Variational inference

Extensions and current topics

Helge Langseth and Thomas Dyhre Nielsen

Oct. 2018

Introduction

Plan for this weeks

Day 1: Bayesian networks – Definition and inference

- Definition of Bayesian networks: Syntax and semantics
- Exact inference
- Approximate inference using MCMC

Day 2: Variational inference – Introduction and basis

- Approximate inference through the *Kullback-Leibler divergence*
- *Variational Bayes*
- The *mean-field* approach to Variational Bayes

Day 3: Variational Bayes – cont'd

- Solving the VB equations
- Introducing Exponential families

Day 4: Scalable Variational Bayes

- Variational message passing
- Stochastic gradient ascent
- Stochastic variational inference

Day 5: Current approaches and extensions

- Variational Auto Encoders
- Black Box variational inference
- Probabilistic Programming Languages

Recap from last time

- The Exponential Family of distributions
- Variational Message Passing
- Stochastic approximations using Robbins-Monro
- Stochastic Variational Bayes

Our motivation for Bayesian models

Motivation

We seek to build models that:

- Reflect human understanding of a domain with a transparent model structure.
- Support a large (potentially unbounded) set of probabilistic models.
- Ability to capture fine structure in data.
- Sound semantics – both wrt. modelling language and interpretation of the generated results.
- Efficient inference algorithms – preferably with quality guarantees.
- Supported by a useful probabilistic programming language that allows simple implementation of these models.

Variational Auto-Encoders

Is a *Deep Neural Network* the solution?

Limits on the scope of deep learning*

Deep learning thus far [January 2018] . . .

- . . . is data hungry
- . . . has no natural way to deal with hierarchical structure
- . . . is not sufficiently transparent
- . . . has not been well integrated with prior knowledge
- . . . works well as an approximation, but **its answers often cannot be fully trusted**

* Gary Marcus: *Deep Learning: A Critical Appraisal*. arXiv:1801.00631 [cs.AI]

Is a *Deep Neural Network* the solution?

Limits on the scope of deep learning*

Deep learning thus far [January 2018] . . .

- . . . is data hungry
- . . . has no natural way to deal with hierarchical structure
- . . . is not sufficiently transparent
- . . . has not been well integrated with prior knowledge
- . . . works well as an approximation, but **its answers often cannot be fully trusted**

* Gary Marcus: *Deep Learning: A Critical Appraisal*. arXiv:1801.00631 [cs.AI]

Deep Bayesian Learning

A marriage of Bayesian thinking and deep learning is a framework that . . .

- . . . allows explicit modelling.
- . . . has a sound probabilistic foundation.
- . . . balances expert knowledge and information from data.
- . . . avoids restrictive assumptions about modelling families.
- . . . supports efficient inference.

The conditional distribution

- Recall that a Bayesian network specification includes the conditional probability distribution $p(x_i \mid \mathrm{pa}(x_i))$ for each variable X_i.
- Typically the CPD is assumed to belong to some distributional family out of convenience — e.g., to obtain conjugacy.
- Deep Bayesian models opens up for the CPDs to be represented through deep neural networks.

Building-blocks of a Variational Auto Encoder

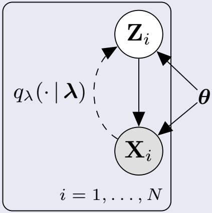
The conditional distribution

- Recall that a Bayesian network specification includes the conditional probability distribution $p(x_i \mid \mathrm{pa}(x_i))$ for each variable X_i.
- Typically the CPD is assumed to belong to some distributional family out of convenience — e.g., to obtain conjugacy.
- Deep Bayesian models opens up for the CPDs to be represented through deep neural networks.

The model structure

- Bayesian models often leverage from *latent variables*. These are variables $\mathbf{Z}$ that are unobserved, yet influence the observed variables $\mathbf{X}$.
- We therefore consider a model of two components:
  - $\mathbf{Z}$ follows some distribution $p_\theta(\mathbf{z} \mid \boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta}$.
  - $\mathbf{X} \mid \mathbf{Z}$ follows some distribution $p_\theta(\mathbf{x} \mid g_\theta(\mathbf{z}))$ where $g_\theta(\mathbf{z})$ is a function represented by a deep neural network.
- In VAE lingo, $\mathbf{Z}$ in a **coded** version of $\mathbf{X}$. Therefore, $p_\theta(\mathbf{x} \mid g_\theta(\mathbf{z}))$ is the **decoder** model. Similarly, the process $\mathbf{X} \rightsquigarrow \mathbf{Z}$ is the **encoder**.

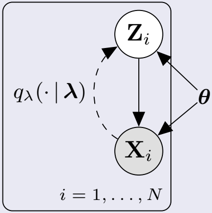
Model of interest



- We assume parametric distributions $p_\theta(\mathbf{z} \mid \boldsymbol{\theta})$ and $p_\theta(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$.
- No further assumptions are made about the generative model.
- We want to learn $\boldsymbol{\theta}$ to maximize the model's fit to the data-set $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$.
- Simultaneously we seek a variational approximation $q_\lambda(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\lambda})$ – parameterized by $\boldsymbol{\lambda}$.
- Notice that while VI approaches "typically" optimize $\boldsymbol{\lambda}$ for each $\mathbf{x}$, we here do **amortized inference**: Chose one $\boldsymbol{\lambda}$ **for all** $\mathbf{x}$, and define $q_\lambda(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\lambda})$ with $\mathbf{x}$ an explicit input to a DNN.

