Exponential"

```
 Y = g(X) = exp(X)
 X ~ Normal(0, 1)  # Univariate.
```

Implies:

```
   g^{-1}(Y) = log(Y)
   |Jacobian(g^{-1})(y)| = 1 / y
   Y ~ LogNormal(0, 1), i.e.,
   prob(Y=y) = |Jacobian(g^{-1})(y)| * prob(X=g^{-1}(y))
             = (1 / y) Normal(log(y); 0, 1)
```

Here is an example of how one might implement the `Exp` bijector:

```
  class Exp(Bijector):

    def __init__(self, event_ndims=0, validate_args=False, name="exp"):
      super(Exp, self).__init__(
          event_ndims=event_ndims, validate_args=validate_args, name=name)

    def _forward(self, x):
      return math_ops.exp(x)

    def _inverse(self, y):
      return math_ops.log(y)

    def _inverse_log_det_jacobian(self, y):
      return -self._forward_log_det_jacobian(self._inverse(y))

    def _forward_log_det_jacobian(self, x):
      if self.event_ndims is None:
        raise ValueError("Jacobian requires known event_ndims.")
      event_dims = array_ops.shape(x)[-self.event_ndims:]
      return math_ops.reduce_sum(x, axis=event_dims)
  ```

"Affine"
```

Y = g(X) = sqrtSigma * X + mu X ~ MultivariateNormal(0, I_d)

```
Implies:

g^{-1}(Y) = inv(sqrtSigma) * (Y - mu)
|Jacobian(g^{-1})(y)| = det(inv(sqrtSigma))
Y ~ MultivariateNormal(mu, sqrtSigma) , i.e.,
prob(Y=y) = |Jacobian(g^{-1})(y)| * prob(X=g^{-1}(y))
          = det(sqrtSigma)^(-d) *
            MultivariateNormal(inv(sqrtSigma) * (y - mu); 0, I_d)
```
```

Jacobian

The Jacobian is a reduction over event dims. To see this, consider the `Exp` `Bijector` applied to a `Tensor` which has sample, batch, and event (S, B, E) shape semantics. Suppose the `Tensor`'s partitioned-shape is `(S=[4], B=[2], E=[3, 3])`. The shape of the `Tensor` returned by `forward` and `inverse` is unchanged, i.e., `[4, 2, 3, 3]`. However the shape returned by `inverse_log_det_jacobian` is `[4, 2]` because the Jacobian is a reduction over the event dimensions.

It is sometimes useful to implement the inverse Jacobian as the negative forward Jacobian. For example,

```
def _inverse_log_det_jacobian(self, y):
   return -self._forward_log_det_jac(self._inverse(y))  # Note negation.
```

The correctness of this approach can be seen from the following claim.

- Claim:

  Assume `Y = g(X)` is a bijection whose derivative exists and is nonzero for its domain, i.e., `dY/dX = d/dX g(X) != 0`. Then:

  `none (log o det o jacobian o g^{-1})(Y) = -(log o det o jacobian o g)(X)`

- Proof:

  From the bijective, nonzero differentiability of `g`, the inverse function theorem implies `g^{-1}` is differentiable in the image of `g`. Applying the chain rule to `y = g(x) = g(g^{-1}(y))` yields `I = g'(g^{-1}(y))*g^{-1}'(y)`. The same theorem also implies `g{-1}'` is non-singular therefore: `inv[ g'(g^{-1}(y)) ] = g^{-1}'(y)`. The claim follows from properties of determinant.

Generally its preferable to directly implement the inverse Jacobian. This should have superior numerical stability and will often share subgraphs with the `_inverse` implementation.

Subclass Requirements

- Subclasses typically implement:
  - `_forward`,
  - `_inverse`,
  - `_inverse_log_det_jacobian`,
  - `_forward_log_det_jacobian` (optional).

The `_forward_log_det_jacobian` is called when the bijector is inverted via the `Invert` bijector. If undefined, a slightly less efficiently calculation, `-1 * _inverse_log_det_jacobian`, is used.

If the bijector changes the shape of the input, you must also implement:

```
- _forward_event_shape_tensor,
- _forward_event_shape (optional),
- _inverse_event_shape_tensor,
- _inverse_event_shape (optional).
```

By default the event-shape is assumed unchanged from input.

- If the `Bijector`'s use is limited to `TransformedDistribution` (or friends like `QuantizedDistribution`) then depending on your use, you may not need to implement all of `_forward` and `_inverse` functions.

