Everything You Didn’t Know About Edward

Dustin Tran
Columbia University

Basics

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.

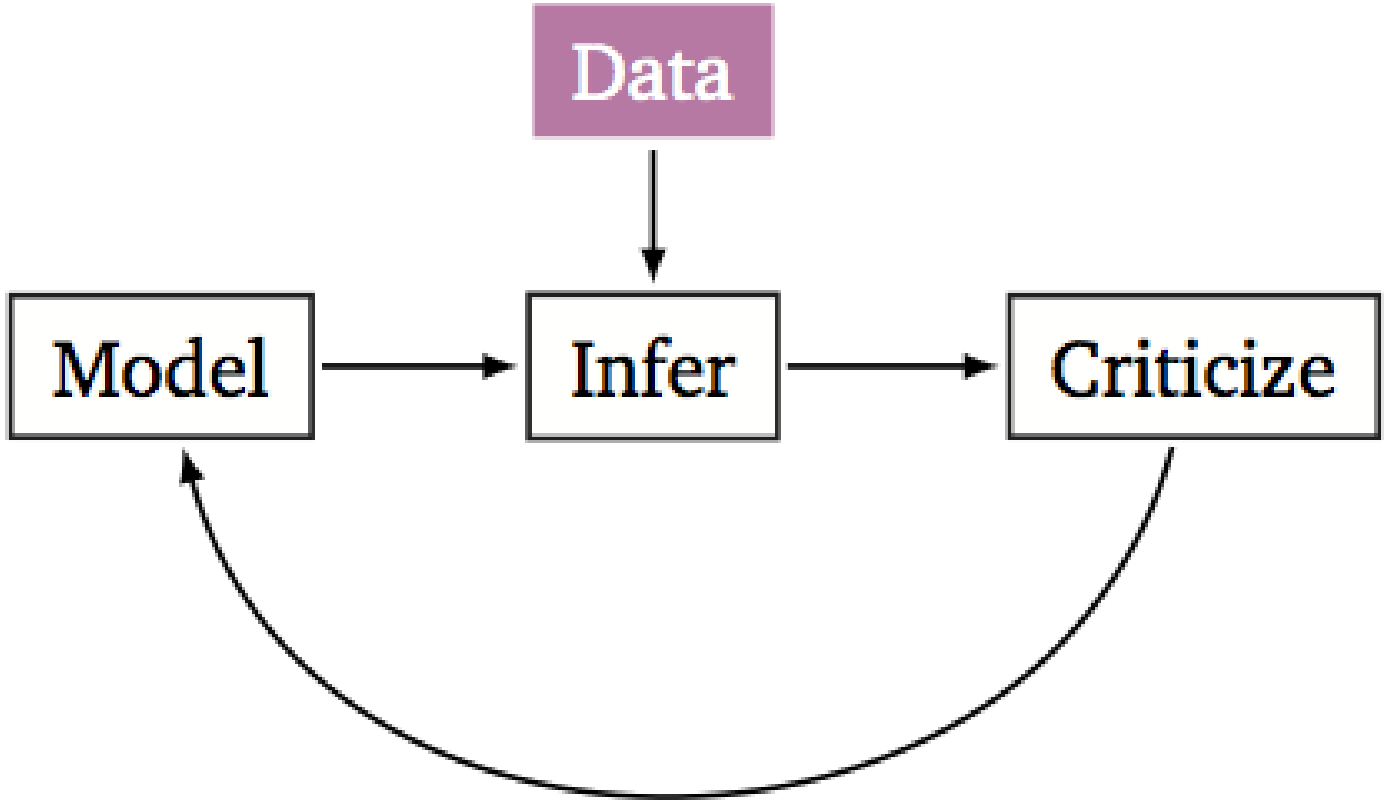
- Any computable inference algorithm.

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

- Any computable application.

