### 1 Introduction

The success of deep neural networks presents exciting implications for probabilistic programming. New classes of “deep” probabilistic models parameterized by neural networks have demonstrated success in modeling images, sounds, and video, as well as on novel problems such as understanding mouse behavior or learning causality in genome-wide association studies. [5, 3, 12] Meanwhile, the deep-learning community’s investment in powerful tooling and reusable modules has accelerated research by enabling reproducibility and fast iteration.

---

Figure 1: General VAE.

```python
s = make_encoder(x)
z = s.samples[n]
r = make_prior()
avg_elbo_loss = tf.reduce_mean(e.log_prob(z) - d.log_prob(x) - r.log_prob(z))
train = tf.train.Optimizer().minimize(avg_elbo_loss)
```

Figure 2: Standard MNIST VAE with Gaussian encoder and Bernoulli decoder.

```python
i.e., tf=tensorflow, ds=tf.contrib.distributions, bs=tf.contrib.bijectors.
```

Figure 3: Shape of $z$, the encoded images.

The TensorFlow Distributions library implements building blocks for probabilistic models. It offers standard and nonstandard [2] distributions over continuous and discrete spaces with methods for sampling, log density, and statistics (mean, mode, variance, entropy, etc), as well as invertible transformations for composing additional structure. Incorporating these in a TensorFlow computational graph [1] enables sophisticated models while inheriting TensorFlow’s GPU acceleration, configurable precision, common subexpression elimination, and automatic differentiation. This vastly simplifies gradient-based inference techniques, e.g., HMC [7] and ADVI [6]. The Distributions library is widely used within Google and DeepMind, serves as the back-end for the probabilistic programming system Edward [13].

Key design goals are numerical stability, vectorization, composability, and debuggability. Figures 1 and 2 demonstrate many of these virtues. Using a Bernoulli decoder and Gaussian encoder, prior, Figure 1 implements a baseline variational autoencoder of MNIST handwritten digits. [5] By changing just a few lines (Figure 4), the architecture becomes state-of-the-art: a PixelCNN++ [10] decoder and a convolutional encoder network pushed through an autoregressive flow, which represents dependence between variables in the posterior [4, 8].

(AutoregressiveImageDist omitted for space; implemented using Bijector.) This demonstrates the power of distribution composition: simple modules combined to form rich models.

*{jvdillon, langmore}@google.com

---

Figure 4: State-of-the-art architecture using PixelCNN++ decoder and AR flows for encoder and prior.
We briefly describe two key aspects of the library that enable this elegance: distribution shape semantics, and the Bijector framework for random variable transformations.

2 Shape Semantics

Distribution methods generally adhere to a Tensor-in, Tensor-out design. Within that framework, distributions (conceptually) partition a Tensor’s shape into three groups. Event shape describes a single draw from the underlying distribution; this is the standard concept of shape in probabilistic models. Sample shape indexes iid draws. We also introduce batch shape; it indexes independent draws from different parameterizations of the same distribution family. Figure 3 illustrates this partition for the VAE model. Combining these concepts in a single Tensor enables efficient vectorized computation and ergonomic broadcasting. For example,

```
# Initialize 3-batch of 2-variate
# MultivariateNormals each with different
# mean.
mvn = ds.MultivariateNormalDiag(
  loc=[[1., 1.], [2., 2.], [3., 3.]])
# Take 10 samples across 3 batch members.
# Each sample in R^2.
x = mvn.sample(10)
# Compute 10 pdf calculations for each of
# the 3 batch members.
pdf = mvn.prob(x)
```

This procedure is automatically vectorized because the internal calculations are over tensors, each representing the differently parameterized Normal distributions. loc and x are automatically broadcast, their value is applied pointwise thus eliding n copies.

3 Bijector: Random Variable Transforms

We also introduce an interface for transformations of samples from a distribution. A Bijector represents a differentiable and bijective map, i.e., a diffeomorphism. It is characterized by three operations: a forward transform, an inverse transform, and the log determinant of its Jacobian matrix, which captures the local volume scaling of the transformation and appears in the change of variables formula for probability densities. Similar abstractions exist in other systems such as PyMC3 [11]; ours is distinguished by support for non-injective transformations, and input-output caching, and by the breadth of transformations integrated natively with Distributions and the TensorFlow ecosystem.

A bijector can transform a Tensor directly, but most commonly is used to construct new Distributions. For example,

```
# Initialize 3-batch of 2-variate
# MultivariateNormals each with different
# mean.
mvn = ds.TransformedDistribution(
  distribution=ds.Normal(0., 1.),
  bijector=bs.Affine(
    shift=mu,
    scale_tril=tf.cholesky(Sigma)),
  event_shape=[d])
```

uses the Affine bijector to implement a Multivariate Gaussian parameterized by a mean vector mu and covariance matrix Sigma. Given a Bijector instance, TransformedDistribution automatically implements sample, log_prob, and prob. It also automatically implements statistics such as mean, variance, entropy, etc. whenever the bijector has a constant Jacobian.

Bijectors may be chained and inverted, enabling simple construction of sophisticated Distributions; for example, a multivariate logit-Normal with matrix-shaped events:

```
matrix_logit_mvn =
  ds.TransformedDistribution(
    distribution=ds.Normal(0., 1.),
    bijector=bs.Chain([bs.Reshape([d, d]),
      bs.SoftmaxCentered(),
      bs.Affine(scale_diag=diag),
    ]),
    event_shape=[d * d])
```

The Bijector API automatically caches input/output pairs of its operations. In many applications this elides an inverse calculation; e.g., in variational inference [9, 6] the approximating posterior $q(z \mid x)$ is only ever asked to compute log_prob of its own samples, so the pre-transformation sample is always known and cached. In the case of an Affine (TriL) transform, this elides back substitution, while for InverseAutoregressiveFlows [4] (e.g., bs.Invert(bs.MaskedAutoregressiveFlow(...))) the complexity is reduced from quadratic to linear (in the event size).

---

2 Non-injective transformations $F$ are supported, provided that, ignoring sets of measure zero, their domain $D$ can be partitioned into $D_1 \cup \cdots \cup D_K$, such that the restriction $F : D_k \to F(D)$ is a diffeomorphism (e.g., AbsoluteValue, Square).
References


