Deep Probabilistic Programming

Dustin Tran
Columbia University

Matt Hoffman
Adobe Research

Kevin Murphy
Google Research

Eugene Brevdo
Google Brain

Rif A. Saurous
Google Research

David M. Blei
Columbia University

Abstract

We propose Edward, a new Turing-complete probabilistic programming language which builds on two compositional representations—random variables and inference. We show how to integrate our language into existing computational graph frameworks such as TensorFlow; this provides significant speedups over existing probabilistic systems. We also show how Edward makes it easy to fit the same model using a variety of composable inference methods, ranging from point estimation, to variational inference, to MCMC. By treating inference as a first class citizen, on a par with modeling, we show that probabilistic programming can be as computationally efficient and flexible as traditional deep learning. For example, we show how to reuse the modeling representation within inference to design rich variational models and generative adversarial networks.

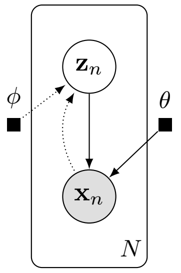
1 Introduction

Deep neural networks have become popular in large part due to their compositional nature. This lets users mix and match layers in novel creative ways, without having to worry about how to perform testing (just use forward propagation) or inference (just use a generic gradient-based optimizer, combined with back propagation and automatic differentiation).

In this paper, we aim to design compositional representations for probabilistic programming. Most previous work has focused on how to build rich probabilistic programs by composing random variables (Goodman et al., 2012; Ghahramani, 2015; Lake et al., 2016). Less work has considered an analogous compositionality for inference. In fact, most existing probabilistic programming languages treat the inference engine as a black box, abstracted away from the model. Such systems cannot capture recent advances in probabilistic modeling such as in variational inference (Kingma and Welling, 2014; Rezende and Mohamed, 2015; Tran et al., 2016b) and generative adversarial networks (Goodfellow et al., 2014). This is because they require reuse of the modeling representation to construct rich variational models and discriminative networks during inference.

We propose Edward[1], a new Turing-complete probabilistic programming language which builds on two compositional representations—random variables and inference. We show how to integrate our language into existing computational graph frameworks such as TensorFlow (Abadi et al., 2016). By leveraging such frameworks, we get distributed training, parallelism, vectorisation, and GPU support "for free". We also show how Edward makes it easy to fit the same model using a variety of composable inference methods, ranging from point estimation, to variational inference, to MCMC. By treating inference as a first class citizen, on a par with modeling, we show that probabilistic programming can be as computationally efficient and flexible as traditional deep learning.

[1]Available at `http://edwardlib.org`. See Tran et al. (2016a) for details of the API. This paper focuses on the algorithmic foundations of Edward; a longer version will be on the arXiv shortly.



```python
# Probabilistic model
z = Normal(mu=tf.zeros([N, d]), sigma=tf.ones([N, d]))
h = slim.fully_connected(z, 256)
x = Bernoulli(logits=slim.fully_connected(h, 28 * 28, activation_fn=None))

# Variational model
qx = tf.placeholder(tf.float32, [N, 28 * 28])
qh = slim.fully_connected(qx, 256)
qz = Normal(mu=slim.fully_connected(qh, d, activation_fn=None),
            sigma=slim.fully_connected(qh, d, activation_fn=tf.nn.softplus))
```

Figure 1: Variational auto-encoder for a data set of 28×28 pixel images: (left) graphical model, with dotted lines for the inference model; (right) probabilistic program, with 2-layer neural networks.

2 Compositional Representations for Probabilistic Models

We define random variables as a key compositional representation. They are class objects with methods, for example, to compute the log density and to sample. Further, each random variable $\mathbf{x}$ is associated to a tensor $\mathbf{x}^*$ in the computational graph, which represents a single sample $\mathbf{x}^* \sim p(\mathbf{x})$. This association embeds the random variable into the computational graph.