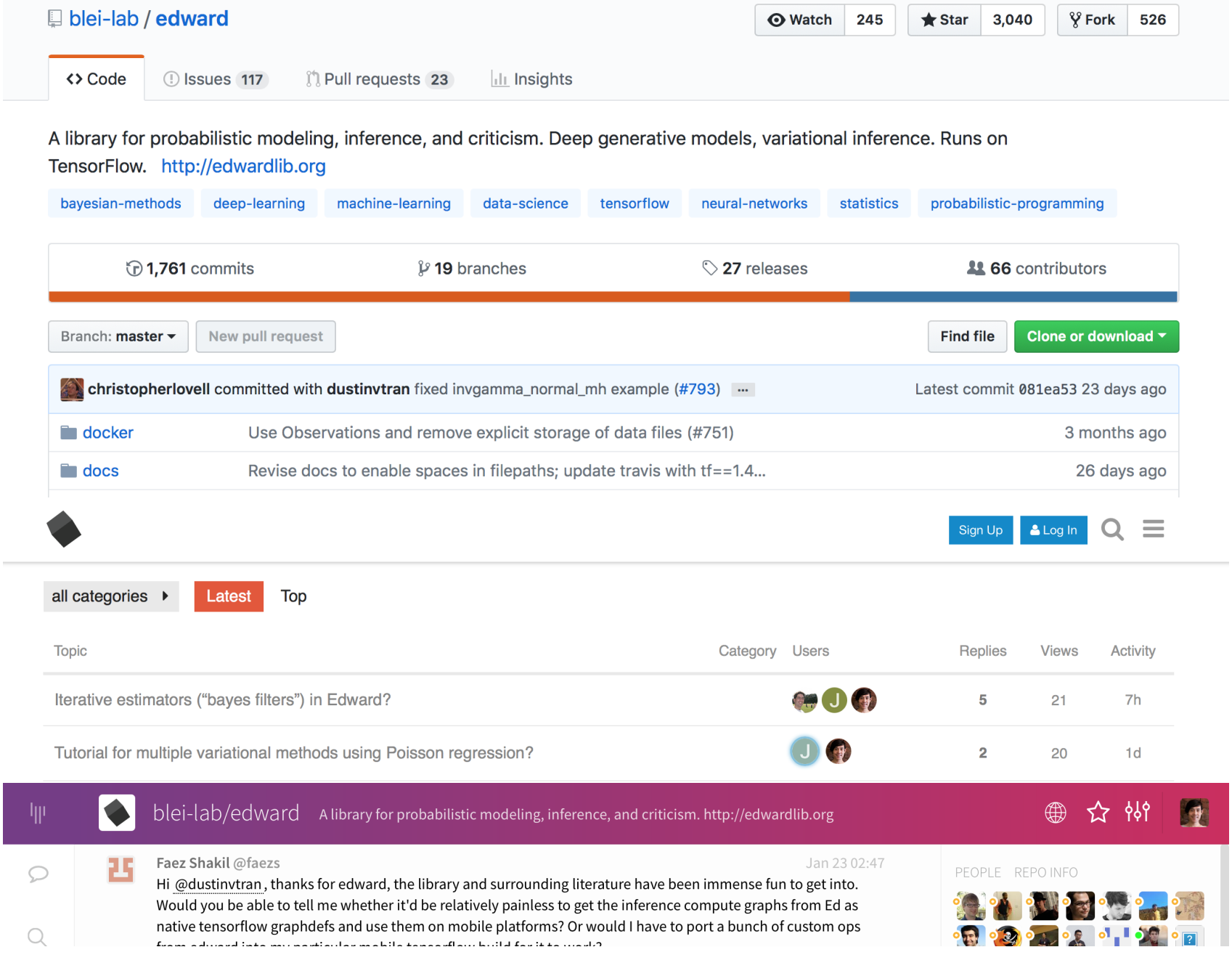
  ex. exploratory analysis; object recognition; code generation; causality.

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.

What impact has Edward had?

Designs

- Edward: A library for probabilistic modeling, inference, and criticism. arXiv, 2016.
- Deep probabilistic programming. ICLR, 2017.
- Book chapter. arXiv next week, 2017.