The Variational Auto Encoder (VAE)

Model of interest



- We assume parametric distributions $p_\theta(\mathbf{z} \mid \boldsymbol{\theta})$ and $p_\theta(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$.
- No further assumptions are made about the generative model.
- We want to learn $\boldsymbol{\theta}$ to maximize the model's fit to the data-set $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$.
- Simultaneously we seek a variational approximation $q_\lambda(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\lambda})$ – parameterized by $\boldsymbol{\lambda}$.
- Notice that while VI approaches "typically" optimize $\boldsymbol{\lambda}$ for each $\mathbf{x}$, we here do **amortized inference**: Chose one $\boldsymbol{\lambda}$ **for all** $\mathbf{x}$, and define $q_\lambda(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\lambda})$ with $\mathbf{x}$ an explicit input to a DNN.

Obvious strategy:

Optimize $\mathcal{L}(q)$ to choose $\boldsymbol{\lambda}$ and $\boldsymbol{\theta}$, where

$$\mathcal{L}(q) = -\mathbb{E}_{q_\lambda}\left[\log \frac{q_\lambda(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\lambda})}{p_\theta(\mathbf{z}, \mathbf{x} \mid \boldsymbol{\theta})}\right]$$

- We will parameterize $p_\theta(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$ as a DNN with inputs $\mathbf{z}$ and weights defined by $\boldsymbol{\theta}$;
- ... and $q_\lambda(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\lambda})$ as a DNN with inputs $\mathbf{x}$ and weights defined by $\boldsymbol{\lambda}$.

ELBO for VAEs

We rephrase the ELBO as follows:

First recall that

$$\mathcal{L}(q) \leq \log p_\theta(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \sum_{i=1}^{N} \log p_\theta(\mathbf{x}_i)$$

We will therefore now look at ELBO **for a single observation** $\mathbf{x}_i$ and later maximize the sum of these contributions. For a given $\mathbf{x}_i$ we get

$$\begin{aligned}
\mathcal{L}(\mathbf{x}_i) &= -\mathbb{E}_{q_\lambda}\left[\log \frac{q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i, \boldsymbol{\lambda})}{p_\theta(\mathbf{z}, \mathbf{x}_i\,|\,\boldsymbol{\theta})}\right] \\
&= -\mathbb{E}_{q_\lambda}\left[\log q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i, \boldsymbol{\lambda})\right] + \left\{\mathbb{E}_{q_\lambda}\left[\log p_\theta(\mathbf{z})\right] + \mathbb{E}_{q_\lambda}\left[\log p_\theta(\mathbf{x}_i\,|\,\mathbf{z}, \boldsymbol{\theta})\right]\right\} \\
&= -\mathrm{KL}\left(q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i, \boldsymbol{\lambda})||p_\theta(\mathbf{z})\right) + \mathbb{E}_{q_\lambda}\left[\log p_\theta(\mathbf{x}_i\,|\,\mathbf{z}, \boldsymbol{\theta})\right]
\end{aligned}$$

The two terms penalizes:

- . . . a posterior over $\mathbf{z}$ far from the prior $p_\theta(\mathbf{z})$
- . . . and poor reconstruction ability – averaged over $q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i, \boldsymbol{\lambda})$

Calculating the ELBO terms

$$\mathcal{L}(\mathbf{x}_i) = \boxed{-\,\mathrm{KL}\left(q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i,\boldsymbol{\lambda})||p_\theta(\mathbf{z})\right)} + \boxed{\mathbb{E}_{q_\lambda}\left[\log p_\theta(\mathbf{x}_i\,|\,\mathbf{z},\boldsymbol{\theta})\right]}$$

- The KL-term is dependent on the distributional families of $p_\theta(\mathbf{z})$ and $q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i,\boldsymbol{\lambda})$.
  - One can assume a simple shape, like:
    - $p_\theta(\mathbf{z})$ being Gaussian with zero mean and isotropic covariance;
    - $q_\lambda(z_\ell\,|\,\mathbf{x}_i,\boldsymbol{\lambda})$ is a Gaussian with mean and variance determined by a DNN.
  - Simplicity is **not required** as long as the KL can be calculated (numerically).

