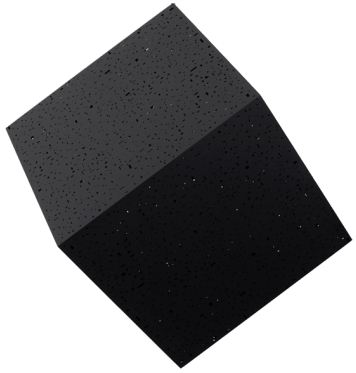
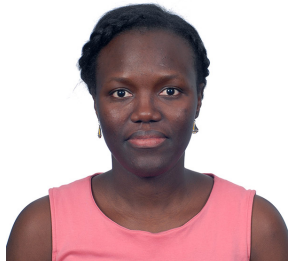
Deep Probabilistic Programming: TensorFlow Distributions and Edward

Rif A. Saurous
Dustin Tran
Google, Columbia University

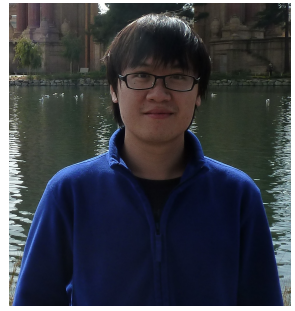


Alp Kucukelbir



Adji Dieng

Dave Moore



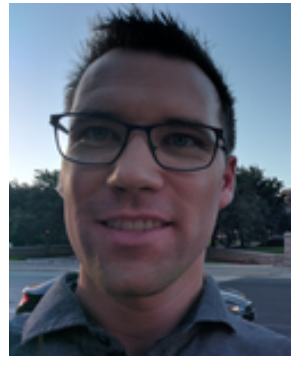
Dawen Liang

Eugene Brevdo

Ian Langmore

Josh Dillon

Maja Rudolph



Brian Patton

Srinivas Vasudevan

Alex Alemi

Matt Hoffman

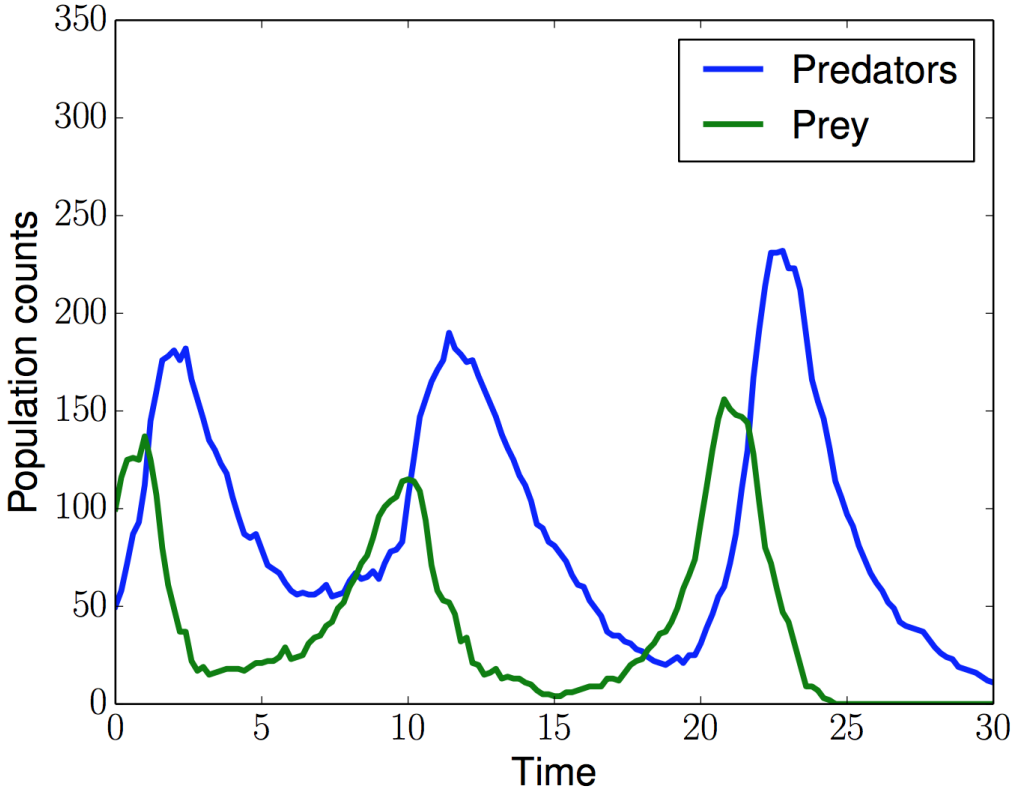
David Blei

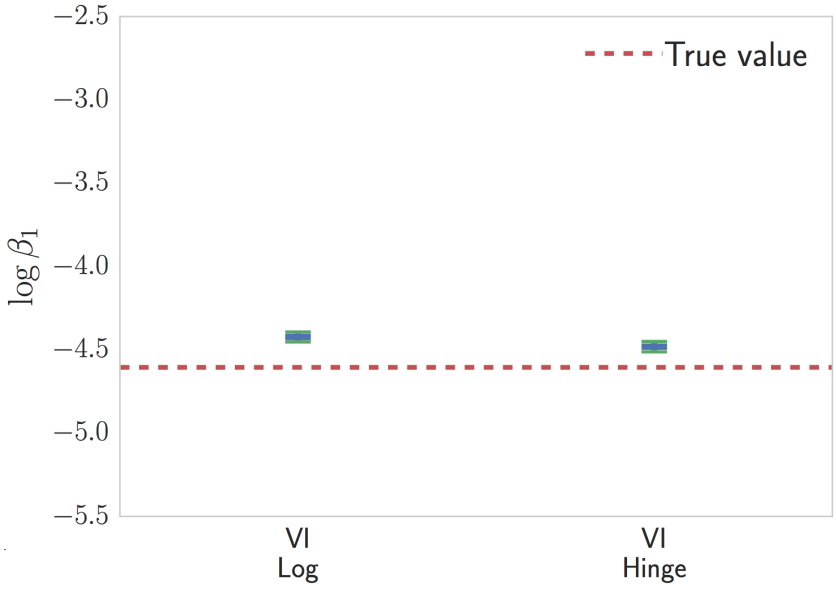
Kevin Murphy

Rif A. Saurous

Exploratory analysis of 1.7M taxi trajectories, in Stan

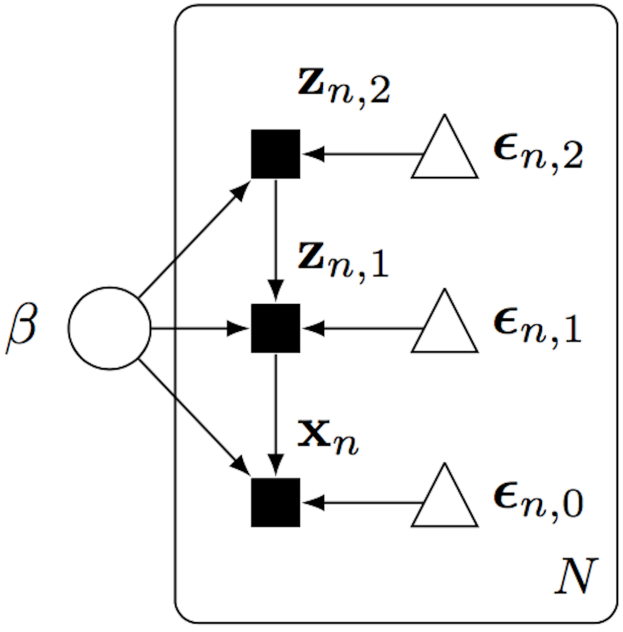
[Kucukelbir+ 2017]

