Edward: A library for probabilistic modeling, inference, and criticism

Dustin Tran, Alp Kucukelbir, Adji B. Dieng,
Maja Rudolph, Dawen Liang, and David M. Blei

Columbia University

February 10, 2017

Abstract

Probabilistic modeling is a powerful approach for analyzing empirical information. We describe *Edward*, a library for probabilistic modeling. Edward's design reflects an iterative process pioneered by George Box: build a model of a phenomenon, make inferences about the model given data, and criticize the model's fit to the data. Edward supports a broad class of probabilistic models, efficient algorithms for inference, and many techniques for model criticism. The library builds on top of TensorFlow to support distributed training and hardware such as GPUs. Edward enables the development of complex probabilistic models and their algorithms at a massive scale.

Keywords: Probabilistic Models; Bayesian Inference; Model Criticism; Neural Networks; Scalable Computation; Probabilistic Programming.[1]

Contents

[1] Details in this paper describe Edward version 1.2.1, released Jan 30, 2017.

1 Introduction

Probabilistic modeling is a powerful approach for analyzing empirical information (Tukey, 1962; Newell and Simon, 1976; Box, 1976). Probabilistic models are essential to fields related to its methodology, such as statistics (Friedman et al., 2001; Gelman et al., 2013) and machine learning (Murphy, 2012; Goodfellow et al., 2016), as well as fields related to its application, such as computational biology (Friedman et al., 2000), computational neuroscience (Dayan and Abbott, 2001), cognitive science (Tenenbaum et al., 2011), information theory (MacKay, 2003), and natural language processing (Manning and Schütze, 1999).

Software systems for probabilistic modeling provide new and faster ways of experimentation. This enables research advances in probabilistic modeling that could not have been completed before.

As an example of such software systems, we point to early work in artificial intelligence. Expert systems were designed from human expertise, which in turn enabled larger reasoning steps according to existing knowledge (Buchanan et al., 1969; Minsky, 1975). With connectionist models, the design focused on neuron-like processing units, which learn from experience; this drove new applications of artificial intelligence (Hopfield, 1982; Rumelhart et al., 1988).

As another example, we point to early work in statistical computing, where interest grew broadly out of efficient computation for problems in statistical analysis. The S language, developed by John Chambers and colleagues at Bell Laboratories (Becker and Chambers, 1984; Chambers and Hastie, 1992), focused on an interactive environment for data analysis, with simple yet rich syntax to quickly turn ideas into software. It is a predecessor to the R language (Ihaka and Gentleman, 1996). More targeted environments such as BUGS (Spiegelhalter et al., 1995), which focuses on Bayesian analysis of statistical models, helped launch the emerging field of probabilistic programming.

We are motivated to build on these early works in probabilistic systems—where in modern applications, new challenges arise in their design and implementation. We highlight two challenges. First, statistics and machine learning have made significant advances in the methodology of probabilistic models and their inference (e.g., Hoffman et al. (2013); Ranganath et al. (2014); Rezende et al. (2014)). For software systems to enable fast experimentation, we require rich abstractions that can capture these advances: it must encompass both a broad class of probabilistic models and a broad class of algorithms for their efficient inference. Second, researchers are increasingly motivated to employ complex probabilistic models and at an unprecedented scale of massive data (Bengio et al., 2013; Ghahramani, 2015; Lake et al., 2016). Thus we require an efficient computing environment that supports distributed training and integration of hardware such as (multiple) GPUs.

We present *Edward*, a probabilistic modeling library named after the statistician George Edward Pelham Box. Edward is built around an iterative process for probabilistic modeling, pioneered by Box and his collaborators (Box and Hunter, 1962, 1965; Box and Hill, 1967; Box, 1976, 1980). The process is as follows: given data from some unknown phenomena, first, formulate a model of the phenomena; second, use an algorithm to infer the model's hidden structure, thus reasoning about the phenomena; third, criticize how well the model captures the data's generative process. As we criticize our model's fit to the data, we revise components of the model and repeat to form an iterative loop (Box, 1976; Blei, 2014; Gelman et al., 2013).

Edward builds infrastructure to enable this loop:

1. For *modeling*, Edward provides a language of random variables to construct a broad class of models: directed graphical models (Pearl, 1988), stochastic neural networks (Neal, 1990), and programs with stochastic control flow (Goodman et al., 2012).
2. For *inference*, Edward provides algorithms such as stochastic and black box variational inference (Hoffman et al., 2013; Ranganath et al., 2014), Hamiltonian Monte Carlo (Neal, 1993), and stochastic gradient Langevin dynamics (Welling and Teh, 2011). Edward also provides infrastructure to make it easy to develop new algorithms.

3. For *criticism*, Edward provides methods from scoring rules (Winkler, 1996) and predictive checks (Box, 1980; Rubin, 1984).

Edward is built on top of TensorFlow, a library for numerical computing using data flow graphs (Abadi et al., 2016). TensorFlow enables Edward to speed up computation with hardware such as GPUs, to scale up computation with distributed training, and to simplify engineering effort with automatic differentiation.

In Section 2, we demonstrate Edward with an example. In Section 3, we describe the design of Edward. In Section 4, we provide examples of how standard tasks in statistics and machine learning can be solved with Edward.

Related work

Probabilistic programming. There has been much work on programming languages which specify broad classes of probabilistic models, or probabilistic programs. Recent works include Church (Goodman et al., 2012), Venture (Mansinghka et al., 2014), Anglican (Wood et al., 2015), Stan (Carpenter et al., 2016), and WebPPL (Goodman and Stuhlmüller, 2014). The most important distinction in Edward stems from motivation. We are interested in deploying probabilistic models to many real world applications, ranging from the size of data and data structure, such as large text corpora or many brief audio signals, to the size of model and class of models, such as small nonparametric models or deep generative models. Thus Edward is built with fast computation in mind.

Black box inference. Black box algorithms are typically based on Monte Carlo methods, and make very few assumptions about the model (Metropolis and Ulam, 1949; Hastings, 1970; Geman and Geman, 1984). Our motivation as outlined above presents a new set of challenges in both inference research and software design. As a first consequence, we focus on variational inference (Hinton and van Camp, 1993; Waterhouse et al., 1996; Jordan et al., 1999). As a second consequence, we encourage active research on inference by providing a class hierarchy of inference algorithms. As a third consequence, our inference algorithms aim to take advantage of as much structure as possible from the model. Edward supports all types of inference, whether they be black box or model-specific (Dempster et al., 1977; Hoffman et al., 2013).

Computational frameworks. There are many computational frameworks, primarily built for deep learning: as of this date, this includes TensorFlow (Abadi et al., 2016), Theano (Al-Rfou et al., 2016), Torch (Collobert and Kavukcuoglu, 2011), neon (Nervana Systems, 2014), and the Stan Math Library (Carpenter et al., 2015). These are incredible tools which Edward employs as a backend. In terms of abstraction, Edward sits one level higher.

High-level deep learning libraries. Neural network libraries such as Keras (Chollet, 2015) and Lasagne (Dieleman et al., 2015) are at a similar abstraction level as Edward. However both are primarily interested in parameterizing complicated functions for supervised learning on large datasets. We are interested in probabilistic models which apply to a wide array of learning tasks. These tasks may have both complicated likelihood and complicated priors (neural networks are an option but not a necessity). Therefore our goals are orthogonal and in fact mutually beneficial to each other. For example, we use Keras' abstraction as a way to easily specify models parameterized by deep neural networks.

2 Getting Started

Probabilistic modeling in Edward uses a simple language of random variables. Here we will show a Bayesian neural network. It is a neural network with a prior distribution on its weights.

First, simulate a toy dataset of 50 observations with a cosine relationship.

```
import numpy as np

x_train = np.linspace(-3, 3, num=50)
y_train = np.cos(x_train) + np.random.normal(0, 0.1, size=50)
x_train = x_train.astype(np.float32).reshape((50, 1))
y_train = y_train.astype(np.float32).reshape((50, 1))
```

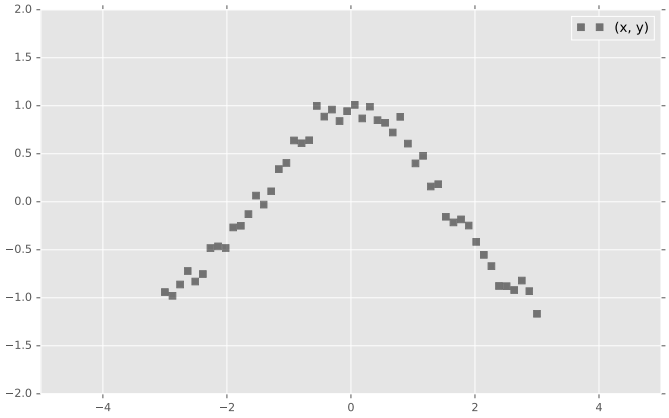

Figure 1: Simulated data with a cosine relationship between x and y.

Next, define a two-layer Bayesian neural network. Here, we define the neural network manually with `tanh` nonlinearities.

```
import tensorflow as tf
from edward.models import Normal

W_0 = Normal(mu=tf.zeros([1, 2]), sigma=tf.ones([1, 2]))
W_1 = Normal(mu=tf.zeros([2, 1]), sigma=tf.ones([2, 1]))
b_0 = Normal(mu=tf.zeros(2), sigma=tf.ones(2))
b_1 = Normal(mu=tf.zeros(1), sigma=tf.ones(1))

x = x_train
y = Normal(mu=tf.matmul(tf.tanh(tf.matmul(x, W_0) + b_0), W_1) + b_1,
           sigma=0.1)
```

Next, make inferences about the model from data. We will use variational inference. Specify a normal approximation over the weights and biases.

```
qW_0 = Normal(mu=tf.Variable(tf.zeros([1, 2])),
              sigma=tf.nn.softplus(tf.Variable(tf.zeros([1, 2]))))
qW_1 = Normal(mu=tf.Variable(tf.zeros([2, 1])),
              sigma=tf.nn.softplus(tf.Variable(tf.zeros([2, 1]))))
qb_0 = Normal(mu=tf.Variable(tf.zeros(2)),
              sigma=tf.nn.softplus(tf.Variable(tf.zeros(2))))
qb_1 = Normal(mu=tf.Variable(tf.zeros(1)),
              sigma=tf.nn.softplus(tf.Variable(tf.zeros(1))))
```

Defining `tf.Variable` allows the variational factors' parameters to vary. They are all initialized at 0. The standard deviation parameters are constrained to be greater than zero according to a softplus transformation[2].