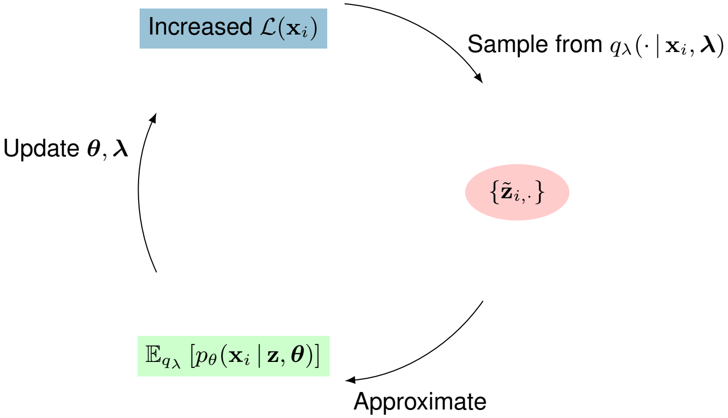
Calculating the ELBO terms

$$\mathcal{L}(\mathbf{x}_i) = \boxed{-\operatorname{KL}\left(q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i,\boldsymbol{\lambda})||p_\theta(\mathbf{z})\right)} + \boxed{\mathbb{E}_{q_\lambda}\left[\log p_\theta(\mathbf{x}_i\,|\,\mathbf{z},\boldsymbol{\theta})\right]}$$

- The KL-term is dependent on the distributional families of $p_\theta(\mathbf{z})$ and $q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i,\boldsymbol{\lambda})$.
  - One can assume a simple shape, like:
    - $p_\theta(\mathbf{z})$ being Gaussian with zero mean and isotropic covariance;
    - $q_\lambda(z_\ell\,|\,\mathbf{x}_i,\boldsymbol{\lambda})$ is a Gaussian with mean and variance determined by a DNN.
  - Simplicity is **not required** as long as the KL can be calculated (numerically).
- The reconstruction term involves two separate operations:
  - For a given $\mathbf{z}$ evaluate the log-probability of the data-point $\mathbf{x}_i$, $\log p_\theta(\mathbf{x}_i\,|\,\mathbf{z},\boldsymbol{\theta})$. The distribution is parameterized by a DNN, getting its weights from $\boldsymbol{\theta}$.
  - The expectation $\mathbb{E}_{q_\lambda}[\cdot]$ is approximated by a random sample that we generate from $q_\lambda(\mathbf{z}\,|\,\mathbf{x}_i,\boldsymbol{\lambda})$:

$$\mathbb{E}_{q_\lambda}\left[\log p_\theta(\mathbf{x}_i\,|\,\mathbf{z},\boldsymbol{\theta})\right] \approx \frac{1}{M}\sum_{j=1}^{M}\log p_\theta\left(\mathbf{x}_i\,|\,\tilde{\mathbf{z}}_{i,j},\boldsymbol{\theta}\right),$$

where $\tilde{\mathbf{Z}}_{i,j} \sim q_\lambda(\cdot\,|\,\mathbf{x}_i,\boldsymbol{\lambda})$.


Increased $\mathcal{L}(\mathbf{x}_i)$
Sample from $q_\lambda(\cdot \mid \mathbf{x}_i, \boldsymbol{\lambda})$
$\{\tilde{\mathbf{z}}_{i,\cdot}\}$
Approximate
$\mathbb{E}_{q_\lambda}\left[p_\theta(\mathbf{x}_i \mid \mathbf{z}, \boldsymbol{\theta})\right]$
Update $\boldsymbol{\theta}, \boldsymbol{\lambda}$

VAE implementation

Algorithm

1. Initialize $\boldsymbol{\lambda}$, $\boldsymbol{\theta}$
2. **Repeat**
   1. **For** $i = 1, \ldots, N$:
      1. Sample $\{\mathbf{z}_{i,1}, \ldots, \mathbf{z}_{i,M}\}$ from $q_\lambda(\cdot \mid \mathbf{x}_i, \boldsymbol{\lambda})$
      2. Approximate ELBO contribution by

$$\tilde{\mathcal{L}}(\mathbf{x}_i) = -\,\mathrm{KL}\left(q_\lambda(\mathbf{z} \mid \mathbf{x}_i, \boldsymbol{\lambda}) || p_\theta(\mathbf{z})\right) + \frac{1}{M}\sum_{j=1}^{M} \log p_\theta\left(\mathbf{x}_i \mid \tilde{\mathbf{z}}_{i,j}, \boldsymbol{\theta}\right)$$

   2. Update $\boldsymbol{\lambda}$, $\boldsymbol{\theta}$ using the approximate ELBO gradients found by

$$\nabla_{\lambda,\theta}\mathcal{L}\left(\mathcal{D}, \boldsymbol{\theta}, \boldsymbol{\lambda}\right) \approx \nabla_{\lambda,\theta}\sum_{i=1}^{N}\tilde{\mathcal{L}}(\mathbf{x}_i).$$

   Until convergence
3. **Return** $\boldsymbol{\lambda}$, $\boldsymbol{\theta}$

Simple implementation

Notice that variational learning is casted as a gradient ascent procedure. We can therefore utilize Tensorflow, Theano or other similar tools.