This design is conceptually simple, making it easy to develop probabilistic programs in a computational graph framework. Importantly, all computation is represented on the graph. This makes it easy to parameterize random variables with complex deterministic structure, such as with deep neural networks and a diverse set of math operations. The design also enables compositions of random variables to capture complex stochastic structure.

With computational graphs, it is also natural to build mutable states within the probabilistic program. As a typical use of computational graphs, such states can define model parameters; in TensorFlow, this is given by a `tf.Variable`. Another use case is for building discriminative models $p(\mathbf{y} \mid \mathbf{x})$, where $\mathbf{x}$ are features that are input as training or test data. The program can be written independent of the data, using a mutable state (`tf.placeholder`) for $\mathbf{x}$ in its graph. During training and testing, we feed the placeholder the appropriate values. In Appendix A.1, we demonstrate this with a Bayesian neural network for classification. We give other examples below.

2.1 Example: Variational Auto-encoder

Figure 1 implements a variational auto-encoder (VAE) (Kingma and Welling, 2014; Rezende et al., 2014) in Edward. There are N data points $\{x_n\}$ and d latent variables per data point $\{z_n\}$. The program uses TensorFlow Slim (Guadarrama and Silberman, 2016) to define the neural networks. The probabilistic model is parameterized by a 2-layer neural network, with 256 hidden units (and ReLU activation), and generates 28×28 pixel images. The variational model is parameterized by a 2-layer inference network, with 256 hidden units and outputs parameters of a normal posterior approximation.

The probabilistic program is concise. Importantly, core elements of the VAE—such as its distributional assumptions and neural net architectures—are all extensible. Model compositionality enables it to be embedded into more complicated models (Gregor et al., 2015; Rezende et al., 2016) and for other learning tasks (Kingma et al., 2014). Inference compositionality (which we discuss later) enables it to be embedded into more complicated algorithms, such as with expressive variational approximations (Rezende and Mohamed, 2015; Tran et al., 2016b; Kingma et al., 2016) and alternative objectives (Ranganath et al., 2016; Li and Turner, 2016; Dieng et al., 2016).

2.2 Stochastic Control Flow and Model Parallelism

Random variables can also be integrated with control flow, enabling probabilistic programs with stochastic control flow. Stochastic control flow defines dynamic conditional dependencies, known in the literature as contingent or existential dependencies (Mansinghka et al., 2014; Wu et al., 2016). See Figure 2, where $\mathbf{x}$ may or may not depend on $\mathbf{a}$ for a given execution.

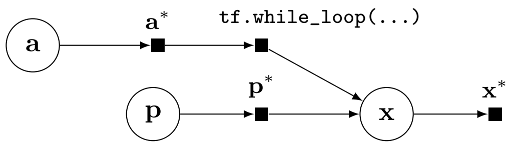


Figure 2: Computational graph for a probabilistic program with stochastic control flow.

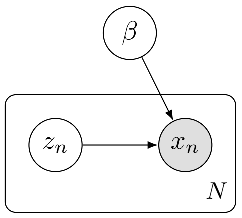
We use stochastic control flow to implement a Dirichlet process mixture model in Appendix A.2. Stochastic control flow produces difficulties for algorithms that leverage the graph structure; the relationship of conditional dependencies changes across execution traces. Importantly, the computational graph provides an elegant way of teasing out static conditional dependence structure ($\mathbf{p}$) from dynamic dependence structure ($\mathbf{a}$). We can perform model parallelism over the static structure with GPUs and batch training, and use generic computations to handle the dynamic structure.

3 Compositional Representations for Inference

We have described random variables, a representation for building rich probabilistic programs over computational graphs. We now describe a compositional representation for inference.

For inference, we desire two criteria: (a) support for many classes of inference, where the form of the inferred posterior depends on the algorithm; and (b) invariance of inference under the computational graph, that is, the posterior can be further composed as part of another model.

