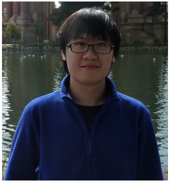
An Overview of Edward:
A Probabilistic Programming System

Dustin Tran
Columbia University

Alp Kucukelbir

Adji Dieng



Dawen Liang

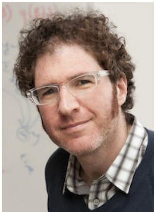
Eugene Brevdo

Maja Rudolph

Matt Hoffman

Rajesh Ranganath

Andrew Gelman



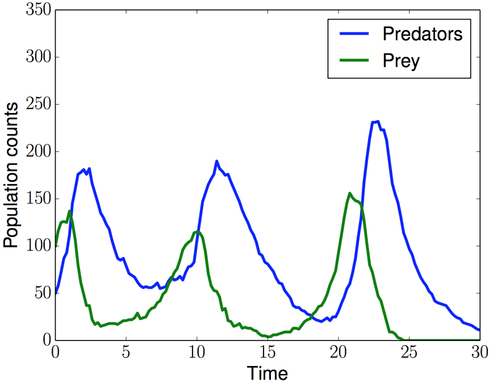
David Blei

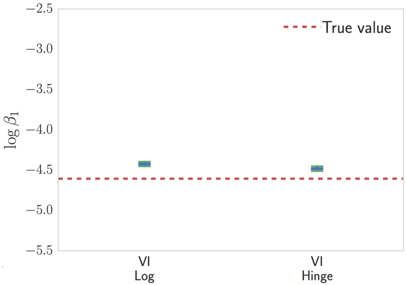
Kevin Murphy

Rif Saurous

Exploratory analysis of 1.7M taxi trajectories, in Stan

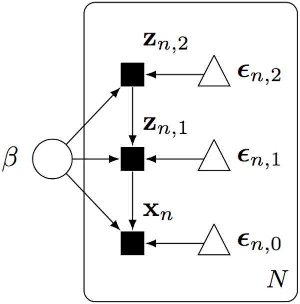
[Kucukelbir+ 2017]

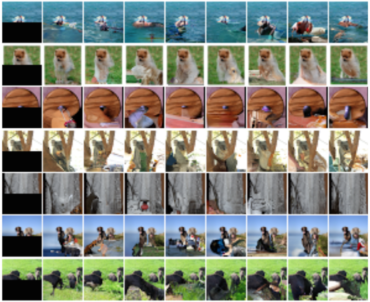




Simulators of 100K time series in ecology, in Edward

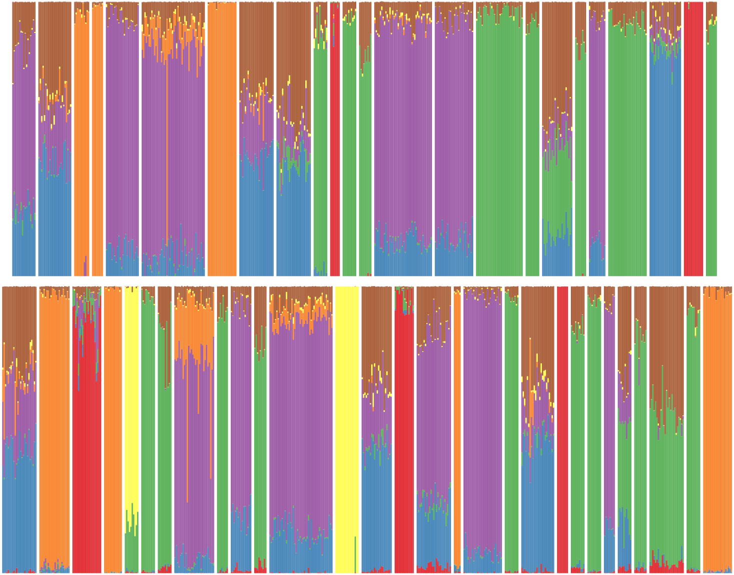
[Tran+ 2017]