Python-code for training a VAE

```
def train(self, dataset, no_epochs, batch_size):
    print("--- Training starts ---")
    for epoch in range(no_epochs):
        avg_cost = 0
        total_batch = int(dataset.num_examples / batch_size)
        # Loop over all batches
        for i in range(total_batch):
            # Get data-batch
            batch_x, _ = dataset.next_batch(batch_size)
            # Send in data, optimize parameters, get loss
            _, cost = self.sess.run(
                 [self.train_op, self.loss],
                 feed_dict={self.input_data_placeholder: batch_x})
            # Compute average loss per epoch
            avg_cost += (cost / dataset.num_examples) * batch_size

        # Display loss per epoch
        print("Epoch: {:4d}: Loss: {:11.6f}".format(epoch,
              self.avg_free_energy_bound[epoch]))

    print("--- Training done ---")
```

Python-code for definition of learning objective

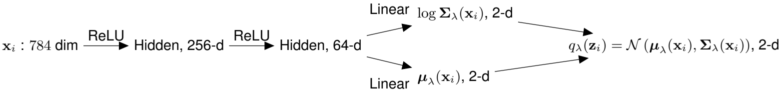
```
def _define_loss(self):
    with tf.name_scope('Loss'):
        with tf.name_scope('KL_divergence'):
            kl_loss = tf.reduce_sum(
                self.z.kl_divergence(tf.distributions.Normal(
                    loc=0., scale=1.)),
                axis=1)
        with tf.name_scope('Reconstruction_loss'):
            _prediction = self.data_reconstruction.mean()
            _reconstruction_loss = \
                - tf.reduce_sum(
                    self.input_data_placeholder * tf.log(_prediction)
                    + (1 - self.input_data_placeholder) *
                    tf.log(1 - _prediction), axis=1)

        _loss = tf.reduce_mean(tf.add(reconstruction_loss, kl_loss))
    self.loss = _loss
```

- The model is learned from $N = 55.000$ training examples.
- Each $\mathbf{x}_i$ is a binary vector of 784 pixel values.
- When seen as a 28×28 array, each $\mathbf{x}_i$ is a picture of a handwritten digit ("0" – "9")

- The model is learned from $N = 55.000$ training examples.
- Each $\mathbf{x}_i$ is a binary vector of 784 pixel values.
- When seen as a 28×28 array, each $\mathbf{x}_i$ is a picture of a handwritten digit ("0" – "9")

- Encoding is done in **two** dimensions. A priori $\mathbf{Z}_i \sim p_\theta(\mathbf{z}_i) = \mathcal{N}(\mathbf{0}_2, \mathbf{I}_2)$.
- The approximate expectation in the ELBO is calculated using $M = 1$ sample per data-point.
- The **encoder network** $\mathbf{X} \rightsquigarrow \mathbf{Z}$ is a $256 + 64$ neural net with ReLU units.
  - The 64 outputs go through a linear layer to define $\boldsymbol{\mu}_\lambda(\mathbf{x}_i)$ and $\log \boldsymbol{\Sigma}_\lambda(\mathbf{x}_i)$.
  - Finally, $q_\lambda(\mathbf{z}_i \,|\, \mathbf{x}_i, \boldsymbol{\lambda}) = \mathcal{N}(\boldsymbol{\mu}_\lambda(\mathbf{x}_i), \boldsymbol{\Sigma}_\lambda(\mathbf{x}_i))$.



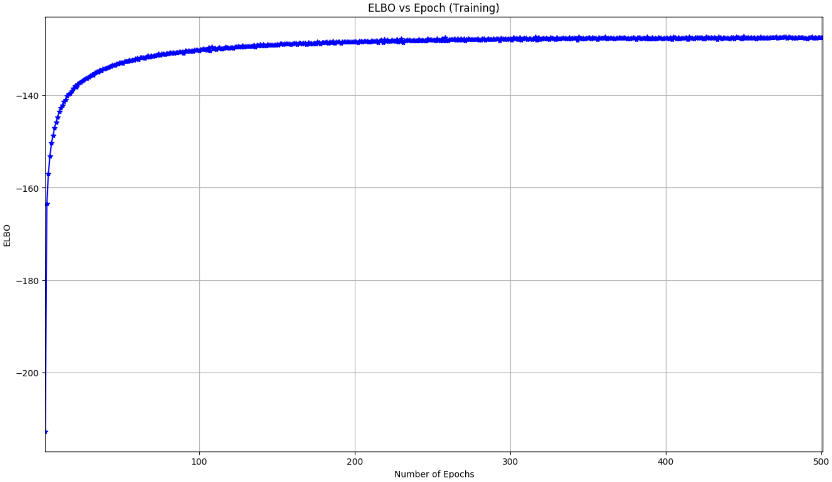
Fun with MNIST – The model

- The model is learned from $N = 55.000$ training examples.
- Each $\mathbf{x}_i$ is a binary vector of 784 pixel values.
- When seen as a 28×28 array, each $\mathbf{x}_i$ is a picture of a handwritten digit ("0" – "9")