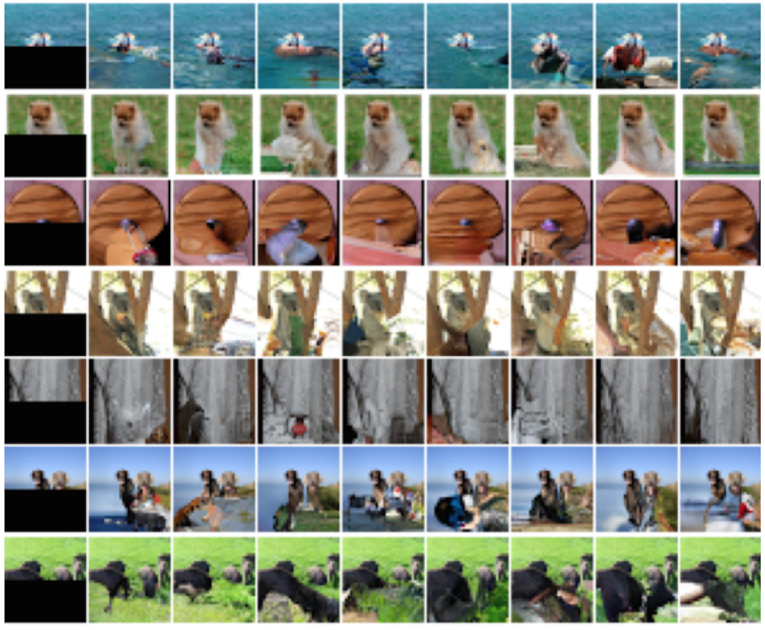




Simulators of 100K time series in ecology, in Edward

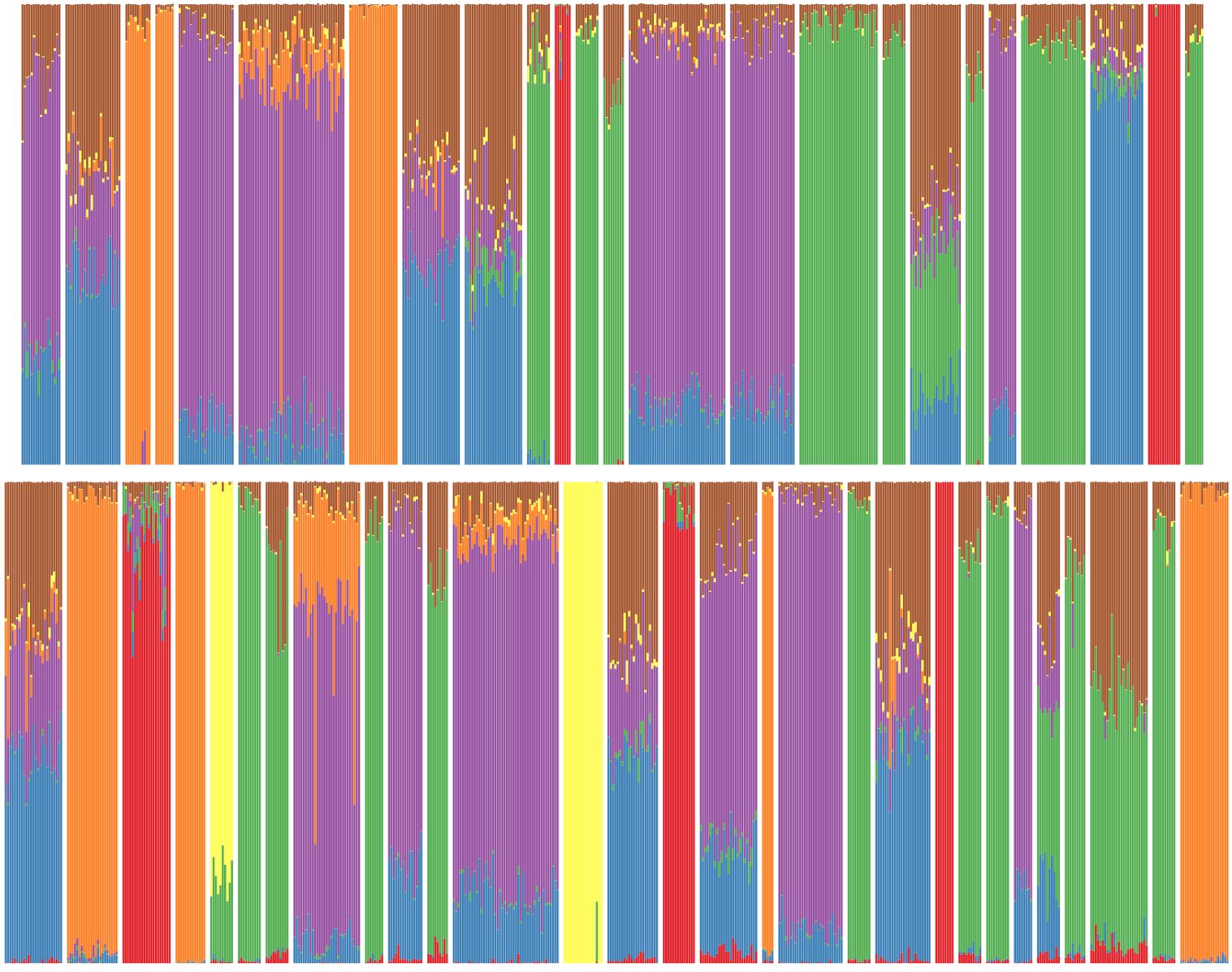
[Tran+ 2017]





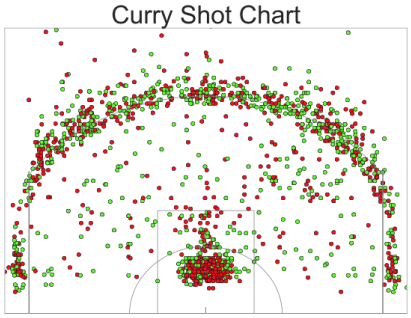
Generation & compression of 10M colored 32x32 images, in Edward

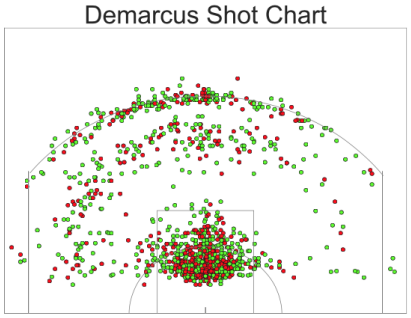
[Tran+ 2017; fig from Van der Oord+ 2016]

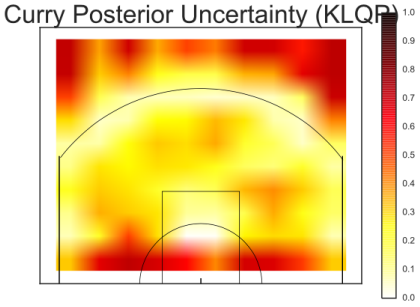