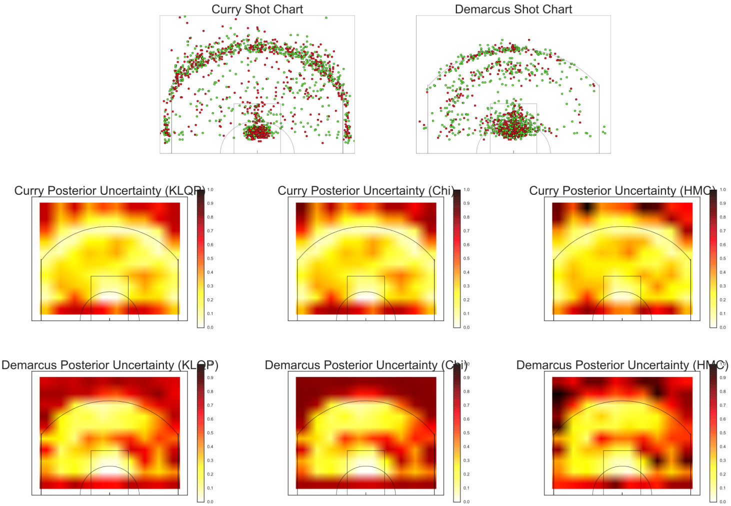
Generation & compression of 10M colored 32x32 images, in Edward

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



Cause and effect of 1.6B genetic measurements, in Edward

[in preparation; 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 $\mathbf{x}$,

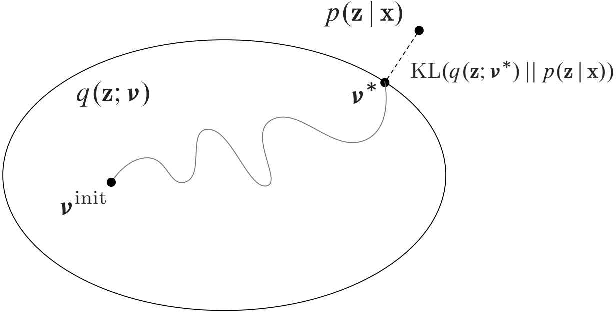
$$p(\mathbf{z}, \mathbf{x}).$$

- 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**.

Variational inference



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

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

- Fit the **variational parameters** ν to be close (in KL) to the exact posterior.

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.

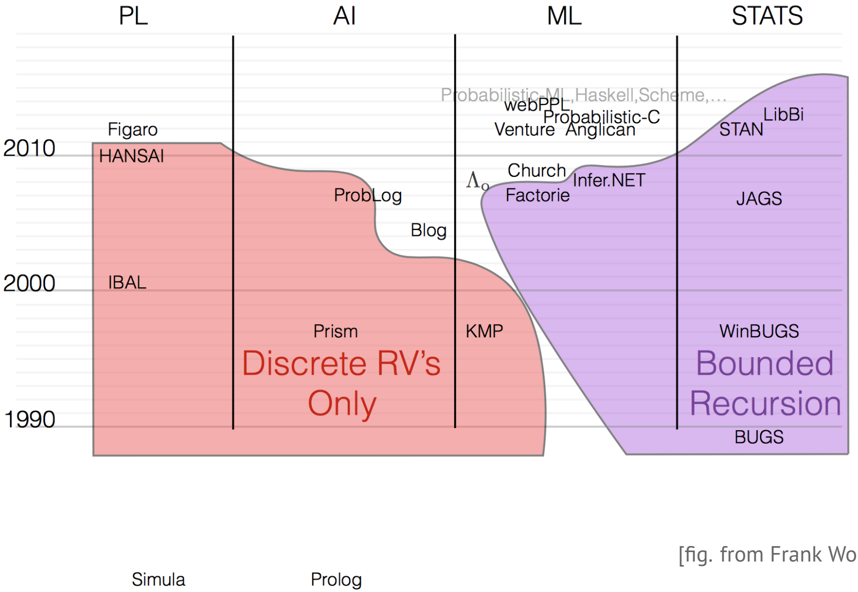
- Any computable inference algorithm.

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

- Any computable application.

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

Languages and Systems



[fig. from Frank Wood]

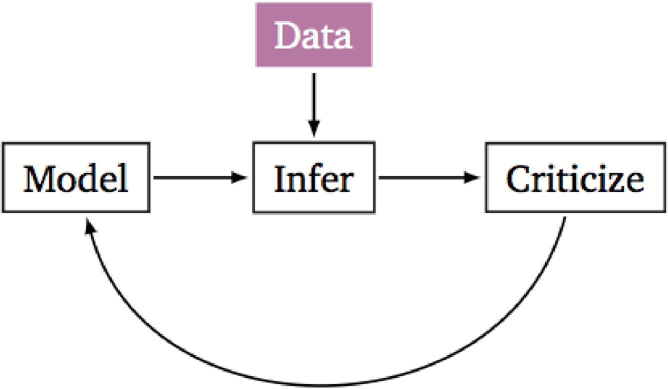
George E.P. Box (1919 - 2013)