Now, run variational inference with the Kullback-Leibler divergence in order to infer the model's latent variables given data. We specify `1000` iterations.

[2] The softplus function is defined as $\text{softplus}(x) = \log(1 + \exp(x))$.

```
import edward as ed

inference = ed.KLqp({W_0: qW_0, b_0: qb_0,
                     W_1: qW_1, b_1: qb_1}, data={y: y_train})
inference.run(n_iter=1000)
```

Finally, criticize the model fit. Bayesian neural networks define a distribution over neural networks, so we can perform a graphical check. Draw neural networks from the inferred model and visualize how well it fits the data.

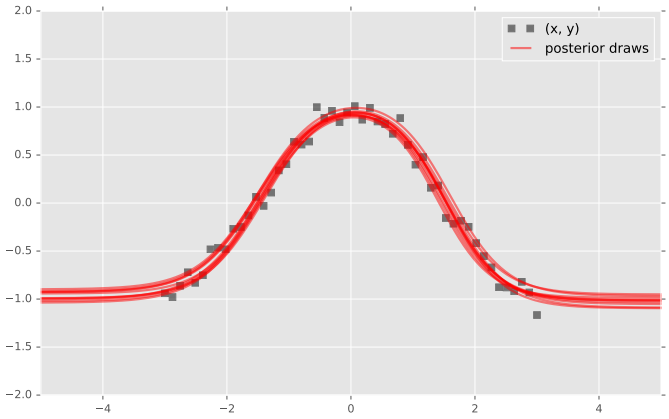

Figure 2: Posterior draws from the inferred Bayesian neural network.

The model has captured the cosine relationship between x and y in the observed domain.

3 Design

Edward's design reflects the building blocks for probabilistic modeling. It defines interchangeable components, enabling rapid experimentation and research with probabilistic models.

Edward is named after the innovative statistician George Edward Pelham Box. Edward follows Box's philosophy of statistics and machine learning (Box, 1976).

First gather data from some real-world phenomena. Then cycle through Box's loop (Blei, 2014).

1. Build a probabilistic model of the phenomena.
2. Reason about the phenomena given model and data.
3. Criticize the model, revise and repeat.

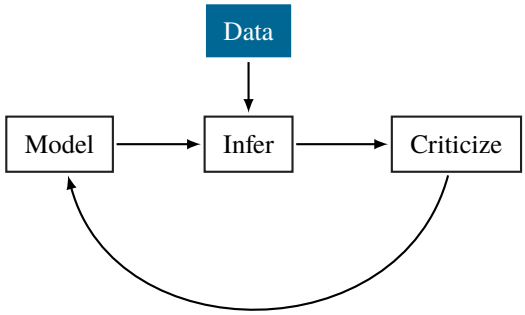

Figure 3: Box's loop.

Here's a toy example. A child flips a coin ten times, with the set of outcomes being

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
```

where `0` denotes tails and `1` denotes heads. She is interested in the probability that the coin lands heads. To analyze this, she first builds a model: suppose she assumes the coin flips are independent and land heads with the same probability. Second, she reasons about the phenomenon: she infers the model's hidden structure given data. Finally, she criticizes the model: she analyzes whether her model captures the real-world phenomenon of coin flips. If it doesn't, then she may revise the model and repeat.

We describe modules enabling this analysis.

3.1 Data

Data defines a set of observations. There are three ways to read data in Edward.

Preloaded data. A constant or variable in the TensorFlow graph holds all the data. This setting is the fastest to work with and is recommended if the data fits in memory.

Represent the data as NumPy arrays or TensorFlow tensors.

```
x_data = np.array([0, 1, 0, 0, 0, 0, 0, 0, 0, 1])
x_data = tf.constant([0, 1, 0, 0, 0, 0, 0, 0, 0, 1])
```

During inference, we store them in TensorFlow variables internally to prevent copying data more than once in memory.

Feeding. Manual code provides the data when running each step of inference. This setting provides the most fine control which is useful for experimentation.

Represent the data as TensorFlow placeholders, which are nodes in the graph that are fed at runtime.

```
x_data = tf.placeholder(tf.float32, [100, 25])  # placeholder of shape (100, 25)
```

During inference, the user must manually feed the placeholders. At each step, call `inference.update()` while passing in a `feed_dict` dictionary which binds placeholders to realized values as an argument. If the values do not change over inference updates, one can also bind the placeholder to values within the `data` argument when first constructing inference.

Reading from files. An input pipeline reads the data from files at the beginning of a TensorFlow graph. This setting is recommended if the data does not fit in memory.

```
filename_queue = tf.train.string_input_producer(...)
reader = tf.SomeReader()
...
```

Represent the data as TensorFlow tensors, where the tensors are the output of data readers. During inference, each update will be automatically evaluated over new batch tensors represented through the data readers.

3.2 Models

A probabilistic model is a joint distribution $p(\mathbf{x}, \mathbf{z})$ of data $\mathbf{x}$ and latent variables $\mathbf{z}$.

In Edward, we specify models using a simple language of random variables. A random variable $\mathbf{x}$ is an object parameterized by tensors θ^*, where the number of random variables in one object is determined by the dimensions of its parameters.

```
from edward.models import Normal, Exponential

# univariate normal
Normal(mu=tf.constant(0.0), sigma=tf.constant(1.0))
# vector of 5 univariate normals
Normal(mu=tf.zeros(5), sigma=tf.ones(5))
# 2 x 3 matrix of Exponentials
Exponential(lam=tf.ones([2, 3]))
```

For multivariate distributions, the multivariate dimension is the innermost (right-most) dimension of the parameters.

```
from edward.models import Dirichlet, MultivariateNormalFull

# K-dimensional Dirichlet
Dirichlet(alpha=tf.constant([0.1] * K)
# vector of 5 K-dimensional multivariate normals
MultivariateNormalFull(mu=tf.zeros([5, K]), sigma=...)
# 2 x 5 matrix of K-dimensional multivariate normals
MultivariateNormalFull(mu=tf.zeros([2, 5, K]), sigma=...)
```

Random variables are equipped with methods such as `log_prob()`, $\log p(\mathbf{x} \mid \theta^*)$, `mean()`, $\mathbb{E}_{p(\mathbf{x}|\theta^*)}[\mathbf{x}]$, and `sample()`, $\mathbf{x}^* \sim p(\mathbf{x} \mid \theta^*)$. Further, each random variable is associated to a tensor $\mathbf{x}^*$ in the computational graph, which represents a single sample $\mathbf{x}^* \sim p(\mathbf{x} \mid \theta^*)$.

This makes it easy to parameterize random variables with complex deterministic structure, such as with deep neural networks, a diverse set of math operations, and compatibility with third party libraries which also build on TensorFlow. The design also enables compositions of random variables to capture complex stochastic structure. They operate on $\mathbf{x}^*$.

```
from edward.models import Normal

x = Normal(mu=tf.zeros(10), sigma=tf.ones(10))
y = tf.constant(5.0)
x + y, x - y, x * y, x / y
tf.tanh(x * y)
tf.gather(x, 2)  # 3rd normal rv in the vector
```

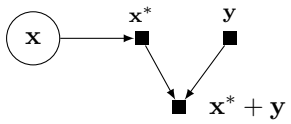

Figure 4: Random variables can be combined with other TensorFlow ops.

Below we describe how to build models by composing random variables.

For a list of random variables supported in Edward, see the TensorFlow distributions API. [3] Edward random variables build on top of them, inheriting the same arguments and class methods. Additional methods are also available, detailed in Edward's API.

Composing Random Variables

Core to Edward's design is compositionality. Compositionality enables fine control of modeling, where models are represented as a collection of random variables.

We outline how to write popular classes of models using Edward: directed graphical models, neural networks, Bayesian nonparametrics, and probabilistic programs.

Directed Graphical Models

Graphical models are a rich formalism for specifying probability distributions (Koller and Friedman, 2009). In Edward, directed edges in a graphical model are implicitly defined when random variables are composed with one another. We illustrate with 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 latent probability shared across the 50 data points $\mathbf{x} \in \{0, 1\}^{50}$.