To explain our approach to this problem, we will use a simple hierarchical model $p(\mathbf{x}, \mathbf{z}, \beta)$, represented in Figure 3, as a running example. The ideas extend to more expressive programs.



```
N = 10000  # number of data points
D = 2  # data dimension
K = 5  # number of clusters

beta = Normal(mu=tf.zeros([K, D]), sigma=tf.ones([K, D]))
z = Categorical(logits=tf.zeros([N, K]))
x = Normal(mu=tf.gather(beta, z), sigma=tf.ones([N, D]))
```

Figure 3: Hierarchical model: (left) graphical model; (right) probabilistic program. It is a mixture of Gaussians over D-dimensional data $\{x_n\} \in \mathbb{R}^{N\times D}$. There are K latent cluster means $\beta \in \mathbb{R}^{K\times D}$.

3.1 Inference as Stochastic Graph Optimization

Given data $\mathbf{x}_{\text{train}}$, inference aims to calculate the posterior $p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are any model parameters that we will compute point estimates for.[2] We formalize this as the problem,

$$\min_{\boldsymbol{\lambda},\boldsymbol{\theta}} \mathcal{L}(p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}}; \boldsymbol{\theta}),\ q(\mathbf{z}, \beta; \boldsymbol{\lambda})),$$

where $q(\mathbf{z}, \beta; \boldsymbol{\lambda})$ is an approximation to the posterior $p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}})$, and $\mathcal{L}(\cdot)$ is some loss function with respect to p and q.

The choice of approximation q, loss $\mathcal{L}$, and rules to update parameters $\{\boldsymbol{\theta}, \boldsymbol{\lambda}\}$ are specified by an inference algorithm. (Note q can be nonparametric, such as a point or a collection of samples.)

In our language, we write this problem as follows:

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

where `qbeta` and `qz` are random variables defined to approximate the posterior. `Inference` is an abstract class which takes two inputs: a collection of latent variables, with model variables bound to posterior variables; and a collection of observed variables, with model variables bound to data. Calling

[2]For example, we could replace `x`'s `sigma` argument with `tf.exp(tf.Variable(0.0))*tf.ones([N, D])`. This defines a model parameter initialized at 0 and positive-constrained.

`inference.initialize()` builds a computational graph to update $\{\boldsymbol{\theta}, \boldsymbol{\lambda}\}$. Calling `inference.update()` runs this computation once to update $\{\boldsymbol{\theta}, \boldsymbol{\lambda}\}$; we call the method in a loop until convergence. Below we will derive subclasses of `Inference` to represent many methods.

3.2 Representing Classes of Inference

We show how to leverage the above to represent a broad class of inference methods.

In variational inference, the idea is to posit a family of approximating distributions and to find the closest member in the family to the posterior (Jordan et al., 1999). We build the variational family in the graph; see Figure 4 (left). The variational family has mutable variables representing its parameters $\boldsymbol{\lambda} = \{\pi, \mu, \sigma\}$, where $q(\beta; \mu, \sigma) = \text{Normal}(\beta; \mu, \sigma)$ and $q(\mathbf{z}; \pi) = \text{Categorical}(\mathbf{z}; \pi)$.

```
qbeta = Normal(
  mu=tf.Variable(tf.zeros([K, D])),
  sigma=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})
```

```
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})
```

Figure 4: (left) Variational inference. (right) Monte Carlo.

Specific variational inference algorithms inherit from the `VariationalInference` class to define their own methods, such as a loss function and gradient. For example, we represent maximum a posteriori (MAP) estimation with an approximating family (`qbeta` and `qz`) of `PointMass` random variables, i.e., with all probability mass concentrated at a point. `MAP` inherits from `VariationalInference` and defines a loss function and update rules; it leverages existing optimizers inside TensorFlow.