- Encoding is done in **two** dimensions. A priori $\mathbf{Z}_i \sim p_\theta(\mathbf{z}_i) = \mathcal{N}(\mathbf{0}_2, \mathbf{I}_2)$.
- The approximate expectation in the ELBO is calculated using $M = 1$ sample per data-point.
- The **encoder network** $\mathbf{X} \rightsquigarrow \mathbf{Z}$ is a $256 + 64$ neural net with ReLU units.
  - The 64 outputs go through a linear layer to define $\boldsymbol{\mu}_\lambda(\mathbf{x}_i)$ and $\log \boldsymbol{\Sigma}_\lambda(\mathbf{x}_i)$.
  - Finally, $q_\lambda(\mathbf{z}_i \,|\, \mathbf{x}_i, \boldsymbol{\lambda}) = \mathcal{N}(\boldsymbol{\mu}_\lambda(\mathbf{x}_i), \boldsymbol{\Sigma}_\lambda(\mathbf{x}_i))$.
- The **decoder network** $\mathbf{Z} \rightsquigarrow \mathbf{X}$ is a $64 + 256$ neural net with ReLU units.
  - The 256 outputs go through a linear layer to define logit$(\mathbf{p}_\theta(\mathbf{z}_i))$.
  - Then $p_\theta(\mathbf{x}_i \,|\, \mathbf{z}_i, \boldsymbol{\theta})$ is Bernoulli with parameters $\mathbf{p}_\theta(\mathbf{z}_i)$.

$$\mathbf{z}_i : 2 \text{ dim} \xrightarrow{\text{ReLU}} \text{Hidden, 64-d} \xrightarrow{\text{ReLU}} \text{Hidden, 256-d} \xrightarrow{\text{Linear}} \text{logit}(\mathbf{p}_i)\text{, 784-d} \longrightarrow p_\theta(\mathbf{x}_i \,|\, \mathbf{z}_i) = \text{Bernoulli}\,(\mathbf{p}_i)\text{, 784-d}$$

Learning progress; learning rate $\mu = 10^{-4}$



Trying to reconstruct $\mathbf{x}_i$ by $\mathbb{E}_{p_\theta}\left[\mathbf{X} \mid \mathbf{Z} = \mathbb{E}_{q_\lambda}\left[\mathbf{Z} \mid \mathbf{x}_i\right]\right]$