```
from edward.models import Bernoulli, Beta

theta = Beta(a=1.0, b=1.0)
x = Bernoulli(p=tf.ones(50) * theta)
```

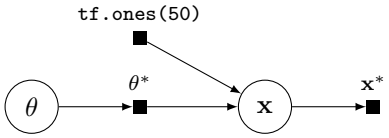

Figure 5: Computational graph for a Beta-Bernoulli program.

The random variable `x` ($\mathbf{x}$) is 50-dimensional, parameterized by the random tensor θ^*. Fetching the object `x.value()` ($\mathbf{x}^*$) from session runs the graph: it simulates from the generative process and outputs a binary vector of 50 elements.

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, that is, parameters that we will always compute point estimates for and not be uncertain about. In TensorFlow, this is given by a `tf.Variable`.

[3] https://www.tensorflow.org/versions/master/api_docs/python/contrib.distributions.html

```
from edward.models import Bernoulli

theta = tf.Variable(0.0)
x = Bernoulli(p=tf.ones(50) * tf.sigmoid(theta))
```

Another use case of mutable states 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.

Neural Networks

As Edward uses TensorFlow, it is easy to construct neural networks for probabilistic modeling (Rumelhart et al., 1988). For example, one can specify stochastic neural networks (Neal, 1990).

High-level libraries such as Keras[4] and TensorFlow Slim[5] can be used to easily construct deep neural networks. We illustrate this with a deep generative model over binary data $\{\mathbf{x}_n\} \in \{0,1\}^{N \times 28 * 28}$.

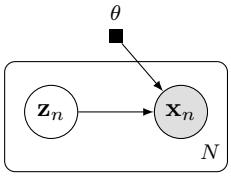

Figure 6: Graphical representation of a deep generative model.

The model specifies a generative process where for each $n = 1, \ldots, N$,

$$\begin{aligned} \mathbf{z}_n &\sim \text{Normal}(\mathbf{z}_n \mid \mathbf{0}, \mathbf{I}), \\ \mathbf{x}_n \mid \mathbf{z}_n &\sim \text{Bernoulli}(\mathbf{x}_n \mid p = \text{NN}(\mathbf{z}_n; \theta)). \end{aligned}$$

The latent space is $\mathbf{z}_n \in \mathbb{R}^d$ and the likelihood is parameterized by a neural network NN with parameters θ. We will use a two-layer neural network with a fully connected hidden layer of 256 units (with ReLU activation) and whose output is $28 * 28$-dimensional. The output will be unconstrained, parameterizing the logits of the Bernoulli likelihood.

With TensorFlow Slim, we write this model as follows:

```
from edward.models import Bernoulli, Normal
from tensorflow.contrib import slim

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))
```

With Keras, we write this model as follows:

```
from edward.models import Bernoulli, Normal
from keras.layers import Dense

z = Normal(mu=tf.zeros([N, d]), sigma=tf.ones([N, d]))
h = Dense(256, activation='relu')(z.value())
x = Bernoulli(logits=Dense(28 * 28)(h))
```

Keras and TensorFlow Slim automatically manage TensorFlow variables, which serve as parameters of the high-level neural network layers. This saves the trouble of having to manage them manually. However, note that neural network parameters defined this way always serve as model parameters. That is, the parameters are not exposed to the user so we cannot be Bayesian about them with prior distributions.

[4] http://keras.io

[5] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim

Bayesian Nonparametrics

Bayesian nonparametrics enable rich probability models by working over an infinite-dimensional parameter space (Hjort et al., 2010). Edward supports the two typical approaches to handling these models: collapsing the infinite-dimensional space and lazily defining the infinite-dimensional space.

For the collapsed approach, see the Gaussian process classification tutorial as an example. We specify distributions over the function evaluations of the Gaussian process, and the Gaussian process is implicitly marginalized out. This approach is also useful for Poisson process models.

To work directly on the infinite-dimensional space, one can leverage random variables with control flow operations in TensorFlow. At runtime, the control flow will lazily define any parameters in the space necessary in order to generate samples. As an example, we use a while loop to define a Dirichlet process according to its stick breaking representation.

Probabilistic Programs

Probabilistic programs greatly expand the scope of probabilistic models (Goodman et al., 2012). Formally, Edward is a Turing-complete probabilistic programming language. This means that Edward can represent any computable probability distribution.

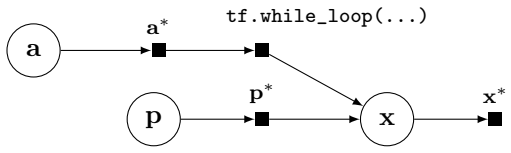

Figure 7: Computational graph for a probabilistic program with stochastic control flow.

Random variables can be composed with control flow operations, 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 above, where $\mathbf{x}$ may or may not depend on $\mathbf{a}$ for a given execution.

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.

Developing Custom Random Variables

Oftentimes we'd like to implement our own random variables. To do so, write a class that inherits the `RandomVariable` class in `edward.models` and the `Distribution` class in `tf.contrib.distributions` (in that order). A template is provided below.

```
from edward.models import RandomVariable
from tensorflow.contrib.distributions import Distribution

class CustomRandomVariable(RandomVariable, Distribution):
  def __init__(self, *args, **kwargs):
    super(CustomRandomVariable, self).__init__(*args, **kwargs)

  def _log_prob(self, value):
    raise NotImplementedError("log_prob is not implemented")

  def _sample_n(self, n, seed=None):
    raise NotImplementedError("sample_n is not implemented")
```

One method that all Edward random variables call during instantiation is `_sample_n()`. It takes an integer `n` as input and outputs a tensor of shape `(n,) + batch_shape + event_shape`. To implement it, you can for example wrap a NumPy/SciPy function inside the TensorFlow operation `tf.py_func()`.

For other methods and attributes one can implement, see the API documentation in TensorFlow's `Distribution` class.

Advanced settings

Sometimes the random variable you'd like to work with already exists in Edward, but it is missing a particular feature. One hack is to implement and overwrite the missing method. For example, to implement your own sampling for `Poisson`:

```
import edward as ed
from edward.models import Poisson
from scipy.stats import poisson

def _sample_n(self, n=1, seed=None):
  # define Python function which returns samples as a Numpy array
  def np_sample(lam, n):
    return poisson.rvs(mu=lam, size=n, random_state=seed).astype(np.float32)

  # wrap python function as tensorflow op
  val = tf.py_func(np_sample, [self.lam, n], [tf.float32])[0]
  # set shape from unknown shape
  batch_event_shape = self.get_batch_shape().concatenate(self.get_event_shape())
  shape = tf.concat([tf.expand_dims(n, 0),
                     tf.constant(batch_event_shape.as_list(), dtype=tf.int32)],
                     0)
  val = tf.reshape(val, shape)
  return val

Poisson._sample_n = _sample_n

sess = ed.get_session()
x = Poisson(lam=1.0)
sess.run(x.value())
## 1.0
sess.run(x.value())
## 4.0
```

(Note the function `np_sample` should broadcast correctly if you'd like to work with non-scalar parameters; it is not correct in this toy implementation.)

Sometimes the random variable you'd like to work with does not even admit (easy) sampling, and you're only using it as a likelihood "node" rather than as some prior to parameters of another random variable. You can avoid having to implement `_sample_n` altogether: after creating `CustomRandomVariable`, instantiate it with the `value` argument:

```
x = CustomRandomVariable(custom_params=params, value=tf.zeros_like(params))
```

This fixes the associated value of the random variable to a bunch of zeros and avoids the `_sample_n` error that appears otherwise. Make sure that the value matches the desired shape of the random variable.

3.3 Inference

We describe how to perform inference in probabilistic models.

Suppose we have a model $p(\mathbf{x}, \mathbf{z}, \beta)$ of data $\mathbf{x}_{\text{train}}$ with latent variables $(\mathbf{z}, \beta)$. Consider the posterior inference problem,

$$q(\mathbf{z}, \beta) \approx p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}}),$$

in which the task is to approximate the posterior $p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}})$ using a family of distributions, $q(\mathbf{z}, \beta; \lambda)$, indexed by parameters λ.

In Edward, let `z` and `beta` be latent variables in the model, where we observe the random variable `x` with data `x_train`. Let `qz` and `qbeta` be random variables defined to approximate the posterior. We write this problem as follows:

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

`Inference` is an abstract class which takes two inputs. The first is a collection of latent random variables `beta` and `z`, along with "posterior variables" `qbeta` and `qz`, which are associated to their respective latent variables. The second is a collection of observed random variables `x`, which is associated to the data `x_train`.

Inference adjusts parameters of the distribution of `qbeta` and `qz` to be close to the posterior $p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}})$.

Running inference is as simple as running one method.

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

Inference also supports fine control of the training procedure.

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

tf.global_variables_initializer().run()

for _ in range(inference.n_iter):
  info_dict = inference.update()
  inference.print_progress(info_dict)

inference.finalize()
```

`initialize()` builds the algorithm's update rules (computational graph) for λ;
`tf.global_variables_initializer().run()` initializes λ (TensorFlow variables in the graph); `update()` runs the graph once to update λ, which is called in a loop until convergence; `finalize()` runs any computation as the algorithm terminates.

The `run()` method is a simple wrapper for this procedure.

Other Settings

We highlight other settings during inference.

Model parameters. Model parameters are parameters in a model that we will always compute point estimates for and not be uncertain about. They are defined with `tf.Variable`s, where the inference problem is

$$\hat{\theta} \leftarrow^{\text{optimize}} p(\mathbf{x}_{\text{train}}; \theta)$$

```
from edward.models import Normal

theta = tf.Variable(0.0)
x = Normal(mu=tf.ones(10) * theta, sigma=1.0)

inference = ed.Inference({}, {x: x_train})
```

Only a subset of inference algorithms support estimation of model parameters. (Note also that this inference example does not have any latent variables. It is only about estimating `theta` given that we observe $\mathbf{x} = \mathbf{x}_{\text{train}}$. We can add them so that inference is both posterior inference and parameter estimation.)

For example, model parameters are useful when applying neural networks from high-level libraries such as Keras and TensorFlow Slim. See the model compositionality subsection for more details.

Conditional inference. In conditional inference, only a subset of the posterior is inferred while the rest are fixed using other inferences. The inference problem is

$$q(\beta)q(\mathbf{z}) \approx p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}})$$

where parameters in $q(\beta)$ are estimated and $q(\mathbf{z})$ is fixed. In Edward, we enable conditioning by binding random variables to other random variables in `data`.

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

In the inference compositionality subsection, we describe how to construct inference by composing many conditional inference algorithms.