Examples:

```
1. Sampling (e.g., `sample`) only requires `_forward`.
2. Probability functions (e.g., `prob`, `cdf`, `survival`) only require
   `_inverse` (and related).
3. Only calling probability functions on the output of `sample` means
  `_inverse` can be implemented as a cache lookup.
```

See "Example Uses" [above] which shows how these functions are used to transform a distribution. (Note: `_forward` could theoretically be implemented as a cache lookup but this would require controlling the underlying sample generation mechanism.)

Non Injective Transforms

WARNING Handing of non-injective transforms is subject to change.

Non injective maps `g` are supported, provided their domain `D` can be partitioned into `k` disjoint subsets, `Union{D1, ..., Dk}`, such that, ignoring sets of measure zero, the restriction of `g` to each subset is a differentiable bijection onto `g(D)`. In particular, this imples that for `y in g(D)`, the set inverse, i.e. `g^{-1}(y) = {x in D : g(x) = y}`, always contains exactly `k` distinct points.

The property, `_is_injective` is set to `False` to indicate that the bijector is not injective, yet satisfies the above condition.

The usual bijector API is modified in the case `_is_injective is False` (see method docstrings for specifics). Here we show by example the `AbsoluteValue` bijector. In this case, the domain `D = (-inf, inf)`, can be partitioned into `D1 = (-inf, 0)`, `D2 = {0}`, and `D3 = (0, inf)`. Let `gi` be the restriction of `g` to `Di`, then both `g1` and `g3` are bijections onto `(0, inf)`, with `g1^{-1}(y) = -y`, and `g3^{-1}(y) = y`. We will use `g1` and `g3` to define bijector methods over `D1` and

`D3` . `D2 = {0}` is an oddball in that `g2` is one to one, and the derivative is not well defined. Fortunately, when considering transformations of probability densities (e.g. in `TransformedDistribution`), sets of measure zero have no effect in theory, and only a small effect in 32 or 64 bit precision. For that reason, we define `inverse(0)` and `inverse_log_det_jacobian(0)` both as `[0, 0]` , which is convenient and results in a left-semicontinuous pdf.

```
abs = tf.contrib.distributions.bijectors.AbsoluteValue()

abs.forward(-1.)
==> 1.

abs.forward(1.)
==> 1.

abs.inverse(1.)
==> (-1., 1.)

# The |dX/dY| is constant, == 1.  So Log|dX/dY| == 0.
abs.inverse_log_det_jacobian(1.)
==> (0., 0.)

# Special case handling of 0.
abs.inverse(0.)
==> (0., 0.)

abs.inverse_log_det_jacobian(0.)
==> (0., 0.)
```

Properties

`dtype`

dtype of `Tensor` s transformable by this distribution.

`event_ndims`

Returns then number of event dimensions this bijector operates on.

`graph_parents`

Returns this `Bijector` 's graph_parents as a Python list.

`is_constant_jacobian`

Returns true iff the Jacobian is not a function of x.

★ **Note:** Jacobian is either constant for both forward and inverse or neither.

Returns:

- `is_constant_jacobian` : Python `bool` .

`name`

Returns the string name of this `Bijector` .

validate_args

Returns True if Tensor arguments will be validated.

Methods

__init__

```
__init__(
    event_ndims=None,
    graph_parents=None,
    is_constant_jacobian=False,
    validate_args=False,
    dtype=None,
    name=None
)
```

Constructs Bijector.

A `Bijector` transforms random variables into new random variables.

Examples:

```
# Create the Y = g(X) = X transform which operates on vector events.
identity = Identity(event_ndims=1)

# Create the Y = g(X) = exp(X) transform which operates on matrices.
exp = Exp(event_ndims=2)
```

See `Bijector` subclass docstring for more details and specific examples.

Args:

- `event_ndims`: number of dimensions associated with event coordinates.
- `graph_parents`: Python list of graph prerequisites of this `Bijector`.
- `is_constant_jacobian`: Python `bool` indicating that the Jacobian is not a function of the input.
- `validate_args`: Python `bool`, default `False`. Whether to validate input with asserts. If `validate_args` is `False`, and the inputs are invalid, correct behavior is not guaranteed.
- `dtype`: `tf.dtype` supported by this `Bijector`. `None` means dtype is not enforced.
- `name`: The name to give Ops created by the initializer.