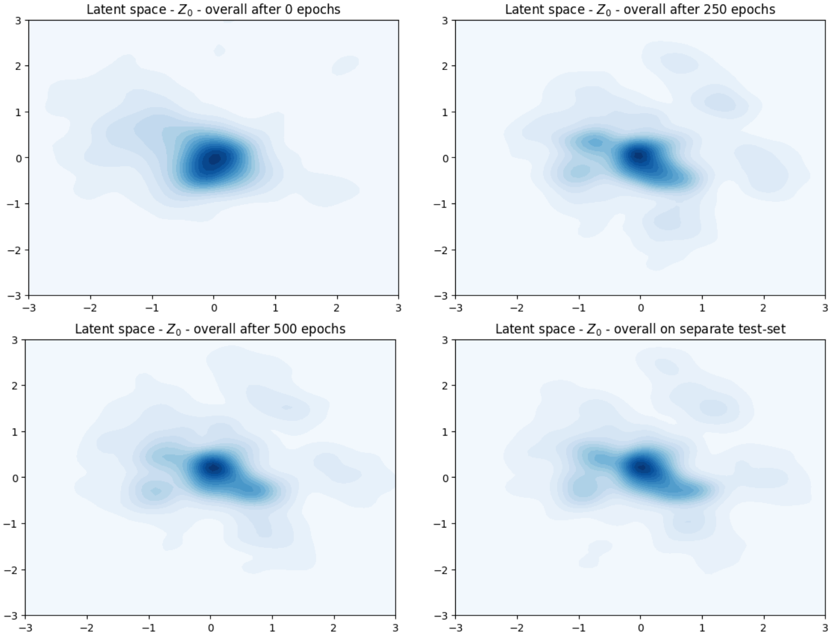
After 1 epoch

After 250 epochs

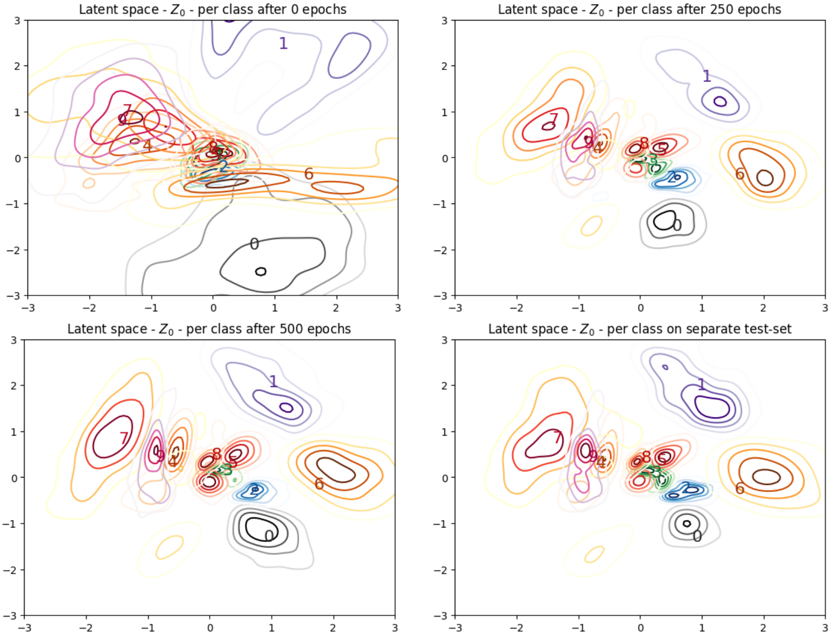
After 500 epoch

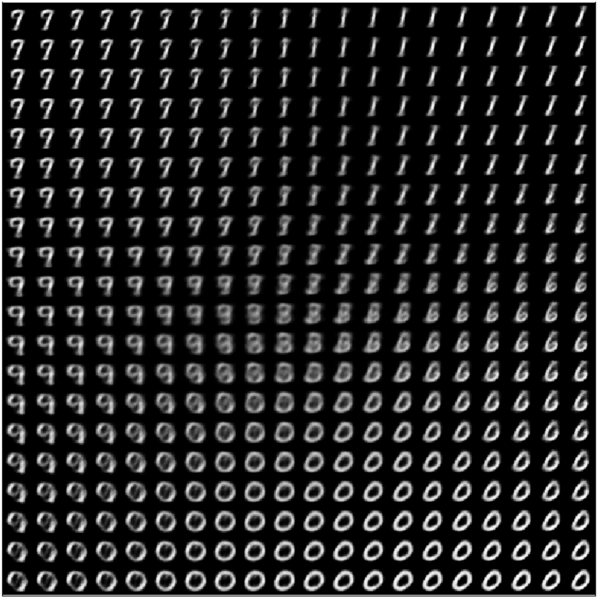
Using separate test-set

Averaged distribution over $\mathbf{Z}$

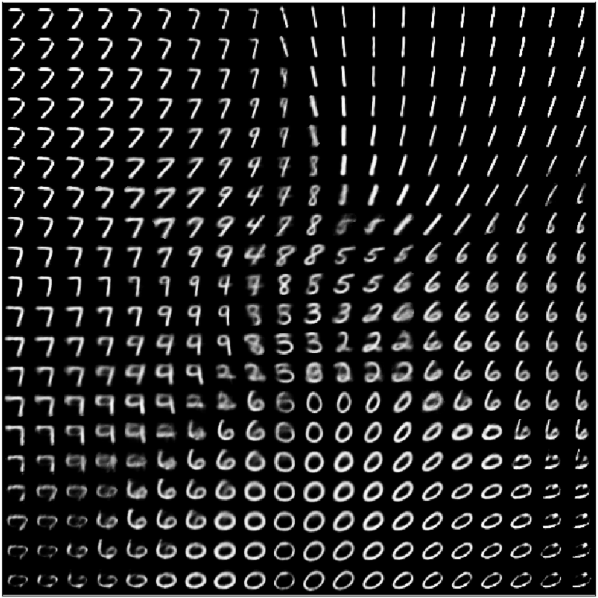


Averaged distribution over $\mathbf{Z}$ – per class

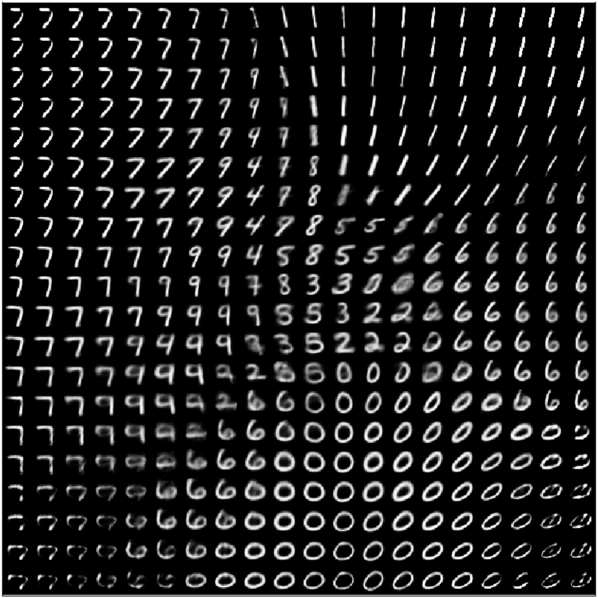




Manifold after 1 epoch



Manifold after 250 epochs



Manifold after 500 epochs

Black Box Variational Inference

- Variational inference will efficiently (both computer time and programming time) do inference in some models:
  - Exponential Family distributions, through **variational message passing**
  - Models that can be formulated as a **variational auto encoder** or other tailor-made structures
- However, we have not seen a general purpose inference technique that works for **all structures** and **all conditional distributions**.
- **Black Box Variational Inference** (BBVI) promises to be just that. . .

Main idea

The key idea is as for VAEs:

1. Cast variational inference as an optimization problem: Maximize ELBO.
2. Then use iterative refinement (stochastic gradient ascent) to optimize the variational distributions.

However, here we do operations directly on a model fully specified by statistical distributions, and do not need a DNN as a “catch-all” representation.

Key requirements

We want the approach to be . . .

"Black Box": Not requiring tailor-made adaptations by the modeller.