Monte Carlo approximates the posterior using samples (Robert and Casella, 1999). We represent Monte Carlo as inference where the approximating family is an empirical distribution, $q(\beta; \{\beta^{(t)}\}) = \frac{1}{T}\sum_{t=1}^{T} \delta(\beta, \beta^{(t)})$ and $q(\mathbf{z}; \{\mathbf{z}^{(t)}\}) = \frac{1}{T}\sum_{t=1}^{T} \delta(\mathbf{z}, \mathbf{z}^{(t)})$. The parameters are $\boldsymbol{\lambda} = \{\beta^{(t)}, \mathbf{z}^{(t)}\}$. See Figure 4 (right). Monte Carlo algorithms proceed by updating one sample $\beta^{(t)}, \mathbf{z}^{(t)}$ at a time in the empirical approximation. Specific MC samplers determine the update rules; they can leverage gradients and graph structure, where applicable.

The approach also extends to exact inference. We are developing a subpackage that does symbolic algebra on the deterministic and stochastic nodes in the computational graph; this uncovers conjugacy relationships between exponential-family random variables. This will allow users to integrate out variables and automatically derive classical Gibbs and mean-field updates (Bishop, 2006) without tedious algebraic manipulation.

3.3 Composing Inferences

Core to Edward's design is that inference can be written as a collection of separate inference programs. Below we demonstrate variational EM, with an (approximate) E-step over local variables and an M-step over global variables. We alternate with one update of each (Neal and Hinton, 1993).

```
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()
```

This extends to many other cases, such as exact EM for exponential families, contrastive divergence (Hinton, 2002), pseudo-marginal methods (Andrieu and Roberts, 2009), and Gibbs sampling within variational inference (Wang and Blei, 2012). We can also write message passing algorithms, which work over a collection of local inference problems (Koller and Friedman, 2009). For example, the local inference can be exact for classical message passing or minimize $\mathrm{KL}(p \,\|\, q)$ for expectation propagation (Minka, 2001). In Appendix B, we demonstrate this approach for stochastic variational inference.

Acknowledgements

We thank the Google BayesFlow team—Joshua Dillon, Ian Langmore, Ryan Sepassi, and Srinivas Vasudevan—as well as Amr Ahmed, Matthew Johnson, Hung Bui, Rajesh Ranganath, Maja Rudolph, and Francisco Ruiz for their helpful feedback and comments. This work is supported by NSF IIS-1247664, ONR N00014-11-1-0651, DARPA FA8750-14-2-0009, DARPA N66001-15-C-4032, Adobe, Google, NSERC PGS-D, Porter Ogden Jacobus Fellowship, Seibel Foundation, and the Sloan Foundation.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zhang, X. (2016). TensorFlow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695*.

Andrieu, C. and Roberts, G. O. (2009). The pseudo-marginal approach for efficient Monte Carlo computations. *The Annals of Statistics*, pages 697–725.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022.

Dieng, A. B., Tran, D., Ranganath, R., Paisley, J., and Blei, D. M. (2016). χ-divergence for approximate inference. In *arXiv preprint arXiv:1611.00328*.

Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Neural Information Processing Systems*.

Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. (2012). Church: A language for generative models. In *Uncertainty in Artificial Intelligence*.

Gregor, K., Danihelka, I., Graves, A., Rezende, D. J., and Wierstra, D. (2015). DRAW: A Recurrent Neural Network For Image Generation. In *International Conference on Machine Learning*.

Guadarrama, S. and Silberman, N. (2016). TensorFlow Slim.

Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800.

Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., and Saul, L. K. (1999). An introduction to variational methods for graphical models. *Machine Learning*, 37(2):183–233.

Kingma, D. P., Mohamed, S., Rezende, D. J., and Welling, M. (2014). Semi-supervised learning with deep generative models. In *Neural Information Processing Systems*.

Kingma, D. P., Salimans, T., and Welling, M. (2016). Improving Variational Inference with Inverse Autoregressive Flow. In *Neural Information Processing Systems*.

