tf.contrib.distributions.bijectors.Affine

Contents

Class `Affine`

Inherits From: `Bijector`

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

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

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

Here `scale = c * I + diag(D1) + tril(L) + V @ diag(D2) @ V.T`.

In TF parlance, the `scale` term is logically equivalent to:

```
scale = (
  scale_identity_multiplier * tf.diag(tf.ones(d)) +
  tf.diag(scale_diag) +
  scale_tril +
  scale_perturb_factor @ diag(scale_perturb_diag) @
    tf.transpose([scale_perturb_factor])
)
```

The `scale` term is applied without necessarily materializing constituent matrices, i.e., the matmul is matrix-free when possible.

Examples:

```
# Y = X
b = Affine()

# Y = X + shift
b = Affine(shift=[1., 2, 3])

# Y = 2 * I @ X.T + shift
b = Affine(shift=[1., 2, 3],
           scale_identity_multiplier=2.)

# Y = tf.diag(d1) @ X.T + shift
b = Affine(shift=[1., 2, 3],
           scale_diag=[-1., 2, 1])         # Implicitly 3x3.

# Y = (I + v * v.T) @ X.T + shift
b = Affine(shift=[1., 2, 3],
           scale_perturb_factor=[[1., 0],
                                 [0, 1],
                                 [1, 1]])

# Y = (diag(d1) + v * diag(d2) * v.T) @ X.T + shift
b = Affine(shift=[1., 2, 3],
           scale_diag=[1., 3, 3],          # Implicitly 3x3.
           scale_perturb_diag=[2., 1],     # Implicitly 2x2.
           scale_perturb_factor=[[1., 0],
                                 [0, 1],
                                 [1, 1]])
```

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_identity_multiplier=None,
    scale_diag=None,
    scale_tril=None,
    scale_perturb_factor=None,
    scale_perturb_diag=None,
    event_ndims=1,
    validate_args=False,
    name='affine'
)
```

Instantiates the `Affine` bijector.

This `Bijector` is initialized with `shift` `Tensor` and `scale` arguments, giving the forward operation:

```
Y = g(X) = scale @ X + shift
```

where the `scale` term is logically equivalent to:

```
scale = (
  scale_identity_multiplier * tf.diag(tf.ones(d)) +
  tf.diag(scale_diag) +
  scale_tril +
  scale_perturb_factor @ diag(scale_perturb_diag) @
    tf.transpose([scale_perturb_factor])
)
```

If none of `scale_identity_multiplier`, `scale_diag`, or `scale_tril` are specified then `scale += IdentityMatrix`. Otherwise specifying a `scale` argument has the semantics of `scale += Expand(arg)`, i.e., `scale_diag != None` means `scale += tf.diag(scale_diag)`.

Args:

- `shift`: Floating-point `Tensor`. If this is set to `None`, no shift is applied.
- `scale_identity_multiplier`: floating point rank 0 `Tensor` representing a scaling done to the identity matrix. When `scale_identity_multiplier = scale_diag = scale_tril = None` then `scale += IdentityMatrix`. Otherwise no

scaled-identity-matrix is added to `scale`.

- `scale_diag`: Floating-point `Tensor` representing the diagonal matrix. `scale_diag` has shape [N1, N2, ... k], which represents a k x k diagonal matrix. When `None` no diagonal term is added to `scale`.
- `scale_tril`: Floating-point `Tensor` representing the diagonal matrix. `scale_diag` has shape [N1, N2, ... k, k], which represents a k x k lower triangular matrix. When `None` no `scale_tril` term is added to `scale`. The upper triangular elements above the diagonal are ignored.
- `scale_perturb_factor`: Floating-point `Tensor` representing factor matrix with last two dimensions of shape `(k, r)`. When `None`, no rank-r update is added to `scale`.
- `scale_perturb_diag`: Floating-point `Tensor` representing the diagonal matrix. `scale_perturb_diag` has shape [N1, N2, ... r], which represents an `r x r` diagonal matrix. When `None` low rank updates will take the form `scale_perturb_factor * scale_perturb_factor.T`.
- `event_ndims`: Scalar `int` `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 `perturb_diag` is specified but not `perturb_factor`.
- `TypeError`: if `shift` has different `dtype` from `scale` arguments.

`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.