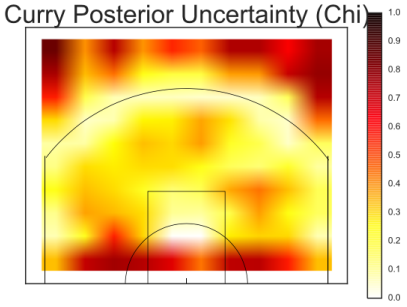
Cause and effect of 1.6B genetic measurements, in Edward

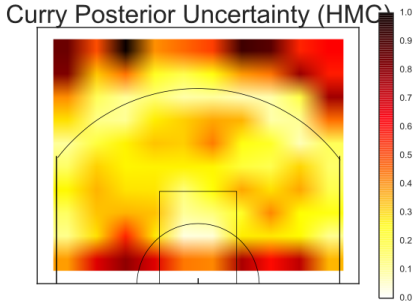
[Tran Blei 2017; fig from Gopalan+ 2017]

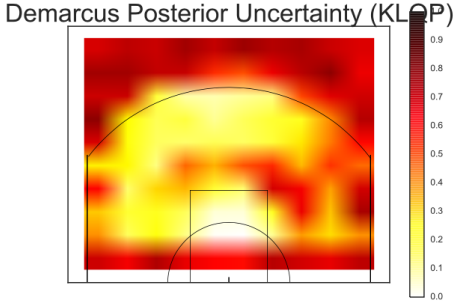


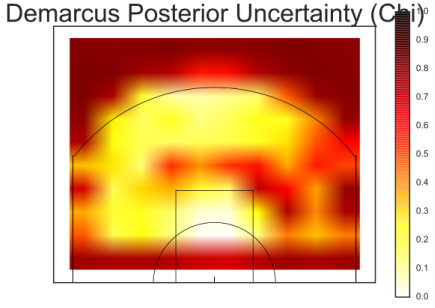


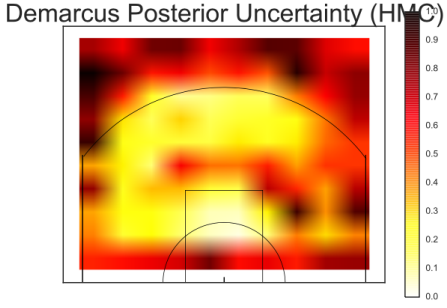












Spatial analysis of 150,000 shots from 308 NBA players, in Edward

[Dieng+ 2017]