Kingma, D. P. and Welling, M. (2014). Auto-encoding variational Bayes. In *International Conference on Learning Representations*.

Koller, D. and Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2016). Building Machines That Learn and Think Like People. *arXiv preprint arXiv:1604.00289*.

Li, Y. and Turner, R. E. (2016). Variational inference with Rényi divergence. In *Neural Information Processing Systems*.

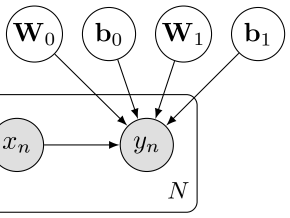
Mansinghka, V., Selsam, D., and Perov, Y. (2014). Venture: A higher-order probabilistic programming platform with programmable inference. *arXiv.org*.

Minka, T. P. (2001). Expectation propagation for approximate Bayesian inference. In *Uncertainty in Artificial Intelligence*.

Neal, R. M. and Hinton, G. E. (1993). A new view of the em algorithm that justifies incremental and other variants. In *Learning in Graphical Models*, pages 355–368. Kluwer Academic Publishers.

Ranganath, R., Altosaar, J., Tran, D., and Blei, D. M. (2016). Operator variational inference. In *Neural Information Processing Systems*.

Rezende, D. J. and Mohamed, S. (2015). Variational inference with normalizing flows. In *International Conference on Machine Learning*.



```
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)
```

Figure 5: Bayesian neural network for classification.

Rezende, D. J., Mohamed, S., Danihelka, I., Gregor, K., and Wierstra, D. (2016). One-shot generalization in deep generative models. In *International Conference on Machine Learning*.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*.

Robert, C. P. and Casella, G. (1999). *Monte Carlo Statistical Methods*. Springer.

Rudolph, M. R., Ruiz, F. J. R., Mandt, S., and Blei, D. M. (2016). Exponential family embeddings. In *Neural Information Processing Systems*.

Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., and Blei, D. M. (2016a). Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*.

Tran, D., Ranganath, R., and Blei, D. M. (2016b). The variational Gaussian process. In *International Conference on Learning Representations*.

Wang, C. and Blei, D. M. (2012). Truncation-free online variational inference for Bayesian nonparametric models. In *Neural Information Processing Systems*, pages 413–421.

Wu, Y., Li, L., Russell, S., and Bodik, R. (2016). Swift: Compiled inference for probabilistic programming languages. *arXiv preprint arXiv:1606.09242*.

A Model Examples

There are many examples available at `http://edwardlib.org`, including models, inference methods, and complete scripts. Below we describe several model examples; Appendix B describes an inference example (stochastic variational inference); Appendix C describes complete scripts.

A.1 Bayesian Neural Network for Classification

A Bayesian neural network is a neural network with a prior distribution on its weights.

Define the likelihood of an observation $(\mathbf{x}_n, y_n)$ with binary label $y_n \in \{0, 1\}$ as

$$p(y_n \mid \mathbf{W}_0, \mathbf{b}_0, \mathbf{W}_1, \mathbf{b}_1 \;;\; \mathbf{x}_n) = \text{Bernoulli}(y_n \mid \text{NN}(\mathbf{x}_n \;;\; \mathbf{W}_0, \mathbf{b}_0, \mathbf{W}_1, \mathbf{b}_1)),$$

where NN is a 2-layer neural network whose weights and biases form the latent variables $\mathbf{W}_0, \mathbf{b}_0, \mathbf{W}_1, \mathbf{b}_1$. Define the prior on the weights and biases to be the standard normal. See Figure 5. There are N data points, D features, and H hidden units.