Implicit prior samples. Latent variables can be defined in the model without any posterior inference over them. They are implicitly marginalized out with a single sample. The inference problem is

$$q(\beta) \approx p(\beta \mid \mathbf{x}_{\text{train}}, \mathbf{z}^*)$$

where $\mathbf{z}^* \sim p(\mathbf{z} \mid \beta)$ is a prior sample.

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

For example, implicit prior samples are useful for generative adversarial networks. Their inference problem does not require any inference over the latent variables; it uses samples from the prior.

Classes of Inference

Inference is broadly classified under three classes: variational inference, Monte Carlo, and exact inference. We highlight how to use inference algorithms from each class.

As an example, we assume a mixture model with latent mixture assignments `z`, latent cluster means `beta`, and observations `x`:

$$p(\mathbf{x}, \mathbf{z}, \beta) = \text{Normal}(\mathbf{x} \mid \beta_{\mathbf{z}}, \mathbf{I}) \, \text{Categorical}(\mathbf{z} \mid \pi) \, \text{Normal}(\beta \mid \mathbf{0}, \mathbf{I}).$$

Variational Inference

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 write an approximating family,

$$q(\beta; \mu, \sigma) = \text{Normal}(\beta; \mu, \sigma),$$

$$q(\mathbf{z}; \pi) = \text{Categorical}(\mathbf{z}; \pi),$$

using TensorFlow variables to represent its parameters $\lambda = \{\pi, \mu, \sigma\}$.

```
from edward.models import Categorical, Normal

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})
```

Given an objective function, variational inference optimizes the family with respect to `tf.Variables`.

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 MAP estimation with an approximating family (`qbeta` and `qz`) of `PointMass` random variables, i.e., with all probability mass concentrated at a point.

```
from edward.models import PointMass

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

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

`MAP` inherits from `VariationalInference` and defines a loss function and update rules; it uses existing optimizers inside TensorFlow.

Monte Carlo

Monte Carlo approximates the posterior using samples (Robert and Casella, 1999). Monte Carlo is an inference where the approximating family is an empirical distribution,

$$q(\beta; \{\beta^{(t)}\}) = \frac{1}{T}\sum_{t=1}^{T} \delta(\beta, \beta^{(t)}),$$

$$q(\mathbf{z}; \{\mathbf{z}^{(t)}\}) = \frac{1}{T}\sum_{t=1}^{T} \delta(\mathbf{z}, \mathbf{z}^{(t)}).$$

The parameters are $\lambda = \{\beta^{(t)}, \mathbf{z}^{(t)}\}$.

```
from edward.models import Empirical

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})
```

Monte Carlo algorithms proceed by updating one sample $\beta^{(t)}, \mathbf{z}^{(t)}$ at a time in the empirical approximation. Monte Carlo algorithms proceed by updating one sample $\beta^{(t)}, \mathbf{z}^{(t)}$ at a time in the empirical approximation. Markov chain Monte Carlo does this sequentially to update the current sample (index t of `tf.Variables`) conditional on the last sample (index $t-1$ of `tf.Variables`). Specific Monte Carlo samplers determine the update rules; they can use gradients such as in Hamiltonian Monte Carlo (Neal, 2011) and graph structure such as in sequential Monte Carlo (Doucet et al., 2001).

Exact Inference

This approach also extends to algorithms that usually require tedious algebraic manipulation. With symbolic algebra on the nodes of the computational graph, we can uncover conjugacy relationships between random variables. Users can then integrate out variables to automatically derive classical Gibbs (Gelfand and Smith, 1990), mean-field updates (Bishop, 2006), and exact inference. (This is currently under development.)

Composing Inferences

Core to Edward's design is compositionality. Compositionality enables fine control of inference, where we can write inference as a collection of separate inference programs.

We outline how to write popular classes of compositional inferences using Edward: hybrid algorithms and message passing algorithms. We use the running example of a mixture model with latent mixture assignments `z`, latent cluster means `beta`, and observations `x`.

Hybrid algorithms

Hybrid algorithms leverage different inferences for each latent variable in the posterior. As an example, 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).

```
from edward.models import Categorical, PointMass

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()
```

In `data`, we include bindings of prior latent variables (`z` or `beta`) to posterior latent variables (`qz` or `qbeta`). This performs conditional inference, where only a subset of the posterior is inferred while the rest are fixed using other inferences.

This extends to many algorithms: for example, exact EM for exponential families; contrastive divergence (Hinton, 2002); pseudo-marginal and ABC methods (Andrieu and Roberts, 2009); Gibbs sampling within variational inference (Wang and Blei, 2012); Laplace variational inference (Wang and Blei, 2013); and structured variational auto-encoders (Johnson et al., 2016).

Message passing algorithms

Message passing algorithms operate on the posterior distribution using a collection of local inferences (Koller and Friedman, 2009). As an example, we demonstrate expectation propagation. We split a mixture model to be over two random variables `x1` and `x2` along with their latent mixture assignments `z1` and `z2`.

```
from edward.models import Categorical, Normal

N1 = 1000  # number of data points in first data set
N2 = 2000  # number of data points in second data set
D = 2  # data dimension
K = 5  # number of clusters

# MODEL
beta = Normal(mu=tf.zeros([K, D]), sigma=tf.ones([K, D]))
z1 = Categorical(logits=tf.zeros([N1, K]))
z2 = Categorical(logits=tf.zeros([N2, K]))
x1 = Normal(mu=tf.gather(beta, z1), sigma=tf.ones([N1, D]))
x2 = Normal(mu=tf.gather(beta, z2), sigma=tf.ones([N2, D]))

# INFERENCE
qbeta = Normal(mu=tf.Variable(tf.zeros([K, D])),
               sigma=tf.nn.softplus(tf.Variable(tf.zeros([K, D]))))
qz1 = Categorical(logits=tf.Variable(tf.zeros[N1, K]))
qz2 = Categorical(logits=tf.Variable(tf.zeros[N2, K]))

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()
```

We alternate updates for each local inference, where the global posterior factor $q(\beta)$ is shared across both inferences (Gelman et al., 2014).

With TensorFlow's distributed training, compositionality enables *distributed* message passing over a cluster with many workers. The computation can be further sped up with the use of GPUs via data and model parallelism.

This extends to many algorithms: for example, classical message passing, which performs exact local inferences; Gibbs sampling, which draws samples from conditionally conjugate inferences (Geman and Geman, 1984); expectation propagation, which locally minimizes $\text{KL}(p||q)$ over exponential families (Minka, 2001); integrated nested Laplace approximation, which performs local Laplace approximations (Rue et al., 2009); and all the instantiations of EP-like algorithms in Gelman et al. (2014).

In the above, we perform local inferences split over individual random variables. At the moment, Edward does not support local inferences within a random variable itself. We cannot do local inferences when representing the random variable for all data points and their cluster membership as `x` and `z` rather than `x1`, `x2`, `z1`, and `z2`.

Data Subsampling

Running algorithms which require the full data set for each update can be expensive when the data is large. In order to scale inferences, we can do *data subsampling*, i.e., update inference using only a subsample of data at a time. (Note that only certain algorithms can support data subsampling such as `MAP`, `KLqp`, and `SGLD`.)

Subgraphs

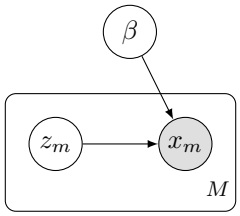

Figure 8: Data subsampling with a hierarchical model. We define a subgraph of the full model, forming a plate of size M rather than N.

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 example, consider a hierarchical model,

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

where there are latent variables z_n for each data point x_n (local variables) and latent variables β which are shared across data points (global variables).

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

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

More concretely, we define a mixture of Gaussians over D-dimensional data $\{x_n\} \in \mathbb{R}^{N \times D}$. There are K latent cluster means $\{\beta_k\} \in \mathbb{R}^{K \times D}$ and a membership assignment $z_n \in \{0, \dots, K-1\}$ for each data point x_n.

```
N = 10000000  # data set size
M = 128  # minibatch size
D = 2  # data dimensionality
K = 5  # number of clusters

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]))
```

For inference, 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 will perform inference with `KLqp`, a variational method that minimizes the divergence measure $\text{KL}(q\|p)$.

We instantiate two algorithms: a global inference over β given the subset of $\mathbf{z}$ and a local inference over the subset of $\mathbf{z}$ given β. We also pass in a TensorFlow placeholder `x_ph` for the data, so we can change the data at each step. (Alternatively, batch tensors can be used.)

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

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

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

Conceptually, the scale argument represents scaling for each random variableâĂŹs plate, as if we had seen that random variable N/M as many times.

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

```
qz_init = tf.initialize_variables([qz_variables])
for _ in range(1000):
  x_batch = next_batch(size=M)
  for _ in range(10):  # make local inferences
    inference_local.update(feed_dict={x_ph: x_batch})

  # update global parameters
  inference_global.update(feed_dict={x_ph: x_batch})
  # reinitialize the local factors
  qz_init.run()
```

After each iteration, we also reinitialize the parameters for $q(\mathbf{z} \mid \beta)$; this is because we do inference on a new set of local variational factors for each batch.

This demo readily applies to other inference algorithms such as `SGLD` (stochastic gradient Langevin dynamics): simply replace `qbeta` and `qz` with `Empirical` random variables; then call `ed.SGLD` instead of `ed.KLqp`.

Advanced settings

If the parameters fit in memory, one can avoid having to reinitialize local parameters or read/write from disk. To do this, define the full set of parameters and index them into the local posterior factors.