Probabilistic machine learning

- A probabilistic model is a joint distribution of hidden variables $\mathbf{z}$ and observed variables $\mathbf{x}$,

$$p(\mathbf{z}, \mathbf{x}).$$

- Inference about the unknowns is through the **posterior**, the conditional distribution of the hidden variables given the observations

$$p(\mathbf{z} \mid \mathbf{x}) = \frac{p(\mathbf{z}, \mathbf{x})}{p(\mathbf{x})}.$$

- For most interesting models, the denominator is not tractable. We appeal to **approximate posterior inference**.

What is probabilistic programming?

Probabilistic programs reify models from mathematics to physical objects.

- Each model is equipped with memory (“bits”, floating point, storage) and computation (“flops”, scalability, communication).

Anything you do lives in the world of probabilistic programming.

- Any computable model.

  ex. graphical models; neural networks; SVMs; stochastic processes.

- Any computable inference algorithm.

  ex. automated inference; model-specific algorithms; inference within inference (learning to learn).

- Any computable application.

  ex. exploratory analysis; object recognition; code generation; causality.

TensorFlow Distributions

From www.tensorflow.org: "TensorFlow" is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API."

TensorFlow Distributions offers efficient, composable manipulations of probability distributions.

Goals: fast, numerically stable, idiomatic TensorFlow. Non-goals:

- Universality: All distributions offer **sample** and **log_prob** computable in polynomial time.
- Approximate inference.

For more details, see our paper on arXiv. A colab with examples is available at http://goo.gl/PHGNkQ.

The Two Abstractions

TensorFlow Distributions offers two primary abstractions:

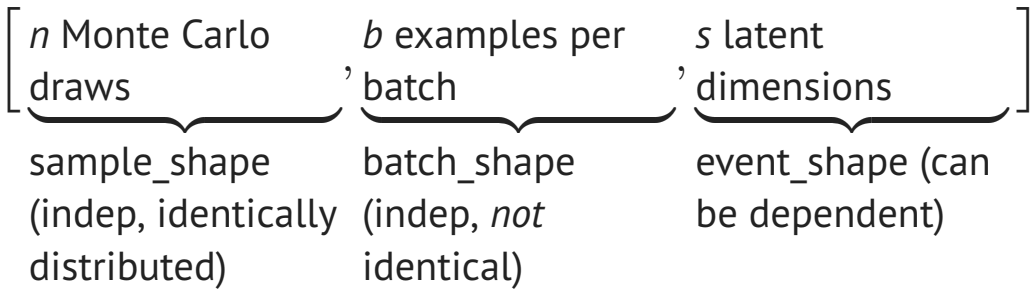
- **Distributions**: Probability distributions, with fast numerically stable methods for sampling and computing log probabilities. 55+ built-in, framework is extensible.
- **Bijectors**: Composable transformations with tractable log det Jacobians. 22+ built-in, framework is extensible.

TensorFlow Distributions Philosophy

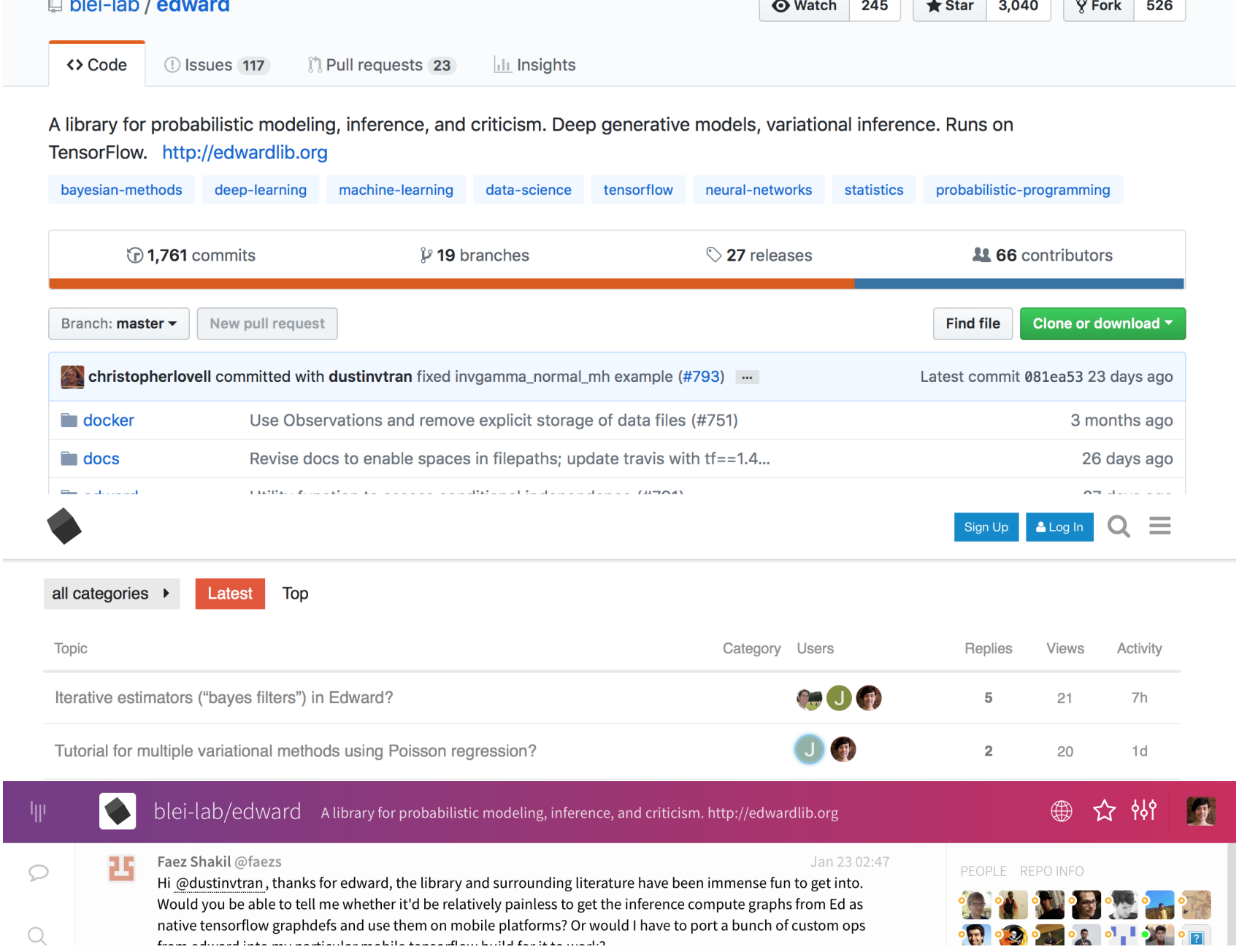
- Low-level, bottom-up.
- Conservative.
- Tookit or library rather than language or solution.
- Mix-and-match with TensorFlow.
- Substrate for multiple higher-level solutions: Probabilistic layers, Edward.