Applications

- The variational Gaussian process. ICLR, 2016.
- Hierarchical variational models. ICML, 2016.
- Exponential family embeddings. NIPS, 2016.
- Deep and hierarchical implicit models. NIPS, 2017.
- Variational inference via χ-upper bound minimization. NIPS, 2017.
- Causal effect inference with deep latent-variable models. NIPS, 2017.
- Implicit causal models for genome-wide association studies. arXiv, 2017
- Feature-matching auto-encoders. OpenReview, 2017

Language: Computational Graphs w/ Random Variables

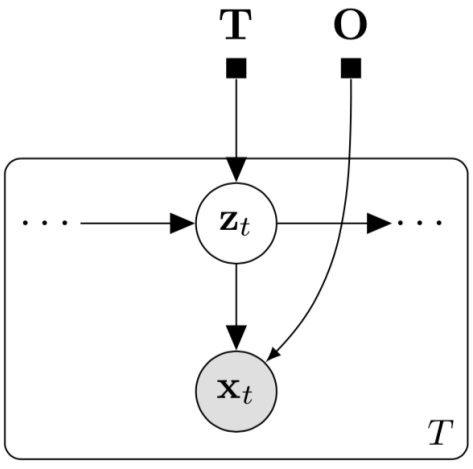
Edward's language augments computational graphs with an abstraction for random variables. Each random variable **x** is associated to a tensor $\mathbf{x}^*$, $\mathbf{x}^* \sim p(\mathbf{x} \mid \theta^*)$.

```
# 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.

Language Example

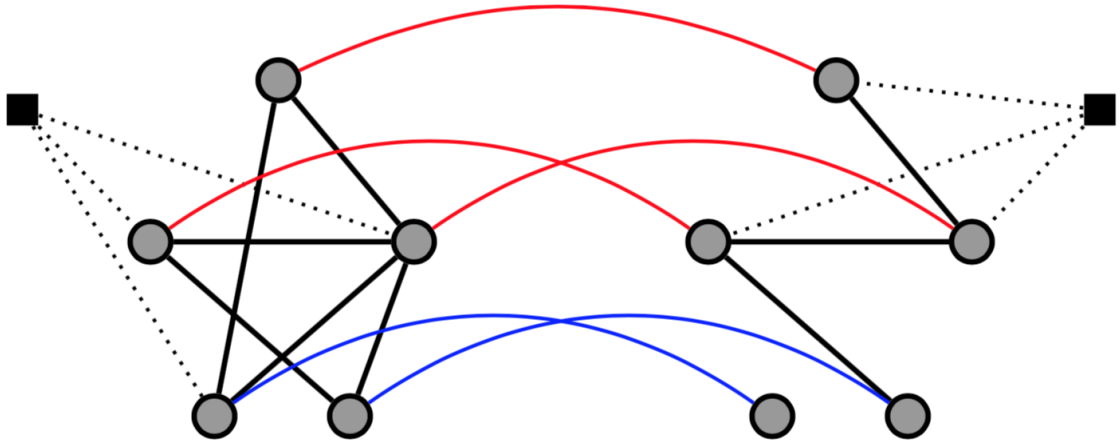



```
1  from edward.models import Normal
2
3  T = tf.Variable(tf.random_normal([K, K]))
4  O = tf.Variable(tf.random_normal([K, D]))
5  z = x = []
6  z[0] = Normal(loc=tf.zeros(K), scale=tf.ones(K))
7  x[0] = Normal(loc=tf.matmul(z[0], O), scale=0.5)
8  for t in range(1, T):
9    z[t] = Normal(loc=tf.matmul(z[t - 1], T), scale=0.1)
10   x[t] = Normal(loc=tf.matmul(z[t], O), scale=0.5)
```

Figure 6: State space model for real-valued sequences $\mathbf{x} = [\mathbf{x}_1, \ldots, \mathbf{x}_T] \in \mathbb{R}^{T \times D}$: **(left)** graphical model; **(right)** probabilistic program. $\mathbf{x}$ and $\mathbf{h}$ are written as a Python list of vector-dimensional random variables.

Edward's language enables a *calculus* on random variables.

Inference as Stochastic Graph Optimization



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 == handwritten TensorFlow at runtime

Probabilistic programming system	Runtime (s)
Handwritten NumPy (1 CPU)	534
Stan (1 CPU) [Carpenter+ 2016]	171
PyMC3 (12 CPU) [Salvatier+ 2015]	30.0
Edward (12 CPU)	**8.2**
Handwritten TensorFlow (GPU)	5.0
Edward (GPU)	**4.9** (35x faster than Stan)

Run HMC for 100 iterations and fixed hyperparameters.

Bayesian logistic regression for Covertype (581012 data points, 54 features).

12-core Intel i7-5930K CPU at 3.50GHz and NVIDIA Titan X (Maxwell) GPU. Single precision.

Edward is orders of magnitude faster than existing software for large data.

[Tran+ 2017]

Composable & Hybrid Inference