```
qz_variables = tf.Variable(tf.zeros([N, K]))
idx_ph = tf.placeholder(tf.int32, [M])
qz = Categorical(logits=tf.gather(qz_variables, idx_ph))
```

We define an index placeholder `idx_ph`. It will be fed index values at runtime to determine which parameters correspond to a given data subsample.

An alternative approach to reduce memory complexity is to use an inference network (Dayan et al., 1995), also known as amortized inference (Stuhlmüller et al., 2013). This can be applied using a global parameterization of $q(\mathbf{z}, \beta)$.

In streaming data, or online inference, the size of the data N may be unknown, or conceptually the size of the data may be infinite and at any time in which we query parameters from the online algorithm, the outputted parameters are from having processed as many data points up to that time. The approach of Bayesian filtering (Doucet et al., 2000; Broderick et al., 2013) can be applied in Edward using recursive posterior inferences; the approach of population posteriors (McInerney et al., 2015) is readily applicable from the subgraph setting.

In other settings, working on a subgraph of the model does not apply, such as in time series models when we want to preserve dependencies across time steps in our variational model. Approaches in the literature can be applied in Edward (Binder et al., 1997; Johnson and Willsky, 2014; Foti et al., 2014).

Development of Inference Methods

Edward uses class inheritance to provide a hierarchy of inference methods. This enables fast experimentation on top of existing algorithms, whether it be developing new black box algorithms or new model-specific algorithms.

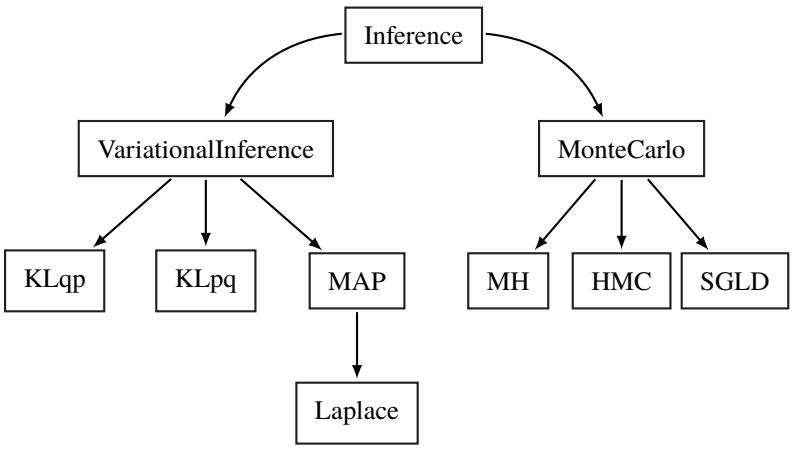

Figure 9: Dependency graph of inference methods. Nodes are classes in Edward and arrows represent class inheritance.

There is a base class `Inference`, from which all inference methods are derived from. Note that `Inference` says nothing about the class of models that an algorithm must work with. One can build inference algorithms which are tailored to a restricted class of models available in Edward (such as

differentiable models or conditionally conjugate models), or even tailor it to a single model. The algorithm can raise an error if the model is outside this class.

We organize inference under two paradigms: `VariationalInference` and `MonteCarlo` (or more plainly, optimization and sampling). These inherit from `Inference` and each have their own default methods.

For example, developing a new variational inference algorithm is as simple as inheriting from `VariationalInference` and writing a `build_loss_and_gradients()` method. `VariationalInference` implements many default methods such as `initialize()` with options for an optimizer. For example, see the importance weighted variational inference script.[6]

3.4 Criticism

We can never validate whether a model is true. In practice, "all models are wrong" (Box, 1976). However, we can try to uncover where the model goes wrong. Model criticism helps justify the model as an approximation or point to good directions for revising the model.

Edward explores model criticism using

- point-based evaluations, such as mean squared error or classification accuracy;
- posterior predictive checks, for making probabilistic assessments of the model fit using discrepancy functions.

We describe them in detail below.

Point-based evaluations

A point-based evaluation is a scalar-valued metric for assessing trained models (Winkler, 1996; Gneiting and Raftery, 2007). For example, we can assess models for classification by predicting the label for each observation in the data and comparing it to their true labels. Edward implements a variety of metrics, such as classification error and mean absolute error.

Formally, point prediction in probabilistic models is given by taking the mean of the posterior predictive distribution,

$$p(\mathbf{x}_{\text{new}} \mid \mathbf{x}) = \int p(\mathbf{x}_{\text{new}} \mid \mathbf{z}) p(\mathbf{z} \mid \mathbf{x}) \mathrm{d}\mathbf{z}.$$

The model's posterior predictive can be used to generate new data given past observations and can also make predictions on new data given past observations. It is formed by calculating the likelihood of the new data, averaged over every set of latent variables according to the posterior distribution.

Implementation

To evaluate inferred models, we first form the posterior predictive distribution. A helpful utility function for this is `copy`. For example, assume the model defines a likelihood `x` connected to a prior `z`. The posterior predictive distribution is

```
x_post = ed.copy(x, {z: qz})
```

[6] https://github.com/blei-lab/edward/blob/master/examples/iwvi.py

Here, we copy the likelihood node `x` in the graph and replace dependence on the prior `z` with dependence on the inferred posterior `qz`.

The `ed.evaluate()` method takes as input a set of metrics to evaluate, and a data dictionary. As with inference, the data dictionary binds the observed random variables in the model to realizations: in this case, it is the posterior predictive random variable of outputs `y_post` to `y_train` and a placeholder for inputs `x` to `x_train`.

```
ed.evaluate('categorical_accuracy', data={y_post: y_train, x: x_train})
ed.evaluate('mean_absolute_error', data={y_post: y_train, x: x_train})
```

The `data` can be data held-out from training time, making it easy to implement cross-validated techniques.

Point-based evaluation applies generally to any setting, including unsupervised tasks. For example, we can evaluate the likelihood of observing the data.

```
ed.evaluate('log_likelihood', data={x_post: x_train})
```

It is common practice to criticize models with data held-out from training. To do this, we first perform inference over any local latent variables of the held-out data, fixing the global variables. Then we make predictions on the held-out data.

```
from edward.models import Categorical

# create local posterior factors for test data, assuming test data
# has N_test many data points
qz_test = Categorical(logits=tf.Variable(tf.zeros[N_test, K]))

# run local inference conditional on global factors
inference_test = ed.Inference({z: qz_test}, data={x: x_test, beta: qbeta})
inference_test.run()

# build posterior predictive on test data
x_post = ed.copy(x, {z: qz_test, beta: qbeta})
ed.evaluate('log_likelihood', data={x_post: x_test})
```

Point-based evaluations are formally known as scoring rules in decision theory. Scoring rules are useful for model comparison, model selection, and model averaging.

Posterior predictive checks

Posterior predictive checks (PPCs) analyze the degree to which data generated from the model deviate from data generated from the true distribution. They can be used either numerically to quantify this degree, or graphically to visualize this degree. PPCs can be thought of as a probabilistic generalization of point-based evaluations (Box, 1980; Rubin, 1984; Meng, 1994; Gelman et al., 1996).

PPCs focus on the posterior predictive distribution

$$p(\mathbf{x}_{\text{new}} \mid \mathbf{x}) = \int p(\mathbf{x}_{\text{new}} \mid \mathbf{z}) p(\mathbf{z} \mid \mathbf{x}) \mathrm{d}\mathbf{z}.$$

The simplest PPC works by applying a test statistic on new data generated from the posterior predictive, such as $T(\mathbf{x}_{\text{new}}) = \max(\mathbf{x}_{\text{new}})$. Applying $T(\mathbf{x}_{\text{new}})$ to new data over many data replications induces a distribution. We compare this distribution to the test statistic applied to the real data $T(\mathbf{x})$.

In the figure, $T(\mathbf{x})$ falls in a low probability region of this reference distribution. This indicates that the model fits the data poorly according to this check; this suggests an area of improvement for the model.

More generally, the test statistic can also be a function of the model's latent variables $T(\mathbf{x}, \mathbf{z})$, known as a discrepancy function. Examples of discrepancy functions are the metrics used for point-based

evaluation. We can now interpret the point-based evaluation as a special case of PPCs: it simply calculates $T(\mathbf{x}, \mathbf{z})$ over the real data and without a reference distribution in mind. A reference distribution allows us to make probabilistic statements about the point, in reference to an overall distribution.

Implementation

To evaluate inferred models, we first form the posterior predictive distribution. A helpful utility function for this is `copy`. For example, assume the model defines a likelihood `x` connected to a prior `z`. The posterior predictive distribution is

```
x_post = ed.copy(x, {z: qz})
```

Here, we copy the likelihood node `x` in the graph and replace dependence on the prior `z` with dependence on the inferred posterior `qz`.

The `ed.ppc()` method provides a scaffold for studying various discrepancy functions.

```
def T(xs, zs):
  return tf.reduce_mean(xs[x_post])

ed.ppc(T, data={x_post: x_train})
```

The discrepancy can also take latent variables as input, which we pass into the PPC.

```
def T(xs, zs):
  return tf.reduce_mean(tf.cast(zs['z'], tf.float32))

ppc(T, data={y_post: y_train, x_ph: x_train},
    latent_vars={'z': qz, 'beta': qbeta})
```

PPCs are an excellent tool for revising models, simplifying or expanding the current model as one examines how well it fits the data. They are inspired by prior checks and classical hypothesis testing, under the philosophy that models should be criticized under the frequentist perspective of large sample assessment.

PPCs can also be applied to tasks such as hypothesis testing, model comparison, model selection, and model averaging. It's important to note that while they can be applied as a form of Bayesian hypothesis testing, hypothesis testing is generally not recommended: binary decision making from a single test is not as common a use case as one might believe. We recommend performing many PPCs to get a holistic understanding of the model fit.

4 End-to-end Examples

4.1 Bayesian Linear Regression

In supervised learning, the task is to infer hidden structure from labeled data, comprised of training examples $\{(x_n, y_n)\}$. Regression (typically) means the output y takes continuous values.

Data

Simulate training and test sets of 500 data points. They comprise pairs of inputs $\mathbf{x}_n \in \mathbb{R}^5$ and outputs $y_n \in \mathbb{R}$. They have a linear dependence of

$$\mathbf{w}_{\text{true}} = (-1.25, 4.51, 2.32, 0.99, -3.44).$$

with normally distributed noise.