Raises:

- `ValueError`: If a member of `graph_parents` is not a `Tensor`.

forward

```
forward(
    x,
    name='forward'
)
```

Returns the forward `Bijector` evaluation, i.e., X = g(Y).

Args:

* `x`: `Tensor`. The input to the "forward" evaluation.
* `name`: The name to give this op.

Returns:

`Tensor`.

Raises:

* `TypeError`: if `self.dtype` is specified and `x.dtype` is not `self.dtype`.
* `NotImplementedError`: if `_forward` is not implemented.

`forward_event_shape`

```
forward_event_shape(input_shape)
```

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

Same meaning as `forward_event_shape_tensor`. May be only partially defined.

Args:

* `input_shape`: `TensorShape` indicating event-portion shape passed into `forward` function.

Returns:

* `forward_event_shape_tensor`: `TensorShape` indicating event-portion shape after applying `forward`. Possibly unknown.

`forward_event_shape_tensor`

```
forward_event_shape_tensor(
    input_shape,
    name='forward_event_shape_tensor'
)
```

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

Args:

* `input_shape`: `Tensor`, `int32` vector indicating event-portion shape passed into `forward` function.
* `name`: name to give to the op

Returns:

* `forward_event_shape_tensor`: `Tensor`, `int32` vector indicating event-portion shape after applying `forward`.

`forward_log_det_jacobian`

```
forward_log_det_jacobian(
    x,
    name='forward_log_det_jacobian'
)
```

Returns both the forward_log_det_jacobian.

Args:

- `x`: `Tensor`. The input to the "forward" Jacobian evaluation.
- `name`: The name to give this op.

Returns:

`Tensor`, if this bijector is injective. If not injective this is not implemented.

Raises:

- `TypeError`: if `self.dtype` is specified and `y.dtype` is not `self.dtype`.
- `NotImplementedError`: if neither `_forward_log_det_jacobian` nor {`_inverse`, `_inverse_log_det_jacobian`} are implemented, or this is a non-injective bijector.

`inverse`

```
inverse(
    y,
    name='inverse'
)
```

Returns the inverse `Bijector` evaluation, i.e., X = g^{-1}(Y).

Args:

- `y`: `Tensor`. The input to the "inverse" evaluation.
- `name`: The name to give this op.

Returns:

`Tensor`, if this bijector is injective. If not injective, returns the k-tuple containing the unique `k` points `(x1, ..., xk)` such that `g(xi) = y`.

Raises:

- `TypeError`: if `self.dtype` is specified and `y.dtype` is not `self.dtype`.
- `NotImplementedError`: if `_inverse` is not implemented.

`inverse_event_shape`

```
inverse_event_shape(output_shape)
```

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

Same meaning as `inverse_event_shape_tensor`. May be only partially defined.

Args:

- `output_shape`: `TensorShape` indicating event-portion shape passed into `inverse` function.

Returns:

- `inverse_event_shape_tensor`: `TensorShape` indicating event-portion shape after applying `inverse`. Possibly unknown.

`inverse_event_shape_tensor`

```
inverse_event_shape_tensor(
    output_shape,
    name='inverse_event_shape_tensor'
)
```

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

Args:

- `output_shape`: `Tensor`, `int32` vector indicating event-portion shape passed into `inverse` function.
- `name`: name to give to the op

Returns:

- `inverse_event_shape_tensor`: `Tensor`, `int32` vector indicating event-portion shape after applying `inverse`.

`inverse_log_det_jacobian`

```
inverse_log_det_jacobian(
    y,
    name='inverse_log_det_jacobian'
)
```

Returns the (log o det o Jacobian o inverse)(y).

Mathematically, returns: `log(det(dX/dY))(Y)`. (Recall that: `X=g^{-1}(Y)`.)

Note that `forward_log_det_jacobian` is the negative of this function, evaluated at `g^{-1}(y)`.

Args:

- `y`: `Tensor`. The input to the "inverse" Jacobian evaluation.
- `name`: The name to give this op.

Returns:

`Tensor`, if this bijector is injective. If not injective, returns the tuple of local log det Jacobians, `log(det(Dg_i^{-1}(y)))`, where `g_i` is the restriction of `g` to the `ith` partition `Di`.

Raises:

- `TypeError` : if `self.dtype` is specified and `y.dtype` is not `self.dtype` .
- `NotImplementedError` : if `_inverse_log_det_jacobian` is not implemented.