An iterative process for science:

1. Build a model of the science
2. Infer the model given data
3. Criticize the model given data

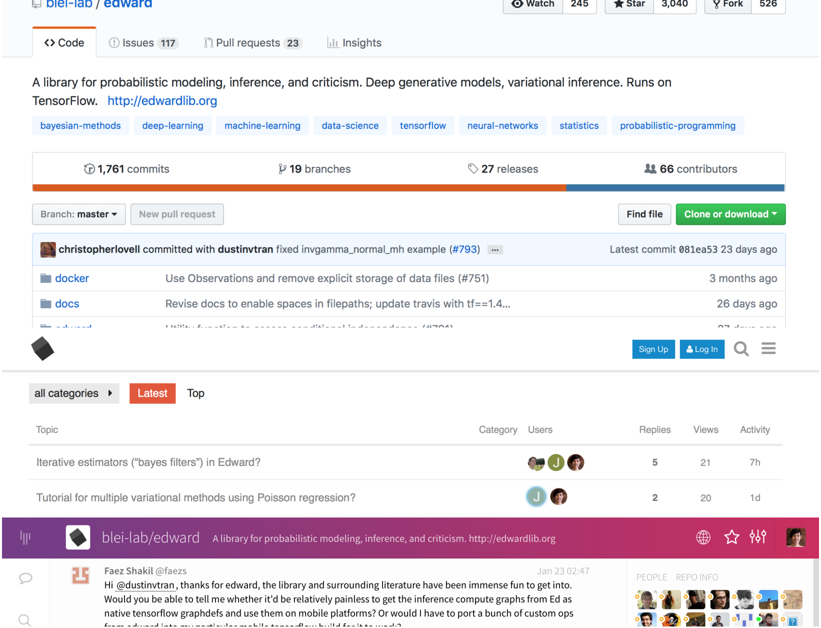
[Box & Hunter 1962, 1965; Box & Hill 1967; Box 1976, 1980]

Box's Loop



Edward is a library designed around this loop.

[Box 1976, 1980; Blei 2014]



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 $\mathbf{x}$ 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]

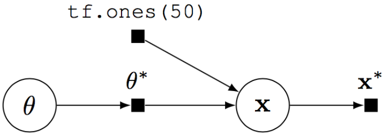
Example: Beta-Bernoulli

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 $\mathbf{x} \in \{0, 1\}^{50}$.

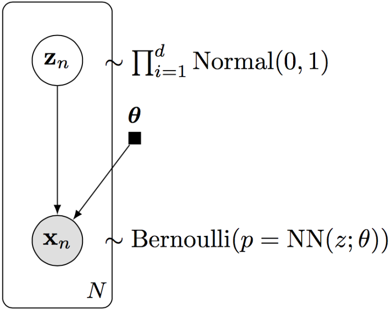
```
theta = Beta(1.0, 1.0)
x = Bernoulli(probs=tf.ones(50) * theta)
```

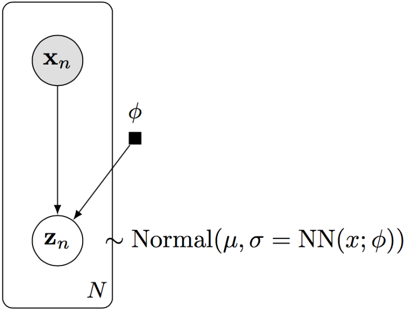


Fetching **x** from the graph generates a binary vector of 50 elements.

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

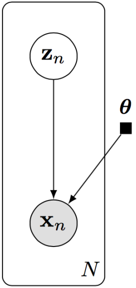
Example: Variational Auto-Encoder for Binarized MNIST



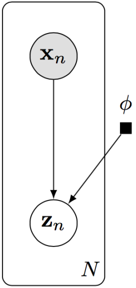


[Kingma & Welling 2014; Rezende+ 2014]

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))
```



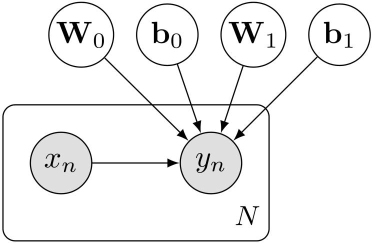