```
def build_toy_dataset(N, w, noise_std=0.1):
  D = len(w)
  x = np.random.randn(N, D).astype(np.float32)
  y = np.dot(x, w) + np.random.normal(0, noise_std, size=N)
  return x, y

N = 500  # number of  data points
D = 5  # number of  features

w_true = 10 * (np.random.rand(D) - 0.5)
X_train, y_train = build_toy_dataset(N, w_true)
X_test, y_test = build_toy_dataset(N, w_true)
```

Model

Posit the model as Bayesian linear regression. It relates outputs $y \in \mathbb{R}$, also known as the response, given a vector of inputs $\mathbf{x} \in \mathbb{R}^D$, also known as the features or covariates. The model assumes a linear relationship between these two random variables (Murphy, 2012).

For a set of N data points $(\mathbf{X}, \mathbf{y}) = \{(\mathbf{x}_n, y_n)\}$, the model posits the following conditional relationships:

$$p(\mathbf{w}) = \text{Normal}(\mathbf{w} \mid \mathbf{0}, \sigma_w^2 \mathbf{I}),$$

$$p(b) = \text{Normal}(b \mid 0, \sigma_b^2),$$

$$p(\mathbf{y} \mid \mathbf{w}, b, \mathbf{X}) = \prod_{n=1}^{N} \text{Normal}(y_n \mid \mathbf{x}_n^\top \mathbf{w} + b, \sigma_y^2).$$

The latent variables are the linear model's weights $\mathbf{w}$ and intercept b, also known as the bias. Assume σ_w^2, σ_b^2 are known prior variances and σ_y^2 is a known likelihood variance. The mean of the likelihood is given by a linear transformation of the inputs $\mathbf{x}_n$.

```
X = tf.placeholder(tf.float32, [N, D])
w = Normal(mu=tf.zeros(D), sigma=tf.ones(D))
b = Normal(mu=tf.zeros(1), sigma=tf.ones(1))
y = Normal(mu=ed.dot(X, w) + b, sigma=tf.ones(N))
```

Inference

We now turn to inferring the posterior using variational inference. Define the variational model to be a fully factorized normal across the weights.

```
qw = Normal(mu=tf.Variable(tf.random_normal([D])),
            sigma=tf.nn.softplus(tf.Variable(tf.random_normal([D]))))
qb = Normal(mu=tf.Variable(tf.random_normal([1])),
            sigma=tf.nn.softplus(tf.Variable(tf.random_normal([1]))))
```

Run variational inference with the Kullback-Leibler divergence, using a default of 1000 iterations.

```
inference = ed.KLqp({w: qw, b: qb}, data={X: X_train, y: y_train})
inference.run()
```

In this case `KLqp` defaults to minimizing the $\text{KL}(q\|p)$ divergence measure using the reparameterization gradient. Minimizing this divergence metric is equivalent to maximizing the evidence lower bound (ELBO). Figure 10 shows the progression of the ELBO across iterations; variational inference appears to converge in approximately 200 iterations.

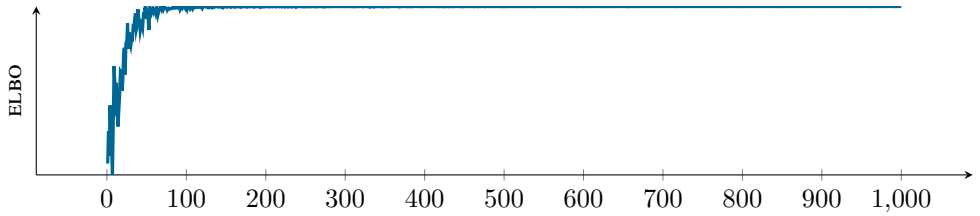

Figure 10: The evidence lower bound (ELBO) as a function of iterations. Variational inference maximizes this quantity iteratively; in this case, the algorithm appears to have converged in approximately 200 iterations.

Figure 11 shows the resulting posteriors from variational inference. We plot the marginal posteriors for each component of the vector of coefficients β. The vertical lines indicate the "true" values of the coefficients that we simulated above.

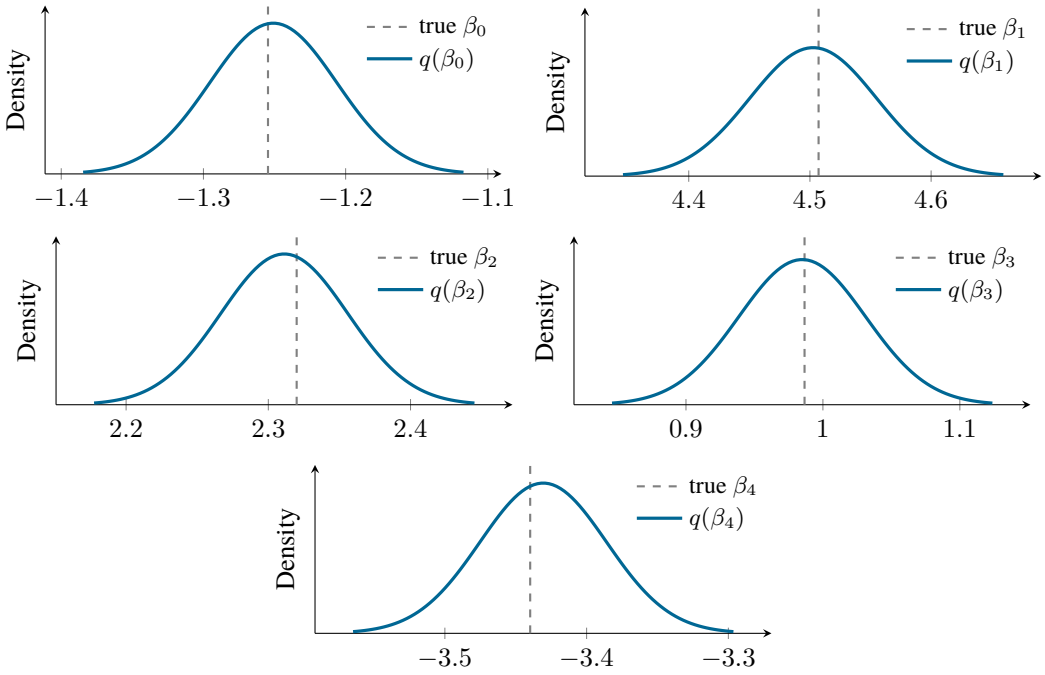

Figure 11: Visualization of the inferred marginal posteriors for Bayesian linear regression. The gray bars indicate the simulated "true" value for each component of the coefficient vector.

Criticism

A standard evaluation in regression is to calculate point-based evaluations on held-out "testing" data. We do this first by forming the posterior predictive distribution.

```
y_post = Normal(mu=ed.dot(X, qw.mean()) + qb.mean(), sigma=tf.ones(N))
```

With this we can evaluate various point-based quantities using the posterior predictive.

```
print(ed.evaluate('mean_squared_error', data={X: X_test, y_post: y_test}))
> 0.012107

print(ed.evaluate('mean_absolute_error', data={X: X_test, y_post: y_test}))
> 0.0867875
```

The trained model makes predictions with low mean squared error (relative to the magnitude of the output).

Edward supports another class of criticism techniques called posterior predictive checks (PPCs). The simplest PPC works by applying a test statistic on new data generated from the posterior predictive, such as $T(\mathbf{x}_{\text{new}}) = \max(\mathbf{x}_{\text{new}})$. Applying $T(\mathbf{x}_{\text{new}})$ to new data over many data replications induces a distribution of the test statistic, $\text{PPD}(T)$. We compare this distribution to the test statistic applied to the original dataset $T(\mathbf{x})$.

Calculating PPCs in Edward is straightforward.

```
def T(xs, zs):
  return tf.reduce_max(xs[y_post])

ppc_max = ed.ppc(T, data={X: X_train, y_post: y_train})
```

This calculates the test statistic on both the original dataset as well as on data replications generated from teh posterior predictive distribution. Figure 12 shows three visualizations of different PPCs; the plotted posterior predictive distributions are kernel density estimates from $N = 500$ data replications.

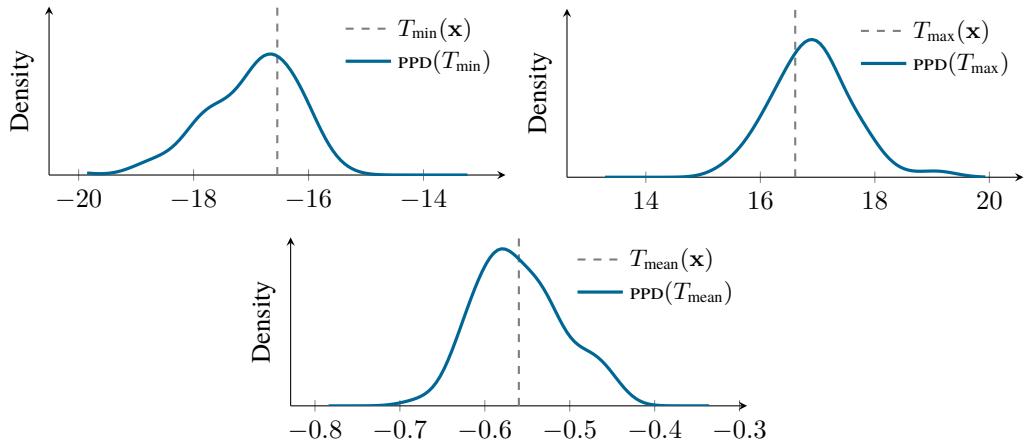

Figure 12: Examples of posterior predictive checks (PPCs) for Bayesian linear regression.

4.2 Logistic and Neural Network Classification

In supervised learning, the task is to infer hidden structure from labeled data, comprised of training examples $\{(x_n, y_n)\}$. Classification means the output y takes discrete values.

Data

We study a two-dimensional simulated dataset with a nonlinear decision boundary. We simulate 100 datapoints using the following snippet.