A.2 Dirichlet Process Mixture Model

See Figure 6.

```
H = Normal(mu=tf.zeros(D), sigma=tf.ones(D))
mu = tf.pack([DirichletProcess(alpha=1.0, base=H) for _ in range(N)])
x = Normal(mu=mu, sigma=tf.ones(N))
```

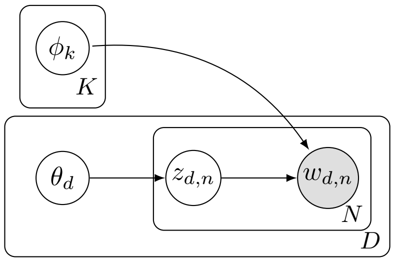
The essential component defining the `DirichletProcess` random variable is a stochastic while loop. We define it below.

```
def dirichlet_process(alpha):
  def cond(k, beta_k):
    flip = Bernoulli(p=beta_k)
    return tf.equal(flip, tf.constant(1))

  def body(k, beta_k):
    beta_k = beta_k * Beta(a=1.0, b=alpha)
    return k + 1, beta_k

  k = tf.constant(0)
  beta_k = Beta(a=1.0, b=alpha)
  stick_num, stick_beta = tf.while_loop(cond, body, loop_vars=[k, beta_k])
  return stick_num
```

Figure 6: Dirichlet process mixture model .

A.3 Latent Dirichlet Allocation

See Figure 7.

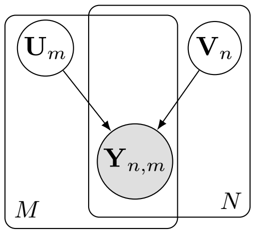


```
D = 50  # number of documents
N = [11502, 213, 1523, 1351, ...]  # words per doc
K = 10  # number of topics
V = 100000  # vocabulary size

theta = Dirichlet(alpha=tf.zeros([D, K]) + 0.1)
phi = Dirichlet(alpha=tf.zeros([K, V]) + 0.05)
z = [[0] * N] * D
w = [[0] * N] * D
for d in range(D):
  for n in range(N[d]):
    z[d][n] = Categorical(pi=theta[d, :])
    w[d][n] = Categorical(pi=phi[z[d][n], :])
```

Figure 7: Latent Dirichlet allocation (Blei et al., 2003).

A.4 Gaussian Matrix Factorizationn

See Figure 8.



```
N = 10
M = 10
K = 5  # latent dimension

U = Normal(mu=tf.zeros([M, K]), sigma=tf.ones([M, K]))
V = Normal(mu=tf.zeros([N, K]), sigma=tf.ones([N, K]))
Y = Normal(mu=tf.matmul(U, V, transpose_b=True), sigma=tf.ones([N, M]))
```

Figure 8: Gaussian matrix factorization.

B Inference Example: Stochastic Variational Inference

In the subgraph setting, we do data subsampling while working with a subgraph of the full model. This setting is necessary when the data and model do not fit in memory. It is scalable in that both the algorithm's computational complexity (per iteration) and memory complexity are independent of the data set size.

For the code, we use the running example, a mixture model described in Figure 3.

```
N = 10000000  # data set size
D = 2  # data dimension
K = 5  # number of clusters
```

The model is

$$p(\mathbf{x}, \mathbf{z}, \beta) = p(\beta) \prod_{n=1}^{N} p(z_n \mid \beta) p(x_n \mid z_n, \beta).$$

To avoid memory issues, we work on only a subgraph of the model,

$$p(\mathbf{x}, \mathbf{z}, \beta) = p(\beta) \prod_{m=1}^{M} p(z_m \mid \beta) p(x_m \mid z_m, \beta)$$

```
M = 128  # mini-batch size

beta = Normal(mu=tf.zeros([K, D]), sigma=tf.ones([K, D]))
z = Categorical(logits=tf.zeros([M, K]))
x = Normal(mu=tf.gather(beta, z), sigma=tf.ones([M, D]))
```