Applicable: Useful independently of the underlying model assumptions.

Efficient: Utilize modelling assumptions, including the mean field assumption, to improve computational speed.

Algorithm: Maximize $\mathcal{L}(q) = \mathbb{E}_{q_\lambda}\left[\log \frac{p_\theta(\mathbf{z},\mathbf{x})}{q_\lambda(\mathbf{z})}\right]$ by gradient ascent

- Initialization:
  - $t \leftarrow 0$;
  - $\hat{\boldsymbol{\lambda}}_0 \leftarrow$ random initialization;
  - $\boldsymbol{\rho} \leftarrow$ a Robbins-Monro sequence.
- Repeat until negligible improvement in terms of $\mathcal{L}(q)$:
  - $t \leftarrow t + 1$;
  - $\hat{\boldsymbol{\lambda}}_t \leftarrow \hat{\boldsymbol{\lambda}}_{t-1} + \rho_t \left. \nabla_\lambda \mathcal{L}(q) \right|_{\hat{\boldsymbol{\lambda}}_{t-1}}$;

BBVI - calculating the gradient

The algorithm requires that we can find

$$\nabla_\lambda \mathcal{L}(q) = \nabla_\lambda \mathbb{E}_q \left[\log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})} \right].$$

We can use these properties to simplify the equation:

1. $\nabla_\lambda \left(f(\mathbf{z}, \boldsymbol{\lambda}) \cdot g(\mathbf{z}, \boldsymbol{\lambda}) \right) = f(\mathbf{z}, \boldsymbol{\lambda}) \cdot \nabla_\lambda g(\mathbf{z}, \boldsymbol{\lambda}) + g(\mathbf{z}, \boldsymbol{\lambda}) \nabla_\lambda f(\mathbf{z}, \boldsymbol{\lambda})$
2. $\nabla_\lambda f(\mathbf{z}, \boldsymbol{\lambda}) = f(\mathbf{z}, \boldsymbol{\lambda}) \nabla_\lambda \log f(\mathbf{z}, \boldsymbol{\lambda})$
3. $\mathbb{E}_{q_\lambda} \left[\nabla_\lambda \log q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda}) \right] = 0$ for a density function $q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})$

Now it follows that

$$\nabla_\lambda \mathcal{L}(q) = \mathbb{E}_{q_\lambda} \left[\log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})} \cdot \nabla_\lambda \log q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda}) \right].$$

Calculating the gradient – Things to notice

$$\nabla_\lambda \mathcal{L}(q) = \mathbb{E}_{q_\lambda}\left[\log \frac{p_\theta(\mathbf{z},\mathbf{x})}{q_\lambda(\mathbf{z}\,|\,\boldsymbol{\lambda})} \cdot \nabla_\lambda \log q_\lambda(\mathbf{z}\,|\,\boldsymbol{\lambda})\right].$$

- We only need access to the un-normalized $p_\theta(\mathbf{z}, \mathbf{x})$ – not $p_\theta(\mathbf{z} \mid \mathbf{x})$.

$$\nabla_\lambda \mathcal{L}(q) = \mathbb{E}_{q_\lambda}\left[\log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})} \cdot \nabla_\lambda \log q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})\right].$$

Calculating the gradient – Things to notice

- We only need access to the un-normalized $p_\theta(\mathbf{z}, \mathbf{x})$ – not $p_\theta(\mathbf{z} \mid \mathbf{x})$.

$$\nabla_\lambda \mathcal{L}(q) = \mathbb{E}_{q_\lambda}\left[\log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})} \cdot \nabla_\lambda \log q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})\right].$$

- $q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})$ factorizes under MF, s.t. we can optimize per variable: $q_{\lambda_i}(z_i \mid \boldsymbol{\lambda}_i)$.

Calculating the gradient – Things to notice

- We only need access to the un-normalized $p_\theta(\mathbf{z}, \mathbf{x})$ – not $p_\theta(\mathbf{z} \,|\, \mathbf{x})$.

$$\nabla_\lambda \mathcal{L}(q) = \mathbb{E}_{q_\lambda}\left[\log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\lambda(\mathbf{z} \,|\, \boldsymbol{\lambda})} \cdot \nabla_\lambda \log q_\lambda(\mathbf{z} \,|\, \boldsymbol{\lambda})\right].$$

- $q_\lambda(\mathbf{z} \,|\, \boldsymbol{\lambda})$ factorizes under MF, s.t. we can optimize per variable: $q_{\lambda_i}(z_i \,|\, \boldsymbol{\lambda}_i)$.
- We must calculate $\nabla_\lambda \log q(\mathbf{z} \,|\, \boldsymbol{\lambda})$, which is also known as the "score function". This depends on the distributional family of $q(\cdot)$; can be precomputed for standard distributions.