```
from scipy.stats import logistic

N = 100  # number of data points
D = 2  # number of features

px1 = np.linspace(-3, 3, 50)
px2 = np.linspace(-3, 3, 50)
px1_m, px2_m = np.mgrid[-3:3:50j, -3:3:50j]

xeval = np.vstack((px1_m.flatten(), px2_m.flatten())).T
x_viz = tf.constant(np.array(xeval, dtype='float32'))

def build_toy_dataset(N):
  x = xeval[np.random.randint(xeval.shape[0],size=N), :]
  y = bernoulli.rvs(p=logistic.cdf( 5 * x[:, 0]**2 + 5 * x[:, 1]**3 ))
  return x, y

x_train, y_train = build_toy_dataset(N)
```

Figure 13 shows the data, colored by label. The red point near the origin makes this a challenging dataset for classification models that assume a linear decision boundary.

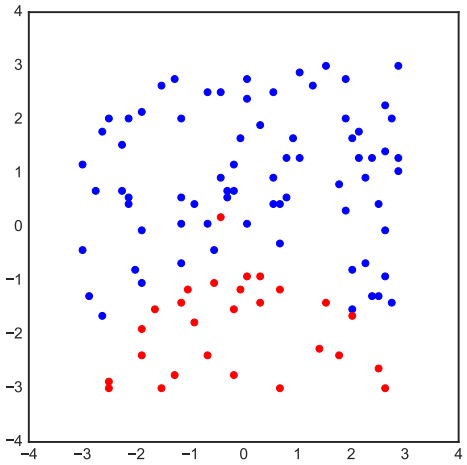

Figure 13: Simulated data for classification. Positive and negative measurements colored by label.

Model: Bayesian Logistic Regression

We begin with a popular classification model: logistic regression. This model relates outputs $y \in \{0, 1\}$, also known as the response, given a vector of inputs $\mathbf{x} \in \mathbb{R}^D$, also known as the features or covariates. The model assumes a latent linear relationship between these two random variables (Gelman et al., 2013).

The likelihood of each datapoint is a Bernoulli with probability

$$\Pr(y_n = 1) = \text{logistic}\left(\mathbf{x}^\top \mathbf{w} + b\right).$$

We posit priors on the latent variables $\mathbf{w}$ and b as

$$p(\mathbf{w}) = \text{Normal}(\mathbf{w} \mid \mathbf{0}, \sigma_w^2 \mathbf{I}),$$
$$p(b) = \text{Normal}(b \mid 0, \sigma_b^2).$$

This model is easy to specify in Edward's native language.

```
W = Normal(mu=tf.zeros(D), sigma=tf.ones(D))
b = Normal(mu=tf.zeros(1), sigma=tf.ones(1))

x = tf.cast(x_train, dtype=tf.float32)
y = Bernoulli(logits=(ed.dot(x, W) + b))
```

Inference

Here, we perform variational inference. Define the variational model to be a fully factorized normal

```
qW = Normal(mu=tf.Variable(tf.random_normal([D])),
            sigma=tf.nn.softplus(tf.Variable(tf.random_normal([D]))))
qb = Normal(mu=tf.Variable(tf.random_normal([1])),
            sigma=tf.nn.softplus(tf.Variable(tf.random_normal([1]))))
```

Run variational inference with the Kullback-Leibler divergence for 1000 iterations.

```
inference = ed.KLqp({W: qW, b: qb}, data={y: y_train})
inference.run(n_iter=1000, n_print=100, n_samples=5)
```

In this case `KLqp` defaults to minimizing the $\text{KL}(q\|p)$ divergence measure using the reparameterization gradient.

Criticism

The first thing to look at are point-wise evaluations on the training dataset.

First form a plug-in estimate of the posterior predictive distribution.

```
y_post = ed.copy(y, {W: qW.mean(), b: qb.mean()})
```

Then evaluate predictive accuracy

```
print('Plugin estimate of posterior predictive log accuracy on training data:')
print(ed.evaluate('log_lik', data={x: x_train, y_post: y_train}))
> -3.12

print('Binary accuracy on training data:')
print(ed.evaluate('binary_accuracy', data={x: x_train, y_post: y_train}))
> 0.71
```

Figure 14 shows the posterior label probability evaluated on a grid. As expected, logistic regression attempts to fit a linear boundary between the two label classes. Can a non-linear model do better?

Model: Bayesian Neural Network Classification

Consider parameterizing the label probability using a neural network; this model is not limited to a linear relationship to the inputs $\mathbf{x}$, as in logistic regression.

The model posits a likelihood for each observation $(\mathbf{x}_n, y_n)$ as

$$\Pr(y_n = 1) = \text{logistic}\left(\text{NN}(\mathbf{x}_n\ ;\ \mathbf{z})\right),$$

where NN is a neural network and the latent random variable $\mathbf{z}$ contains its weights and biases.

We can specify a Bayesian neural network in Edward as follows. Here we specify a fully connected two-layer network with two nodes in each layer; we posit standard normal priors on all weights and biases.

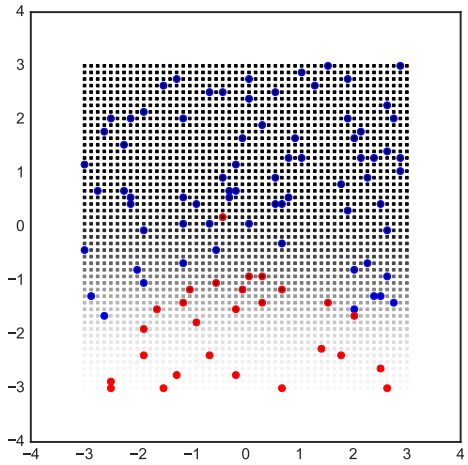

Figure 14: Logistic regression struggles to separate the measurements.

```
def neural_network(x, W_0, W_1, W_2, b_0, b_1, b_2):
    h = tf.nn.tanh(tf.matmul(x, W_0) + b_0)
    h = tf.nn.tanh(tf.matmul(h, W_1) + b_1)
    h = tf.matmul(h, W_2) + b_2
    return tf.reshape(h, [-1])

H = 2  # number of hidden units in each layer

W_0 = Normal(mu=tf.zeros([D, H]), sigma=tf.ones([D, H]))
W_1 = Normal(mu=tf.zeros([H, H]), sigma=tf.ones([H, H]))
W_2 = Normal(mu=tf.zeros([H, 1]), sigma=tf.ones([H, 1]))
b_0 = Normal(mu=tf.zeros(H), sigma=tf.ones(H))
b_1 = Normal(mu=tf.zeros(H), sigma=tf.ones(H))
b_2 = Normal(mu=tf.zeros(1), sigma=tf.ones(1))

x = tf.cast(x_train, dtype=tf.float32)
y = Bernoulli(logits=neural_network(x, W_0, W_1, W_2, b_0, b_1, b_2))
```

Inference

Similar to the above, we perform variational inference. Define the variational model to be a fully factorized normal over all latent variables

```
qW_0 = Normal(mu=tf.Variable(tf.random_normal([D, H])),
              sigma=tf.nn.softplus(tf.Variable(tf.random_normal([D, H]))))
qW_1 = Normal(mu=tf.Variable(tf.random_normal([H, H])),
              sigma=tf.nn.softplus(tf.Variable(tf.random_normal([H, H]))))
qW_2 = Normal(mu=tf.Variable(tf.random_normal([H, 1])),
              sigma=tf.nn.softplus(tf.Variable(tf.random_normal([H, 1]))))
qb_0 = Normal(mu=tf.Variable(tf.random_normal([H])),
              sigma=tf.nn.softplus(tf.Variable(tf.random_normal([H]))))
qb_1 = Normal(mu=tf.Variable(tf.random_normal([H])),
              sigma=tf.nn.softplus(tf.Variable(tf.random_normal([H]))))
qb_2 = Normal(mu=tf.Variable(tf.random_normal([1])),
              sigma=tf.nn.softplus(tf.Variable(tf.random_normal([1]))))
```

Run variational inference for 1000 iterations.

```
inference = ed.KLqp({W_0: qW_0, b_0: qb_0,
                     W_1: qW_1, b_1: qb_1,
                     W_2: qW_2, b_2: qb_2}, data={y: y_train})
inference.run(n_iter=1000, n_print=100, n_samples=5)
```

Criticism

Again, we form a plug-in estimate of the posterior predictive distribution.

```
y_post = ed.copy(y, {W_0: qW_0.mean(), b_0: qb_0.mean(),
                     W_1: qW_1.mean(), b_1: qb_1.mean(),
                     W_2: qW_2.mean(), b_1: qb_2.mean()})
```

Both predictive accuracy metrics look better.