Assume the variational model is

$$q(\mathbf{z}, \beta) = q(\beta; \lambda) \prod_{n=1}^{N} q(z_n \mid \beta; \gamma_n),$$

parameterized by $\{\lambda, \{\gamma_n\}\}$. Again, we work on only a subgraph of the model,

$$q(\mathbf{z}, \beta) = q(\beta; \lambda) \prod_{m=1}^{M} q(z_m \mid \beta; \gamma_m).$$

parameterized by $\{\lambda, \{\gamma_m\}\}$. Importantly, only M parameters are stored in memory for $\{\gamma_m\}$ rather than N.

```
qbeta = Normal(mu=tf.Variable(tf.zeros([K, D])),
               sigma=tf.nn.softplus(tf.Variable(tf.zeros[K, D])))
qz_variables = tf.Variable(tf.zeros([M, K]))
qz = Categorical(logits=qz_variables)
```

We instantiate the inference algorithm to perform inference over β and the subset of $\mathbf{z}$. We use `KLqp`, a variational method that minimizes the divergence measure $\text{KL}(q \,\|\, p)$ (Jordan et al., 1999). We also pass in a TensorFlow placeholder `x_ph` for the data, so we can change the data at each step. (Alternatively, batch tensors in TensorFlow can be used.)

```
x_ph = tf.placeholder(tf.float32, [M])
inference = ed.KLqp({beta: qbeta, z: qz}, data={x: x_ph})
```

We initialize the algorithm with the `scale` argument, so that computation on `z` and `x` will be scaled appropriately. This enables unbiased estimates for stochastic gradients.

```
inference.initialize(scale={x: float(N) / M, z: float(N) / M})
```

We now run the algorithm, assuming there is a `next_batch` function which provides the next batch of data.

```
qz_init = tf.initialize_variables([qz_variables])
for _ in range(10000):
  x_batch = next_batch(size=M)
  for _ in range(10): # run multiple updates for each batch
    inference.update(feed_dict={x_ph: x_batch})
  # reinitialize the local factors
  qz_init.run()
```

After each iteration, we reinitialize the parameters for $q(\mathbf{z} \mid \beta; \boldsymbol{\gamma})$; this is because we do inference on a new set of local variational factors for each batch. This demo readily applies to other stochastic inference algorithms such as stochastic gradient Langevin dynamics: simply replace `qbeta` and `qz` with `Empirical` random variables; then call `ed.SGLD` instead of `ed.KLqp`.

Note that if the data and model fit in memory but you'd still like to perform data subsampling for fast inference, we recommend not defining subgraphs. You can reify the full model, and simply index the local variables with a placeholder. The placeholder is fed at runtime to determine which of the local variables to update at a time. (For more details, see the website's API.)

C Complete Examples

C.1 Variational Auto-encoder

See Figure 9.

```
import edward as ed
import tensorflow as tf

from edward.models import Bernoulli, Normal
from scipy.misc import imsave
from tensorflow.contrib import slim
from tensorflow.examples.tutorials.mnist import input_data

M = 100  # batch size during training
d = 2  # latent variable dimension

# Probability model (subgraph)
z = Normal(mu=tf.zeros([M, d]), sigma=tf.ones([M, d]))
h = slim.fully_connected(z, 256)
x = Bernoulli(logits=slim.fully_connected(h, 28 * 28, activation_fn=None))

# Variational model (subgraph)
x_ph = tf.placeholder(tf.float32, [M, 28 * 28])
qh = slim.fully_connected(x_ph, 256)
qz = Normal(mu=slim.fully_connected(qh, d, activation_fn=None),
            sigma=slim.fully_connected(qh, d, activation_fn=tf.nn.softplus))

# Bind p(x, z) and q(z | x) to the same TensorFlow placeholder for x.
mnist = input_data.read_data_sets("data/mnist", one_hot=True)
data = {x: x_ph}

inference = ed.KLqp({z: qz}, data)
optimizer = tf.train.RMSPropOptimizer(0.01, epsilon=1.0)
inference.initialize(optimizer=optimizer)

tf.initialize_all_variables().run()

n_epoch = 100
n_iter_per_epoch = 1000
for _ in range(n_epoch):
  for _ in range(n_iter_per_epoch):
    x_train, _ = mnist.train.next_batch(M)
    info_dict = inference.update(feed_dict={x_ph: x_train})

  # Generate images.
  imgs = x.value().eval()
  for m in range(M):
    imsave("img/%d.png" % m, imgs[m].reshape(28, 28))
```

Figure 9: Complete script for a VAE (Kingma and Welling, 2014) with batch training. It generates MNIST digits after every 1000 updates.