```
n_samples = n_iter_per_epoch * n_epoch
qz = Empirical(tf.Variable(tf.random_normal([n_samples, mnist.train.size, K])))

inference_e = ed.SGHMC({z: qz}, data={x: x_train})
inference_e.initialize()

inference_m = ed.MAP(data={x: x_train, z: qz.params[inference_e.t]})
optimizer = tf.train.RMSPropOptimizer(1e-2, epsilon=1.0)
inference_m.initialize(optimizer=optimizer)
```

(a) Change from variational EM to EM with stochastic gradient Hamiltonian Monte Carlo (Chen et al., 2014).

```
inference_z1 = ed.KLpq({beta: qbeta, z1: qz1}, {x1: x1_train})
inference_z2 = ed.KLpq({beta: qbeta, z2: qz2}, {x2: x2_train})
...
for _ in range(10000):
  inference_z1.update()
  inference_z2.update()
```

edwardlib.org/api/inference-compositionality

Non-Bayesian Inference

```
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()
```

edwardlib.org/api/inference-classes

Some Cool Ideas

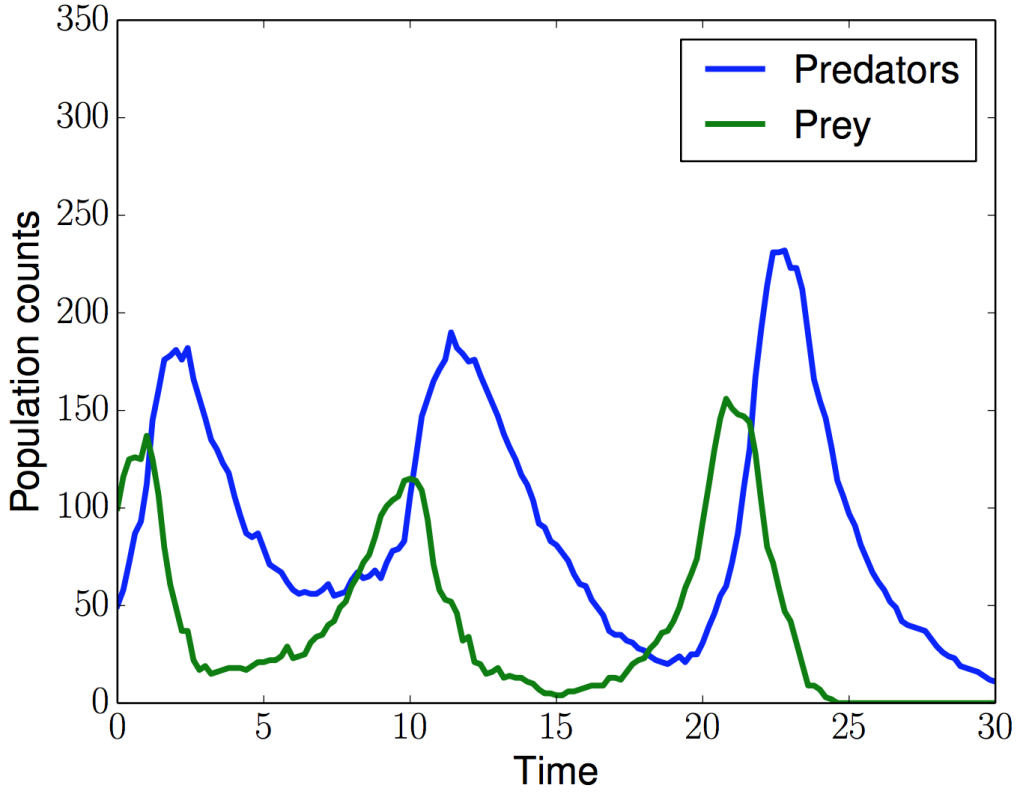
Against the “Program -> Query” Modality

1. Unlike other PPLs, **Edward has no explicit “model” or “inference” block**: A model is simply a collection of random variables in the graph. Inference specifies how to modify parameters in that collection subject to another.

   This offers significant flexibility: we can *infer only parts of a model* (e.g., layer-wise training), *infer parts used in multiple models* (e.g., multi-task learning), or *plug in a posterior into a new model* (e.g., Bayesian updating).

2. Unlike other PPLs, **Edward has no `observe` operator**, which is called before an `infer` operation. Each `Inference` can specify its own alignment of random variables and observations.

Implicit Models (& Likelihood-Free Inference)