Shape Semantics

TensorFlow distributions has a fairly complex notion of shape:



- **Event shape** is the shape of a draw from a single distribution. Corresponds to the minimal size output produced by a call to sample in SciPy.
- **Batch shape** is used to represent a collection of independent *non-identical* distributions. There is no way to draw or work with just part of the batch. This notion has no SciPy equivalent.
- **Sample shape** describes the shape of sample (each of size given by the event shape) to draw.



We have an active community of several thousand users & many contributors.

Model

Edward's language augments computational graphs with an abstraction for random variables. Each random variable **x** is associated to a tensor $\mathbf{x}^*$, $\mathbf{x}^* \sim p(\mathbf{x} \mid \theta^*)$.

```
# univariate normal
Normal(loc=0.0, scale=1.0)
# vector of 5 univariate normals
Normal(loc=tf.zeros(5), scale=tf.ones(5))
# 2 x 3 matrix of Exponentials
Exponential(rate=tf.ones([2, 3]))
```

Unlike `tf.Tensors`, `ed.RandomVariables` carry an explicit density with methods such as `log_prob()` and `sample()`.

For implementation, we wrap all TensorFlow Distributions and call `sample` to produce the associated tensor.

[Tran+ 2017]

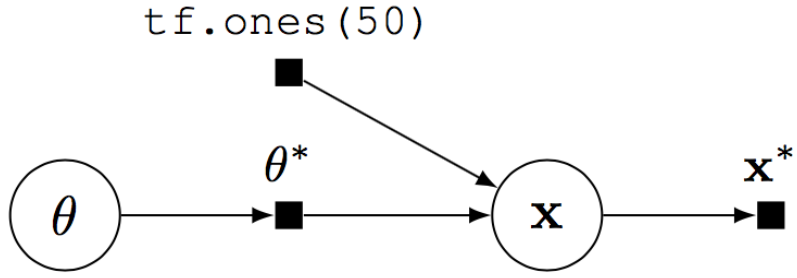
Example: Beta-Bernoulli

Consider a Beta-Bernoulli model,

$$p(\mathbf{x}, \theta) = \text{Beta}(\theta \mid 1, 1) \prod_{n=1}^{50} \text{Bernoulli}(x_n \mid \theta),$$

where θ is a probability shared across 50 data points $\mathbf{x} \in \{0, 1\}^{50}$.

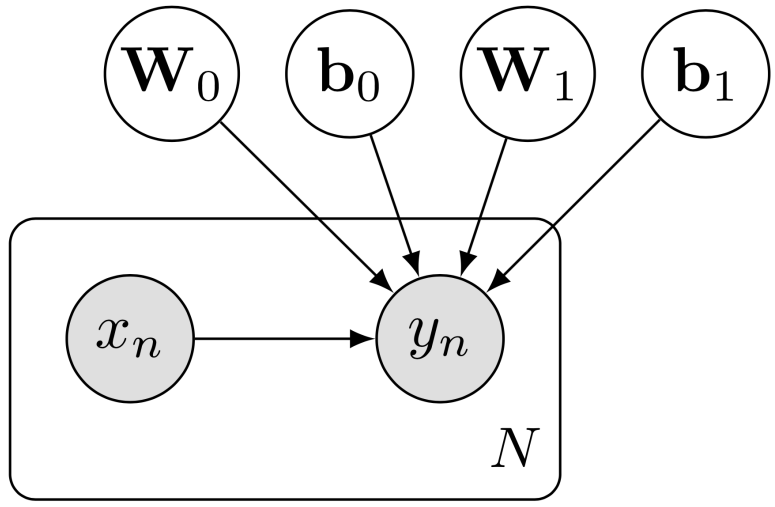
```
1  theta = Beta(1.0, 1.0)
2  x = Bernoulli(probs=tf.ones(50) * theta)
```



Fetching **x** from the graph generates a binary vector of 50 elements.