```
# Variational model
qx = tf.placeholder(tf.float32, [N, 28 * 28])
qh = Dense(256, activation='relu')(qx)
qz = Normal(loc=Dense(d, activation=None)(qh),
            scale=Dense(d, activation='softplus')(qh))
```

[Kingma & Welling 2014; Rezende+ 2014]

Example: Variational Auto-Encoder for Binarized MNIST

[Demo]

Example: Bayesian neural network for classification

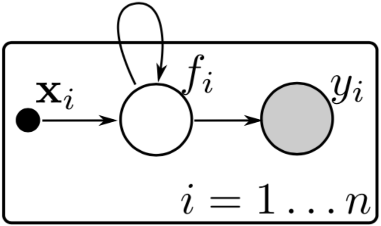



```
W_0 = Normal(mu=tf.zeros([D, H]), sigma=tf.ones([D, H]))
W_1 = Normal(mu=tf.zeros([H, 1]), sigma=tf.ones([H, 1]))
b_0 = Normal(mu=tf.zeros(H), sigma=tf.ones(L))
b_1 = Normal(mu=tf.zeros(1), sigma=tf.ones(1))

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: 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 $\mathbf{z}, \beta$.

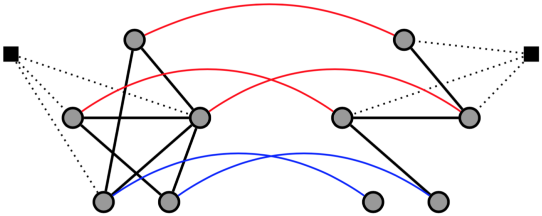
Goal

- Calculate posterior distribution

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

This is the key problem in Bayesian inference.

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

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.

Inference: Composing Inference

Core to Edward's design is that inference can be written as a collection of separate inference programs.

For example, here is variational EM.

```
qbeta = PointMass(params=tf.Variable(tf.zeros([K, D])))
qz = Categorical(logits=tf.Variable(tf.zeros[N, K]))

inference_e = ed.VariationalInference({z: qz}, data={x: x_data, beta: qbeta})
inference_m = ed.MAP({beta: qbeta}, data={x: x_data, z: qz})

for _ in range(10000):
  inference_e.update()
  inference_m.update()
```

We can also write message passing algorithms, which work over a collection of local inference problems. This includes expectation propagation.

[Neal & Hinton 1993; Minka 2001; Gelman+ 2017]

Non-Bayesian Methods: GANs

GANs posit a generative process,

$$
\begin{aligned}
\epsilon &\sim \text{Normal}(0, 1) \\
\mathbf{x} &= G(\epsilon; \theta)
\end{aligned}
$$

for some generative network G.

Training uses a discriminative network D via the optimization problem

$$
\min_{\theta} \max_{\phi} \mathbb{E}_{p^*(\mathbf{x})}[\log D(\mathbf{x}; \phi)] + \mathbb{E}_{p(\mathbf{x};\theta)}[\log(1 - D(\mathbf{x}; \phi))]
$$

The generator tries to generate samples indistinguishable from true data.

The discriminator tries to discriminate samples from the generator and samples from the true data.

[Goodfellow+ 2014]

Example: Generative Adversarial Network for MNIST

[Demo]

http://edwardlib.org/tutorials/gan

Non-Bayesian Methods: GANs

```
def generative_network(eps):
  h = Dense(256, activation='relu')(eps)
  return Dense(28 * 28, activation=None)(h)

def discriminative_network(x):
  h = Dense(28 * 28, activation='relu')(x)
  return Dense(h, activation=None)(1)

# Probabilistic model
eps = Normal(loc=tf.zeros([N, d]), scale=tf.ones([N, d]))
x = generative_network(eps)

inference = ed.GANInference(data={x: x_train},
    discriminator=discriminative_network)
inference.run()
```

[Goodfellow+ 2014]

Non-Bayesian Methods: GANs

```
def generative_network(eps):
  h = Dense(256, activation='relu')(eps)
  return Dense(28 * 28, activation=None)(h)

def discriminative_network(x):
  h = Dense(28 * 28, activation='relu')(x)
  return Dense(h, activation=None)(1)

# Probabilistic model
eps = Normal(loc=tf.zeros([N, d]), scale=tf.ones([N, d]))
x = generative_network(eps)

inference = ed.WGANInference(data={x: x_train},
    discriminator=discriminative_network)
inference.run()
```

[Arjovsky+ 2017; Gulrajani+ 2017]

Current Work

Dynamic Graphs

PYRO

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:

Distributions Backend

```python
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]

Distributed, Compiled, Accelerated Systems

Probabilistic programming over multiple machines. XLA compiler optimization and TPUs. More flexible programmable inference.

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.