```
beta = LogNormal(tf.zeros(3), tf.ones(3))
x1 = []  # prey population
x2 = []  # predator population
for t in range(1000):
  x1[t+1] = beta[0] * x1[t] - beta[1] * x1[t] * x2[t] + Normal(0.0, 10.0)
  x2[t+1] = -beta[1] * x2[t] + beta[1] * x1[t] * x2[t] + Normal(0.0, 10.0)
```

`tf.Tensors` implicitly carry densities. We can infer (approximate) them via `Inference`.

Dynamic Stochasticity

Parameter shapes can be stochastic.

```
n = Poisson(5.0)
x = tf.ones([tf.cast(n, tf.int32)])
```

Event shapes can be stochastic.

```
# Prior on size of Dirichlet.
n = 1 + tf.cast(Exponential(0.5), tf.int32)

# Build a vector of ones whose size is n; multiply it by alpha.
p = Dirichlet(tf.ones([n]) * alpha)
```

Control flow can be stochastic.

```
def geometric(p):
  i = tf.constant(0)
  sample = tf.while_loop(
      cond=lambda i: tf.cast(1 - Bernoulli(probs=p), tf.bool),
      body=lambda i: i + 1,
      loop_vars=[i])
  return sample
```

No recursion. Must be rewritten as a (stochastic) while loop. We implemented a DirichletProcess this way.

Taxonomy of Inference

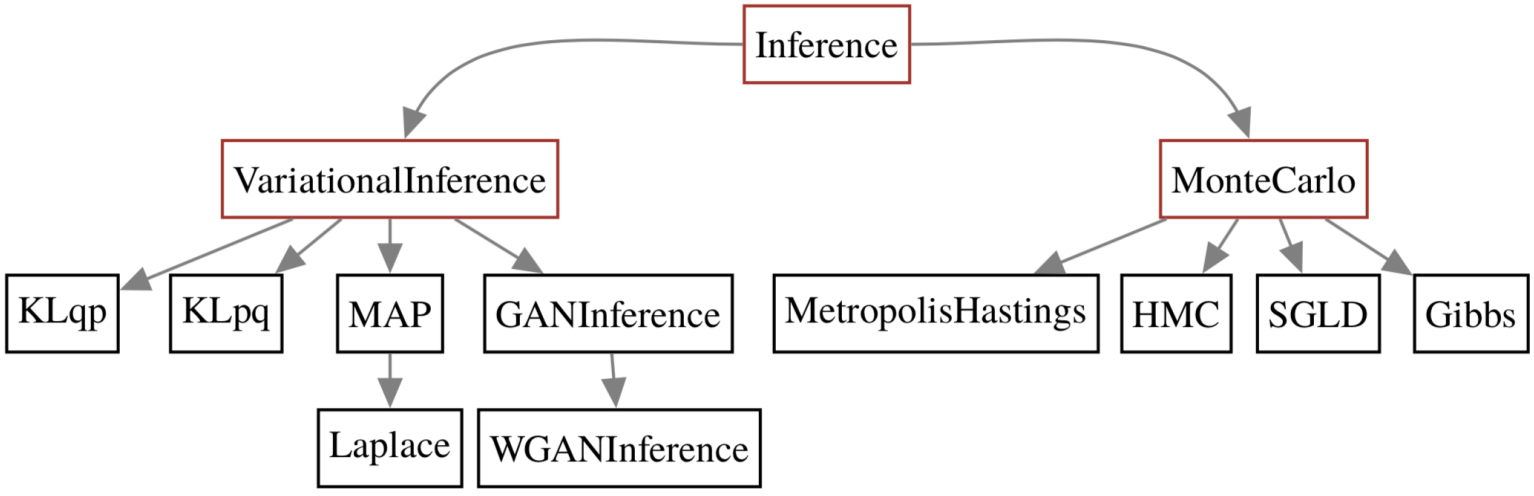


Figure 9: Taxonomy of inference. It is a dependency graph where nodes are classes in Edward (red denotes abstract classes) and arrows denote class inheritance. Currently, the hierarchy is at most four levels. Additional algorithms on the third and fourth levels are omitted for space.

Compiler Connections

Recall Edward interprets inference as stochastic graph optimization.

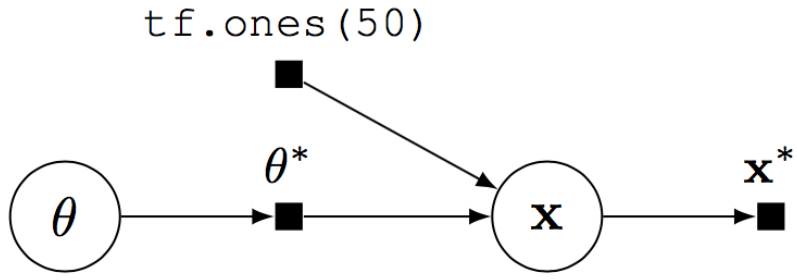
This may have deep implication to extend compiler optimization with probability(?). Examples:

1. **Operation fusion to marginalization.** Fusing two stochastic nodes into one corresponds to computing the marginal density. (E.g., turn naive Monte Carlo sample to exact normal-normal compound density.)
2. **Common subexpression elimination to common distribution elimination.** Suppose a graph involves compiling two subgraphs which are approximately equal in distributions. Avoid compiling them separately.
3. **Amortized inference as ahead-of-time compilation.** Training can amortize inference over data generated from the model; no data need be involved until runtime.

These are half-baked, experimental ideas. Worth more investigation.

Exploiting Graphs: Conditional Independence

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