```
print('Plugin estimate of posterior predictive log accuracy on training data:')
print(ed.evaluate('log_lik', data={x: x_train, y_post: y_train}))
> -0.170941

print('Binary accuracy on training data:')
print(ed.evaluate('binary_accuracy', data={x: x_train, y_post: y_train}))
> 0.81
```

Figure 15 shows the posterior label probability evaluated on a grid. The neural network has captured the nonlinear decision boundary between the two label classes.

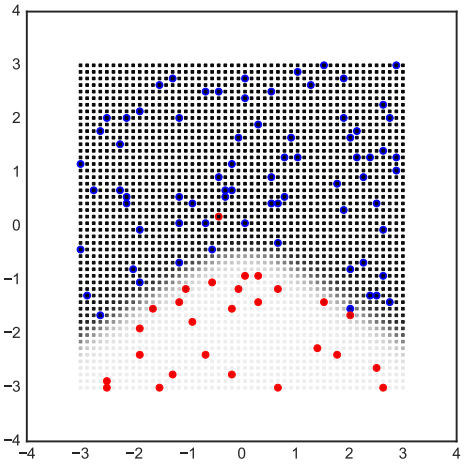

Figure 15: A Bayesian neural network does a better job of separating the two label classes.

5 Acknowledgments

Edward has benefited enormously from the helpful feedback and advice of many individuals: Jaan Altosaar, Eugene Brevdo, Allison Chaney, Joshua Dillon, Matthew Hoffman, Kevin Murphy, Rajesh Ranganath, Rif Saurous, and other members of the Blei Lab, Google Brain, and Google Research. 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, 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.org*.

Al-Rfou, R., Alain, G., and Almahairi, A. (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv.org*.

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

Becker, R. A. and Chambers, J. M. (1984). *S: an interactive environment for data analysis and graphics*. CRC Press.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828.

Binder, J., Murphy, K., and Russell, S. (1997). Space-efficient inference in dynamic probabilistic networks. In *International Joint Conference on Artificial Intelligence*.

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

Blei, D. M. (2014). Build, compute, critique, repeat: Data analysis with latent variable models. *Annual Review of Statistics and Its Application*.

Box, G. E. (1980). Sampling and Bayes' inference in scientific modelling and robustness. *Journal of the Royal Statistical Society. Series A (General)*, pages 383–430.

Box, G. E. and Hill, W. J. (1967). Discrimination among mechanistic models. *Technometrics*, 9(1):57–71.

Box, G. E. and Hunter, W. G. (1965). The experimental study of physical mechanisms. *Technometrics*, 7(1):23–42.

Box, G. E. P. (1976). Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799.

Box, G. E. P. and Hunter, W. G. (1962). A useful method for model-building. *Technometrics*, 4(3):301–318.

Broderick, T., Boyd, N., Wibisono, A., Wilson, A. C., and Jordan, M. I. (2013). Streaming variational Bayes. In *Neural Information Processing Systems*.

Buchanan, B., Sutherland, G., and Feigenbaum, E. A. (1969). *Heuristic DENDRAL: a program for generating explanatory hypotheses in organic chemistry*. American Elsevier.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2016). Stan: a probabilistic programming language. *Journal of Statistical Software*.

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., and Betancourt, M. (2015). The Stan Math Library: Reverse-Mode Automatic Differentiation in C++. *arXiv.org*.

Chambers, J. M. and Hastie, T. J. (1992). *Statistical Models in S*. Chapman & Hall, London.

Chollet, F. (2015). Keras. https://github.com/fchollet/keras.

Collobert, R. and Kavukcuoglu, K. (2011). Torch7: A matlab-like environment for machine learning. In *Neural Information Processing Systems*.

Dayan, P. and Abbott, L. F. (2001). *Theoretical neuroscience*, volume 10. Cambridge, MA: MIT Press.

Dayan, P., Hinton, G. E., Neal, R. M., and Zemel, R. S. (1995). The helmholtz machine. *Neural computation*, 7(5):889–904.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38.

Dieleman, S., SchlÃijter, J., Raffel, C., Olson, E., SÃÿnderby, S. K., Nouri, D., Maturana, D., Thoma, M., Battenberg, E., Kelly, J., Fauw, J. D., Heilman, M., de Almeida, D. M., McFee, B., Weideman, H., TakÃącs, G., de Rivaz, P., Crall, J., Sanders, G., Rasul, K., Liu, C., French, G., and Degrave, J. (2015). Lasagne: First release.

Doucet, A., De Freitas, N., and Gordon, N. (2001). An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer.

Doucet, A., Godsill, S., and Andrieu, C. (2000). On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, 10(3):197–208.

Foti, N., Xu, J., Laird, D., and Fox, E. (2014). Stochastic variational inference for hidden Markov models. In *Neural Information Processing Systems*.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin.

Friedman, N., Linial, M., Nachman, I., and Pe'er, D. (2000). Using bayesian networks to analyze expression data. *Journal of computational biology*, 7(3-4):601–620.

Gelfand, A. E. and Smith, A. F. (1990). Sampling-based approaches to calculating marginal densities. *Journal of the American statistical association*, 85(410):398–409.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2013). *Bayesian data analysis*. Texts in Statistical Science Series. CRC Press, Boca Raton, FL, third edition.

Gelman, A., Meng, X.-L., and Stern, H. (1996). Posterior predictive assessment of model fitness via realized discrepancies. *Statistica sinica*, pages 733–760.

Gelman, A., Vehtari, A., Jylänki, P., Robert, C., Chopin, N., and Cunningham, J. P. (2014). Expectation propagation as a way of life. *arXiv preprint arXiv:1412.4869*.

Geman, S. and Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6):721–741.

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

Gneiting, T. and Raftery, A. E. (2007). Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

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*.

Goodman, N. D. and Stuhlmüller, A. (2014). The Design and Implementation of Probabilistic Programming Languages. http://dippl.org.

Hastings, W. K. (1970). Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109.

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

Hinton, G. E. and van Camp, D. (1993). Keeping the neural networks simple by minimizing the description length of the weights. In *Conference on Learning Theory*, pages 5–13, New York, New York, USA. ACM.

Hjort, N. L., Holmes, C., Müller, P., and Walker, S. G. (2010). *Bayesian nonparametrics*, volume 28. Cambridge University Press.

Hoffman, M. D., Blei, D. M., Wang, C., and Paisley, J. (2013). Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558.

Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.

Johnson, M. and Willsky, A. S. (2014). Stochastic variational inference for Bayesian time series models. In *International Conference on Machine Learning*.

Johnson, M. J., Duvenaud, D., Wiltschko, A. B., Datta, S. R., and Adams, R. P. (2016). Composing graphical models with neural networks for structured representations and fast inference. *arXiv preprint arXiv:1603.06277*.

Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., and Saul, L. K. (1999). An Introduction to Variational Methods for Graphical Models. *Machine Learning*, pages 1–51.

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.org*.

MacKay, D. J. (2003). *Information theory, inference and learning algorithms*. Cambridge university press.

Manning, C. D. and Schütze, H. (1999). *Foundations of statistical natural language processing*, volume 999. MIT Press.

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

McInerney, J., Ranganath, R., and Blei, D. M. (2015). The Population Posterior and Bayesian Inference on Streams. In *Neural Information Processing Systems*.

Meng, X.-L. (1994). Posterior predictive p-values. *The Annals of Statistics*, pages 1142–1160.

Metropolis, N. and Ulam, S. (1949). The Monte Carlo method. *Journal of the American statistical association*, 44(247):335–341.

Minka, T. P. (2001). Expectation propagation for approximate bayesian inference. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 362–369. Morgan Kaufmann Publishers Inc.

Minsky, M. (1975). A framework for representing knowledge.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

Neal, R. M. (1990). Learning Stochastic Feedforward Networks. Technical report.

Neal, R. M. (1993). Probabilistic Inference using Markov Chain Monte Carlo Methods. Technical report.

Neal, R. M. (2011). MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*.

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.

Nervana Systems (2014). neon. http://dippl.org.

Newell, A. and Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126.

Pearl, J. (1988). Probabilistic reasoning in intelligent systems: Networks of plausible inference.

Ranganath, R., Gerrish, S., and Blei, D. M. (2014). Black Box Variational Inference. In *Artificial Intelligence and Statistics*.

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.

Rubin, D. B. (1984). Bayesianly justifiable and relevant frequency calculations for the applied statistician. *The Annals of Statistics*, 12(4):1151–1172.

Rue, H., Martino, S., and Chopin, N. (2009). Approximate bayesian inference for latent gaussian models by using integrated nested laplace approximations. *Journal of the royal statistical society: Series b (statistical methodology)*, 71(2):319–392.

Rumelhart, D. E., McClelland, J. L., Group, P. R., et al. (1988). *Parallel distributed processing*, volume 1. IEEE.

Spiegelhalter, D. J., Thomas, A., Best, N. G., and Gilks, W. R. (1995). BUGS: Bayesian inference using Gibbs sampling, version 0.50. *MRC Biostatistics Unit, Cambridge*.

Stuhlmüller, A., Taylor, J., and Goodman, N. (2013). Learning stochastic inverses. In *Advances in neural information processing systems*, pages 3048–3056.

Tenenbaum, J. B., Kemp, C., Griffiths, T. L., and Goodman, N. D. (2011). How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022):1279–1285.

Tukey, J. W. (1962). The Future of Data Analysis. *Annals of Mathematical Statistics*, 33(1):1–67.

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

Wang, C. and Blei, D. M. (2013). Variational inference in nonconjugate models. *Journal of Machine Learning Research*, 14:1005–1031.

Waterhouse, S., MacKay, D., Robinson, T., et al. (1996). Bayesian methods for mixtures of experts. *Advances in neural information processing systems*, pages 351–357.

Welling, M. and Teh, Y. W. (2011). Bayesian learning via stochastic gradient Langevin dynamics. In *International Conference on Machine Learning*.

Winkler, R. L. (1996). Scoring rules and the evaluation of probabilities. *Test*, 5(1):1–60.

Wood, F., van de Meent, J. W., and Mansinghka, V. (2015). A New Approach to Probabilistic Programming Inference. *arXiv.org*.

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