TensorFlow Distributions

Joshua V. Dillon, Ian Langmore, Eugene Brevdo, Srinivas Vasudevan,
Brian Patton, Matt Hoffman, Dave Moore, Dustin Tran, Rif A. Saurous

Google Research*

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.

```
e = make_encoder(x)
z = e.sample(n)
d = make_decoder(z)
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 1: General VAE.[1]

```
def make_encoder(x, z_size=8):
  net = make_nn(x, z_size*2)
  return ds.MultivariateNormalDiag(
      loc=net[:z_size],
      scale_diag=net[z_size:]))

def make_decoder(z, x_shape=(28, 28, 1)):
  net = make_nn(z, tf.reduce_prod(x_shape))
  return ds.Independent(
    ds.Bernoulli(logits=tf.reshape(net, x_shape)),
    reduce_batch_ndims=tf.shape(x_shape)[0])

def make_prior(z_size=8, dtype=tf.float32):
  return ds.MultivariateNormalDiag(
    loc=tf.zeros(z_size, dtype)))

def make_nn(x, out_size, hidden_size=[128, 32]):
  net = tf.flatten(x)
  for h in hidden_size:
    net = tf.layers.dense(net, h)
  return tf.layers.dense(net, out_size, activation=None)
```

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

$$\left[\underbrace{\text{n Monte Carlo draws}}_{\texttt{sample\_shape}\text{ (iid latent draws)}}, \underbrace{\text{b examples per batch}}_{\texttt{batch\_shape}\text{ (indep, \textit{not} identical)}}, \underbrace{\text{8 latent dimensions}}_{\texttt{event\_shape}}\right]$$

Figure 3: Shape of $\mathbf{z}$, the encoded images.

```
import convnet, pixelcnnpp, AutoregressiveImageDist

def make_encoder(x, z_size=8):
  net = convnet(x, z_size*2)
  return make_arflow(ds.MultivariateNormalDiag(
      loc=net[:z_size], scale_diag=net[z_size:])))

def make_decoder(z, x_shape=(28, 28, 1)):
  logit_template = pixelcnnpp(z, x_shape)
  return AutoregressiveImageDist(logit_template)

def make_prior(z_size=8, dtype=tf.float32):
  return make_arflow(ds.MultivariateNormalDiag(
    loc=tf.zeros([z_size], dtype)))

def make_arflow(z, n_flows=4, hidden_layers=(640,)*3):
  chain = list(itertools.chain.from_iterable([
      bs.Invert(bs.MaskedAutoregressiveFlow(
        shift_and_log_scale_fn=(
          bs.masked_autoregressive_default_template(
            hidden_layers)))),
      bs.Permute(np.random.permutation(n_z)),
    ] for _ in range(n_flows)))
  return ds.TransformedDistribution(
    distribution=z, bijector=bs.Chain(chain[:-1]))
```

Figure 4: State-of-the-art architecture using PixelCNN++ decoder and AR flows for encoder and prior.

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`

[1]Throughout examples we abbreviate namespaces, i.e., `tf=tensorflow`, `ds=tf.contrib.distributions`, `bs=ds.bijectors`.

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)
# ==> x.shape=[10, 3, 2].
# Compute 10 pdf calculations for each of
# the 3 batch members.
pdf = mvn.prob(x)
# ==> pdf.shape=[10, 3].
```

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,[2] 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,

```
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 $q(z|x)$ 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 diffeomorphisim (e.g., `AbsoluteValue`, `Square`).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Joshua V. Dillon and Ian Langmore. Quadrature Compound Distribution: An Approximating Family. *(preprint, to appear)*, 2017.

[3] Matthew Johnson, David K Duvenaud, Alex Wiltschko, Ryan P Adams, and Sandeep R Datta. Composing graphical models with neural networks for structured representations and fast inference. In *Advances in neural information processing systems*, pages 2946–2954, 2016.

[4] Diederik P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improving Variational Inference with Inverse Autoregressive Flow. In *Advances in Neural Information Processing Systems*, pages 4743–4751, 2016.

[5] Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[6] Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. Automatic Differentiation Variational Inference. *arXiv preprint arXiv:1603.00788*, 2016.

[7] Radford M Neal et al. MCMC Using Hamiltonian Dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11), 2011.

[8] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked Autoregressive Flow for Density Estimation. *arXiv preprint arXiv:1705.07057*, 2017.

[9] Rajesh Ranganath, Sean Gerrish, and David Blei. Black box variational inference. In *Artificial Intelligence and Statistics*, pages 814–822, 2014.

[10] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P Kingma. PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications. *arXiv preprint arXiv:1701.05517*, 2017.

[11] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

[12] Dustin Tran and David Blei. Implicit Causal Models. *(preprint, to appear)*, 2017.

[13] Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. Deep Probabilistic Programming. *arXiv preprint arXiv:1701.03757*, 2017.