tf.contrib.distributions.bijectors.AffineLinearOperator

Contents

Class `AffineLinearOperator`

Inherits From: `Bijector`

Defined in `tensorflow/contrib/distributions/python/ops/bijectors/affine_linear_operator_impl.py`.

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

Compute `Y = g(X; shift, scale) = scale @ X + shift`.

`shift` is a numeric `Tensor` and `scale` is a `LinearOperator`.

If `X` is a scalar then the forward transformation is: `scale * X + shift` where `*` denotes the scalar product.

Note: we don't always simply transpose X (but write it this way for brevity). Actually the input X undergoes the following transformation before being premultiplied by `scale`:

1. If there are no sample dims, we call `X = tf.expand_dims(X, 0)`, i.e., `new_sample_shape = [1]`. Otherwise do nothing.
2. The sample shape is flattened to have one dimension, i.e., `new_sample_shape = [n]` where `n = tf.reduce_prod(old_sample_shape)`.
3. The sample dim is cyclically rotated left by 1, i.e., `new_shape = [B1,...,Bb, k, n]` where `n` is as above, `k` is the event_shape, and `B1,...,Bb` are the batch shapes for each of `b` batch dimensions.

(For more details see `shape.make_batch_of_event_sample_matrices`.)

The result of the above transformation is that `X` can be regarded as a batch of matrices where each column is a draw from the distribution. After premultiplying by `scale`, we take the inverse of this procedure. The input `Y` also undergoes the same transformation before/after premultiplying by `inv(scale)`.

Example Use:

```
linalg = tf.contrib.linalg

x = [1., 2, 3]

shift = [-1., 0., 1]
diag = [1., 2, 3]
scale = linalg.LinearOperatorDiag(diag)
affine = AffineLinearOperator(shift, scale)
# In this case, `forward` is equivalent to:
# y = scale @ x + shift
y = affine.forward(x)  # [0., 4, 10]

shift = [2., 3, 1]
tril = [[1., 0, 0],
        [2, 1, 0],
        [3, 2, 1]]
scale = linalg.LinearOperatorTriL(tril)
affine = AffineLinearOperator(shift, scale)
# In this case, `forward` is equivalent to:
# np.squeeze(np.matmul(tril, np.expand_dims(x, -1)), -1) + shift
y = affine.forward(x)  # [3., 7, 11]
```

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`.

`scale`

The `scale` `LinearOperator` in `Y = scale @ X + shift`.

`shift`

The `shift` `Tensor` in `Y = scale @ X + shift`.

`validate_args`

Returns True if Tensor arguments will be validated.

Methods

`__init__`

```
__init__(
    shift=None,
    scale=None,
    event_ndims=1,
    validate_args=False,
    name='affine_linear_operator'
)
```

Instantiates the `AffineLinearOperator` bijector.

Args:

- `shift`: Floating-point `Tensor`.
- `scale`: Subclass of `LinearOperator`. Represents the (batch) positive definite matrix `M` in `R^{k x k}`.
- `event_ndims`: Scalar `integer` `Tensor` indicating the number of dimensions associated with a particular draw from the distribution. Must be 0 or 1.
- `validate_args`: Python `bool` indicating whether arguments should be checked for correctness.
- `name`: Python `str` name given to ops managed by this object.

Raises:

- `ValueError`: if `event_ndims` is not 0 or 1.
- `TypeError`: if `scale` is not a `LinearOperator`.
- `TypeError`: if `shift.dtype` does not match `scale.dtype`.
- `ValueError`: if not `scale.is_non_singular`.

`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.

Last updated November 2, 2017.