Edward writes all programs as part of the computational graph. This lets us exploit graph structure. One powerful application is conditional independence.

- Edward can query a random variable for its parents, ancestors, children, and Markov blanket.
- Edward uses Bayes-Ball to assess conditional independence of two nodes given a third set of nodes.
- Operators for causal graphs (e.g., backdoor criterion) are also possible.

Why is this important? Conditional independence is fundamental to distributed inference by enabling parallel, concurrent computation.

Exploiting Graphs: Symbolic Algebra

We can exploit graphs not just in the graph structure but in the symbolic operations.

This enables a `complete_conditional` function. Basically it:

1. Take a random variable's Markov blanket; then take the log joint.
2. Simplify the log joint into a set of sufficient statistic and natural parameter pairs.
3. Perform a table lookup to find the distribution corresponding to those sufficient statistics.

Why is this important? It is core to Edward's `Gibbs` and VI with natural gradients.

Limitations

- **Graph copying** is a key function to all Edward's inferences. It's a cognitive PITA, carries graph junk, and hides computation from users.
- **Object-oriented inference** has a high cognitive burden. Arguably no one except me understands it enough to exploit.
- Certain models, such as autoregressive distributions, are easier to **build the loss** (log prob) than it is to build the generative program.
- **Programmable inference is hard**. Matt and I thought hard about use cases. But we didn't cover all of them. Ex: gradient clipping, registering specific inference ops to devices.

Questions

- Edward's biggest competitor is not an existing PPL, but TensorFlow itself.

  Rather than start from universal PPLs and bend backwards to support programmable inference, can we take a **computation-first** approach?

- Edward is limited to single machine.

  How do we scale probabilistic programming to **multi-machine** environments with billions of data points?

- Our **symbolic algebra** works on only toy problems and requires TensorFlow graph introspection.

  Can we pursue this at the XLA graph level?

- Our ICLR paper proposed a **model zoo**. The idea is to go beyond Stan's model examples to also include inference algs, tuned hyperparameters, and pre-training. It never took off.

  How do we make it usable?