All computation is represented on the graph, enabling us to leverage model structure during inference.

Example: Bayesian neural network for classification



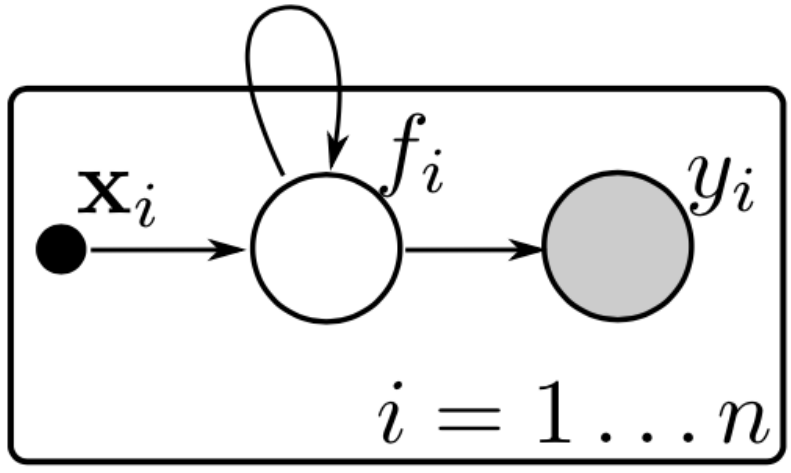

```
W_0 = Normal(mu=tf.zeros([D, H]), sigma=tf.ones([D, H]))
W_1 = Normal(mu=tf.zeros([H, 1]), sigma=tf.ones([H, 1]))
b_0 = Normal(mu=tf.zeros(H), sigma=tf.ones(L))
b_1 = Normal(mu=tf.zeros(1), sigma=tf.ones(1))

x = tf.placeholder(tf.float32, [N, D])
y = Bernoulli(logits=tf.matmul(tf.nn.tanh(tf.matmul(x, W_0) + b_0), W_1) + b_1)
```

[Denker+ 1987; MacKay 1992; Hinton & Van Camp, 1993; Neal 1995]

Example: Bayesian neural network for classification

[Demo]

Example: Gaussian process classification




```
X = tf.placeholder(tf.float32, [N, D])
f = MultivariateNormalTriL(loc=tf.zeros(N),
                           scale_tril=tf.cholesky(rbf(X)))
y = Bernoulli(logits=f)
```

[Rasmussen & Williams, 2006; fig from Hensman+ 2013]

Inference

Given

- Data $\mathbf{x}_{\text{train}}$.
- Model $p(\mathbf{x}, \mathbf{z}, \beta)$ of observed variables $\mathbf{x}$ and latent variables $\mathbf{z}, \beta$.

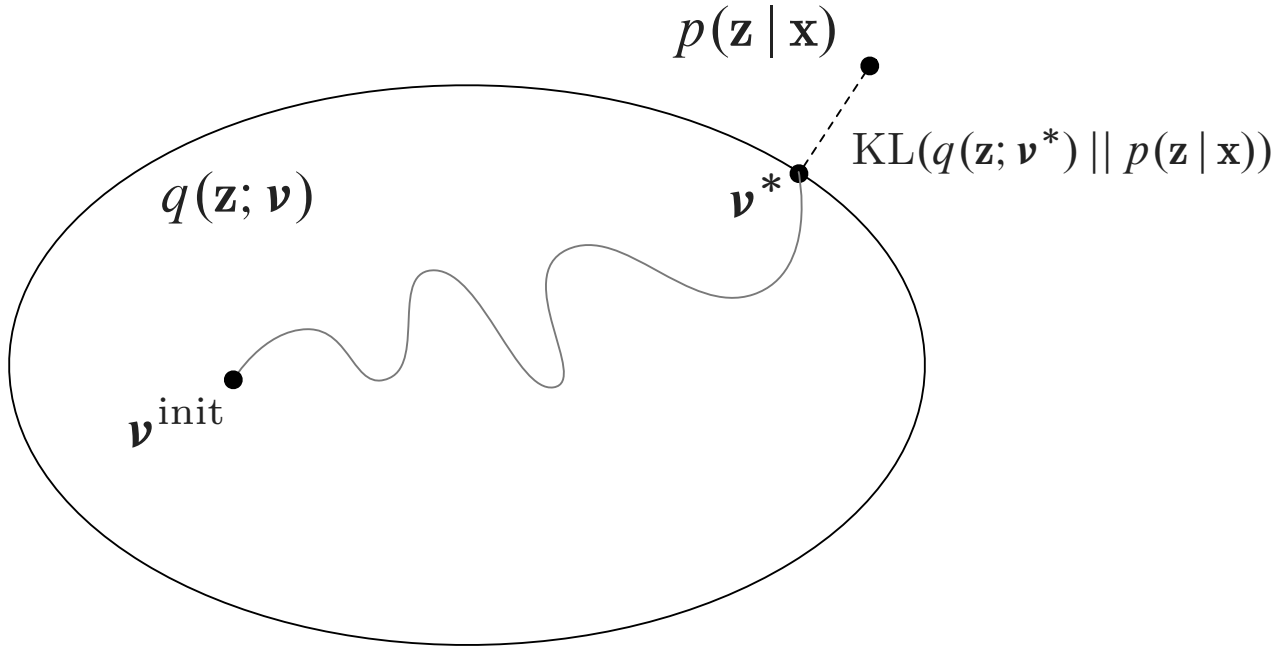
Goal

- Calculate posterior distribution

$$p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}}) = \frac{p(\mathbf{x}_{\text{train}}, \mathbf{z}, \beta)}{\int p(\mathbf{x}_{\text{train}}, \mathbf{z}, \beta) \, d\mathbf{z} \, d\beta}.$$

This is the key problem in Bayesian inference.

