Romain Hardy

20 Followers About Follow

Building custom bijectors with Tensorflow Probability

How to make your own bijectors and normalizing flows

Romain Hardy Jan 18 · 3 min read

Introduction

We often need to approximate distributions using models. If the target distribution has a known form, such as a Gaussian, then we can simply find the values of the mean and variance that best fit the data. What if the data has a more complex distribution? Chances are, if you try to fit a simple distribution to complex data, the result will be mediocre. Luckily, Tensorflow Probability has straightforward tools for modelling complex distributions, via **bijectors**. A bijector is a Tensorflow component representing a diffeomorphism — a bijective, differentiable function — that allows us to move freely between random variables. To understand how this works, let's take a look at the change of variables formula. Let X and Z be random variables, and f a bijective, differentiable function such that

$$f(X) = Z \quad (1)$$

The probability distributions of X and Z are related by

$$p_Z(\mathbf{z}) = p_X(f^{-1}(\mathbf{x}))\left|\det\left(\frac{df^{-1}(\mathbf{z})}{d\mathbf{z}}\right)\right| \quad (2)$$

themselves diffeomorphisms, bijectors are composable; we refer to a series of bijectors as a **normalizing flow**. With the right bijectors, normalizing flows can transform simple distributions into complex ones.

Making custom bijectors

Tensorflow already implements many interesting bijectors, but we can build custom ones from scratch. Let's make a bijector representing a rotation in two dimensions. We need to specify three things: the forward transformation, the inverse transformation, and the Jacobian determinant. Referring to the formulas above, these are all you need to sample the transformed variable and evaluate its density.

```
import tensorflow as tf
import tensorflow_probability as tfp

tfb = tfp.bijectors

class Rotation2D(tfb.Bijector):
  def __init__(self, theta, validate_args=False, name="rotation_2d"):
    super(Rotation2D, self).__init__(
      validate_args=validate_args,
      forward_min_event_ndims=1,
      inverse_min_event_ndims=1,
      name=name,
      is_constant_jacobian=True
    )

    self.cos_theta = tf.math.cos(theta)
    self.sin_theta = tf.math.sin(theta)
    self.event_ndim = 1

  def _forward(self, x):
    batch_ndim = len(x.shape) - self.event_ndim
    x0 = tf.expand_dims(x[..., 0], batch_ndim)
    x1 = tf.expand_dims(x[..., 1], batch_ndim)
    y0 = self.cos_theta * x0 - self.sin_theta * x1
    y1 = self.sin_theta * x0 + self.cos_theta * x1
    return tf.concat((y0, y1), axis=-1)

  def _inverse(self, y):
    batch_ndim = len(y.shape) - self.event_ndim
    y0 = tf.expand_dims(y[..., 0], batch_ndim)
    y1 = tf.expand_dims(y[..., 1], batch_ndim)
```

```
35
36    def _forward_log_det_jacobian(self, x):
37      return tf.constant(0., x.dtype)
```

rotation2d.py hosted with ♡ by GitHub view raw

Code for a custom rotation bijector

There's a lot to unpack here, so let's go through it step by step. The constructor takes in an argument `theta` representing the angle of the rotation. The parameters `forward_min_event_dims` and `inverse_min_event_dims` are the minimum event dimensions of tensors on which the forward and inverse transformations operate, respectively. Since our bijector transforms 2D tensors, these are both set to 1. The forward transformation is defined by the following matrix multiplication:

$$R_\theta(x_0, x_1) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \quad (3)$$

The inverse transformation is similar, replacing θ with its negative. Finally, the Jacobian determinant of the forward transformation is

$$\det \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \cos^2\theta + \sin^2\theta = 1 \quad (4)$$

Notice that the function we implemented actually returns the logarithm of the Jacobian determinant, since we usually prefer to work in log-space. Also note that we do not need to explicitly define the log-Jacobian determinant of the inverse transformation, since according to the inverse function theorem, the Jacobian determinant of the inverse transformation is the reciprocal of the Jacobian determinant of the forward transformation.

Great! Our rotation bijector is ready to go.

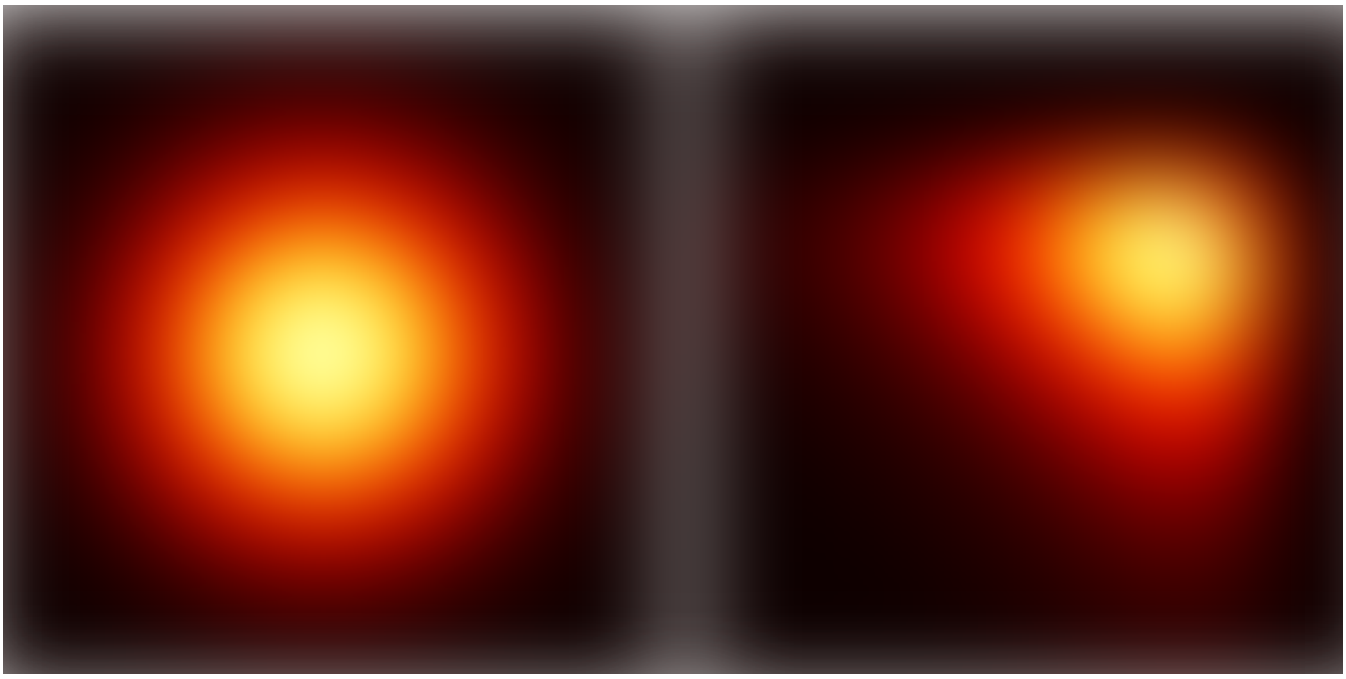
Bijectors in practice

Let's use our new bijector to transform a multivariate normal distribution. A rotation won't do much by itself, but we can compose it with other bijectors to produce a more intricate flow. We'll create the transformed distribution using Tensorflow's

Code for a transformed distribution

Notice that we have to reverse the order of the bijectors before chaining them. Let's compare the contour plots of the base and transformed distributions.

Code for creating contour plots



Contour plots for the base distribution (left) and transformed distribution (right)

What's next?

tuned for future articles about training normalizing flows!

Machine Learning Data Science Mathematics TensorFlow