C.2 Generative Model for Word Embeddings

See Figure 10. This example uses data subsampling (Appendix B). The priors and conditional likelihoods are defined only for a minibatch of data. Similarly the variational model only models the embeddings used in a given minibatch. TensorFlow variables contain the embedding vectors for the entire vocabulary. TensorFlow placeholders ensure that the correct embedding vectors are used as variational parameters for a given minibatch.

The Bernoulli variables `y_pos` and `y_neg` are fixed to be 1's and 0's respectively. They model whether a word is indeed the target word for a given context window or has been drawn as a negative sam-

```
import edward as ed
import tensorflow as tf

from edward.models import Bernoulli, Normal, PointMass

N = 581238  # number of total words
M = 128  # batch size during training
K = 100  # number of factors
ns = 3  # number of negative samples
cs = 4  # context size
L = 50000  # vocabulary size

# Prior over embedding vectors
p_rho = Normal(mu=tf.zeros([M, K]),
               sigma=tf.sqrt(N) * tf.ones([M, K]))
n_rho = Normal(mu=tf.zeros([M, ns, K]),
               sigma=tf.sqrt(N) * tf.ones([M, ns, K]))

# Prior over context vectors
ctx_alphas = Normal(mu=tf.zeros([M, cs, K]),
                    sigma=tf.sqrt(N)*tf.ones([M, cs, K]))

# Conditional likelihoods
ctx_sum = tf.reduce_sum(ctx_alphas, [1])
p_eta = tf.expand_dims(tf.reduce_sum(tf.mul(p_rho, ctx_sum), -1),1)
n_eta = tf.reduce_sum(n_rho * tf.tile(tf.expand_dims(ctx_sum, 1), [1, ns, 1]), -1)
y_pos = Bernoulli(logits=p_eta)
y_neg = Bernoulli(logits=n_eta)

# placeholders for batch training
p_idx = tf.placeholder(tf.int32, [M, 1])
n_idx = tf.placeholder(tf.int32, [M, ns])
ctx_idx = tf.placeholder(tf.int32, [M, cs])

# Variational parameters (embedding vectors)
rho_params = tf.Variable(tf.random_normal([L, K]))
alpha_params = tf.Variable(tf.random_normal([L, K]))

# Variational distribution on embedding vectors
q_p_rho = PointMass(params=tf.squeeze(tf.gather(rho_params, p_idx)))
q_n_rho = PointMass(params=tf.gather(rho_params, n_idx))
q_alpha = PointMass(params=tf.gather(alpha_params, ctx_idx))

inference = ed.MAP(
  {p_rho: q_p_rho, n_rho: q_n_rho, ctx_alphas: q_alpha},
  data={y_pos: tf.ones((M, 1)), y_neg: tf.zeros((M, ns))})

inference.initialize()
tf.initialize_all_variables().run()

for _ in range(inference.n_iter):
  targets, windows, negatives = next_batch(M)  # a function to generate data
  info_dict = inference.update(feed_dict={p_idx: targets, ctx_idx: windows, n_idx: negatives})
  inference.print_progress(info_dict)
```

Figure 10: Exponential family embedding for binary data (Rudolph et al., 2016). Here, MAP is used to maximize the total sum of conditional log-likelihoods and log-priors.

ple. Without regularization (via priors), the objective we optimize is identical to negative sampling.