Variational inference

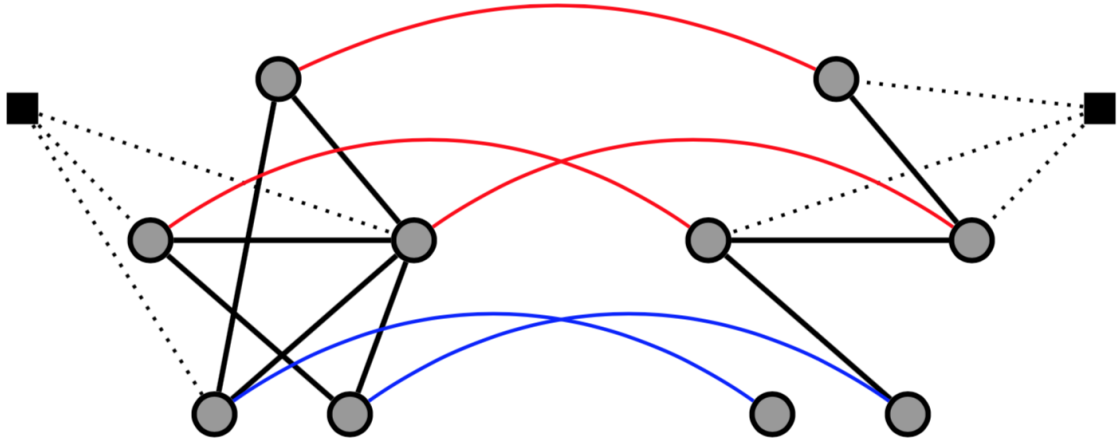


- VI solves **inference** with **optimization**.
- Posit a **variational family** of distributions over the latent variables,

$$q(\mathbf{z}; \boldsymbol{\nu})$$

- Fit the **variational parameters** $\boldsymbol{\nu}$ to be close (in KL) to the exact posterior.

Inference



All `Inference` has (at least) two inputs:

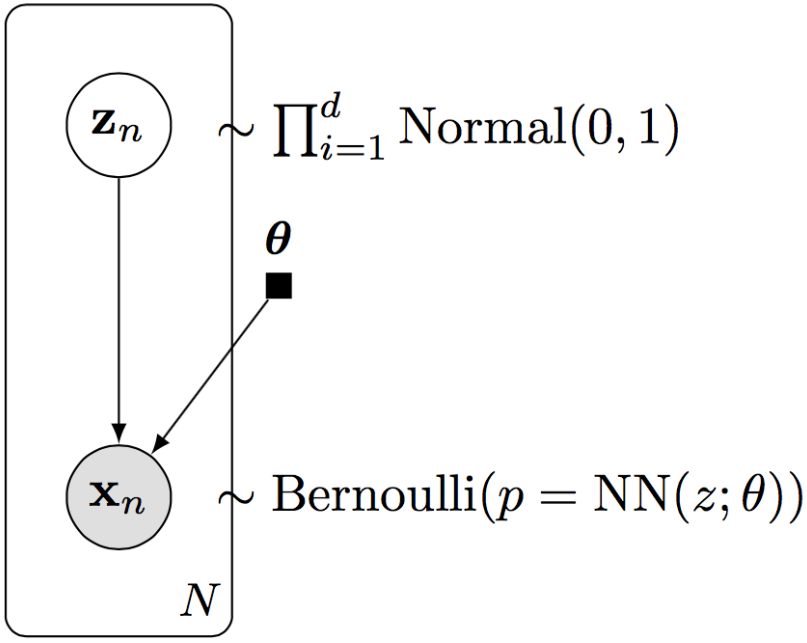
1. red aligns latent variables and posterior approximations;
2. blue aligns observed variables and realizations.

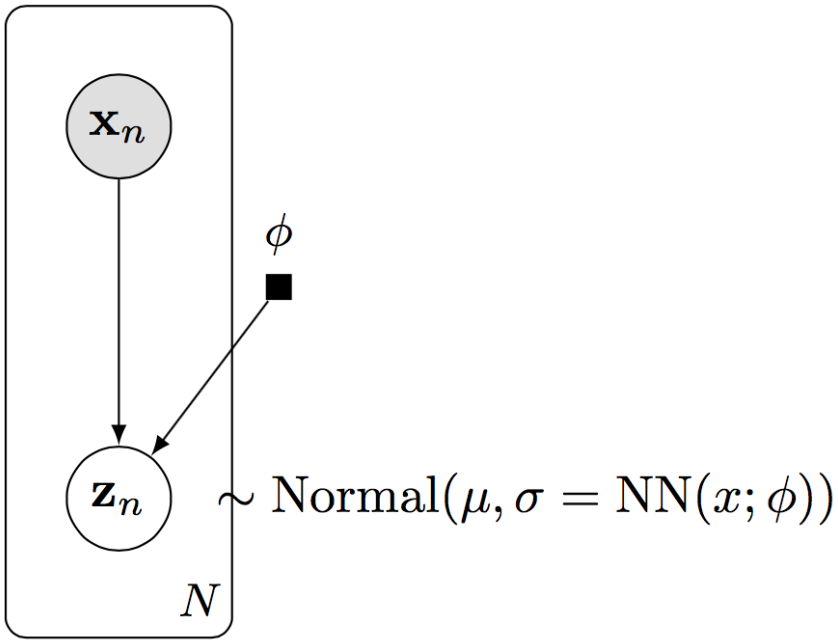
```
inference = ed.Inference({beta: qbeta, z: qz}, data={x: x_train})
```

`Inference` has class methods to finely control the algorithm. Edward is fast as handwritten TensorFlow at runtime.

edwardlib.org/api

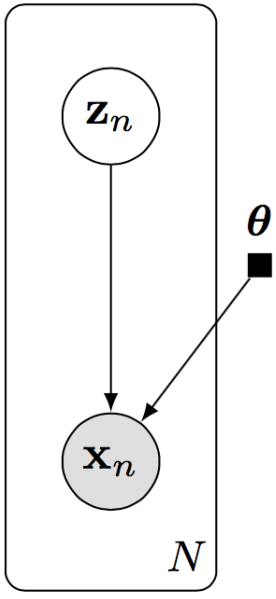
Example: Variational Auto-Encoder for Binarized MNIST



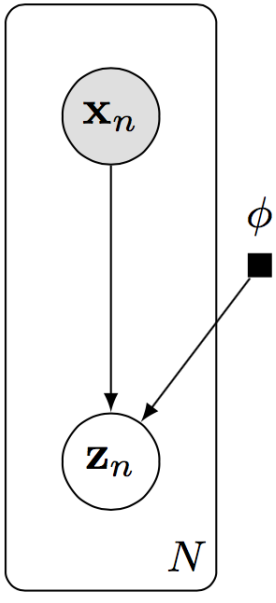


[Kingma & Welling 2014; Rezende+ 2014]

Example: Variational Auto-Encoder for Binarized MNIST




```
# Probabilistic model
z = Normal(loc=tf.zeros([N, d]), scale=tf.ones([N, d]))
h = Dense(256, activation='relu')(z)
x = Bernoulli(logits=Dense(28 * 28, activation=None)(h))
```




```
# Variational model
qx = tf.placeholder(tf.float32, [N, 28 * 28])
qh = Dense(256, activation='relu')(qx)
qz = Normal(loc=Dense(d, activation=None)(qh),
            scale=Dense(d, activation='softplus')(qh))
```

[Kingma & Welling 2014; Rezende+ 2014]

Example: Variational Auto-Encoder for Binarized MNIST

[Demo]

Inference

Variational inference. It uses a variational model.

```
qbeta = Normal(loc=tf.Variable(tf.zeros([K, D])),
               scale=tf.exp(tf.Variable(tf.zeros([K, D]))))
qz = Categorical(logits=tf.Variable(tf.zeros([N, K])))

inference = ed.VariationalInference({beta: qbeta, z: qz}, data={x: x_train})
```

Monte Carlo. It uses an Empirical approximation.

```
T = 10000  # number of samples
qbeta = Empirical(params=tf.Variable(tf.zeros([T, K, D])))
qz = Empirical(params=tf.Variable(tf.zeros([T, N])))

inference = ed.MonteCarlo({beta: qbeta, z: qz}, data={x: x_train})
```

Conjugacy & exact inference. It uses symbolic algebra on the graph.

Advanced Topics

1. **Non-Bayesian inference.** Maximum likelihood, divergence minimization.
2. **Model-specific inference.** Conjugacy (e.g., Gibbs sampling), exact inference.
3. **Likelihood-free inference.** Implicit models, generative adversarial networks, approximate Bayesian computation.
4. **Composable inference.** Message passing algorithms (e.g., expectation propagation), hybrid algorithms (e.g., Monte Carlo EM).
5. **Designing new inferences.** Object-oriented inheritance.

Current Work

Dynamic Graphs

PROB TORCH

Probabilistic Torch is library for deep generative models that extends PyTorch. It is similar in spirit and design goals to Edward and Pyro, sharing many design characteristics with the latter.

The design of Probabilistic Torch is intended to be as PyTorch-like as possible. Probabilistic Torch models are written just like you would write any PyTorch model, but make use of three additional constructs:

Distributed, Compiled, Accelerated Systems

Probabilistic programming over multiple machines. XLA compiler optimization and TPUs. More model-specific inference.

Distributions Backend

```
def pixelcnn_dist(params, x_shape=(32, 32, 3)):
  def _logit_func(features):
    # single autoregressive step on observed features
    logits = pixelcnn(features)
    return logits
  logit_template = tf.make_template("pixelcnn", _logit_func)
  make_dist = lambda x: tfd.Independent(tfd.Bernoulli(logit_template(x)))
  return tfd.Autoregressive(make_dist, tf.reduce_prod(x_shape))

x = pixelcnn_dist()
loss = -tf.reduce_sum(x.log_prob(images))
train = tf.train.AdamOptimizer().minimize(loss)  # run for training
generate = x.sample()  # run for generation
```

TensorFlow Distributions consists of a large collection of distributions. `Bijector` enable efficient, composable manipulation of probability distributions.

Pytorch PPLs are consolidating on a backend for distributions.

[Dillon+ 2017]

References

edwardlib.org

- Edward: A library for probabilistic modeling, inference, and criticism.
  arXiv preprint arXiv:1610.09787, 2016.
- Deep probabilistic programming.
  International Conference on Learning Representations, 2017.
- TensorFlow Distributions.
  arXiv preprint arXiv:1711.10604, 2017.