Calculating the gradient – Things to notice

- We only need access to the un-normalized $p_\theta(\mathbf{z}, \mathbf{x})$ – not $p_\theta(\mathbf{z} \mid \mathbf{x})$.

$$\nabla_\lambda \mathcal{L}(q) = \mathbb{E}_{q_\lambda}\left[\log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})} \cdot \nabla_\lambda \log q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})\right].$$

- $q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})$ factorizes under MF, s.t. we can optimize per variable: $q_{\lambda_i}(z_i \mid \boldsymbol{\lambda}_i)$.
- We must calculate $\nabla_\lambda \log q(\mathbf{z} \mid \boldsymbol{\lambda})$, which is also known as the "score function". This depends on the distributional family of $q(\cdot)$; can be precomputed for standard distributions.
- The expectation will be approximated using a sample $\{\mathbf{z}_1, \ldots, \mathbf{z}_M\}$ generated from $q(\mathbf{z} \mid \boldsymbol{\lambda})$. Hence we require that we can **sample from** $q_{\lambda_i}(\cdot)$.

Calculating the gradient – Things to notice

- We only need access to the un-normalized $p_\theta(\mathbf{z}, \mathbf{x})$ – not $p_\theta(\mathbf{z} \mid \mathbf{x})$.

$$\nabla_\lambda \mathcal{L}(q) = \mathbb{E}_{q_\lambda}\left[\log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})} \cdot \nabla_\lambda \log q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})\right].$$

- $q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})$ factorizes under MF, s.t. we can optimize per variable: $q_{\lambda_i}(z_i \mid \boldsymbol{\lambda}_i)$.
- We must calculate $\nabla_\lambda \log q(\mathbf{z} \mid \boldsymbol{\lambda})$, which is also known as the "score function". This depends on the distributional family of $q(\cdot)$; can be precomputed for standard distributions.
- The expectation will be approximated using a sample $\{\mathbf{z}_1, \ldots, \mathbf{z}_M\}$ generated from $q(\mathbf{z} \mid \boldsymbol{\lambda})$. Hence we require that we can **sample from** $q_{\lambda_i}(\cdot)$.

Calculating the gradient – in summary

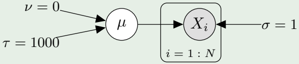
We have observed the datapoint $\mathbf{x}$, and our current estimate for $\boldsymbol{\lambda}_i$ is $\hat{\boldsymbol{\lambda}}_i$. Then

$$\nabla_{\lambda_i} \mathcal{L}(q)\big|_{\lambda=\hat{\lambda}_i} \approx \frac{1}{M}\sum_{j=1}^{M} \log \frac{p(z_{i,j}, \mathbf{x})}{q(z_{i,j} \mid \hat{\lambda}_i)} \cdot \nabla_{\lambda_i} \log q_i(z_{i,j} \mid \hat{\lambda}_i).$$

where $\{z_{i,1}, \ldots z_{i,M}\}$ are samples from $q_{\lambda_i}(\cdot \mid \hat{\lambda}_i)$.

Example model

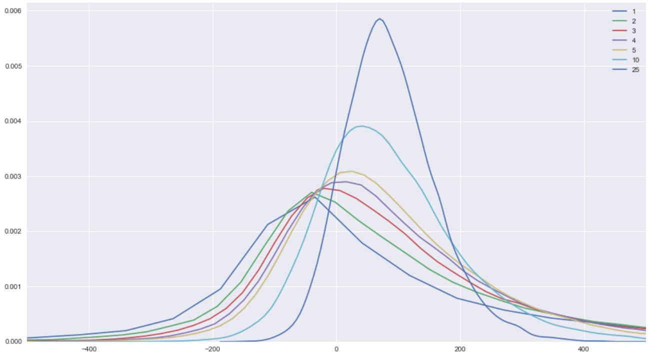
We assume a simple generative model:



Now, data is generated using $\mu = 10$, we assume a variational model with $q(\mu \,|\, \lambda) = \mathcal{N}(\lambda, 1)$ and want to maximize the ELBO wrt. the posterior mean for μ in the variational formulation.

BBVI.ipynb

PDF for the gradient calculated at $\lambda = 9$, which is below the optimum ≈ 10.
Several values for M, the sample size used to generate the estimate, are shown.



- Since the gradient estimate is based on a random sample, it is meaningful to evaluate the estimators' "robustness" in terms of a density function.
- We would hope to see robust estimates, also for small M, and in particular high probability for moving in the correct direction (gradient larger than 0).
- This is not the case, which has lead to a major focus on **variance reduction techniques**: while important we will **not cover them here**.

Probabilistic Programming Languages

Edward

Edward (`edwardlib.org`) is a Python library for probabilistic modeling, inference, and criticism, integrated with Tensorflow.

Modeling:
- Directed graphical models
- Neural networks (via libraries such as `tf.layers` and `tf.keras`)
- ...

Inference:
- Variational inference – including BBVI, SVI
- Monte Carlo – including Gibbs, Hamiltonian Monte Carlo
- Traditional Message passing algorithms
- ...

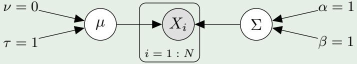
Criticism:
- Point-based evaluations
- Posterior predictive checks
- ...

... and there are also many other possibilities

`Tensorflow` is integrating probabilistic thinking into its core, Uber has recently released `Pyro`, `InferPy` is a local alternative, etc.

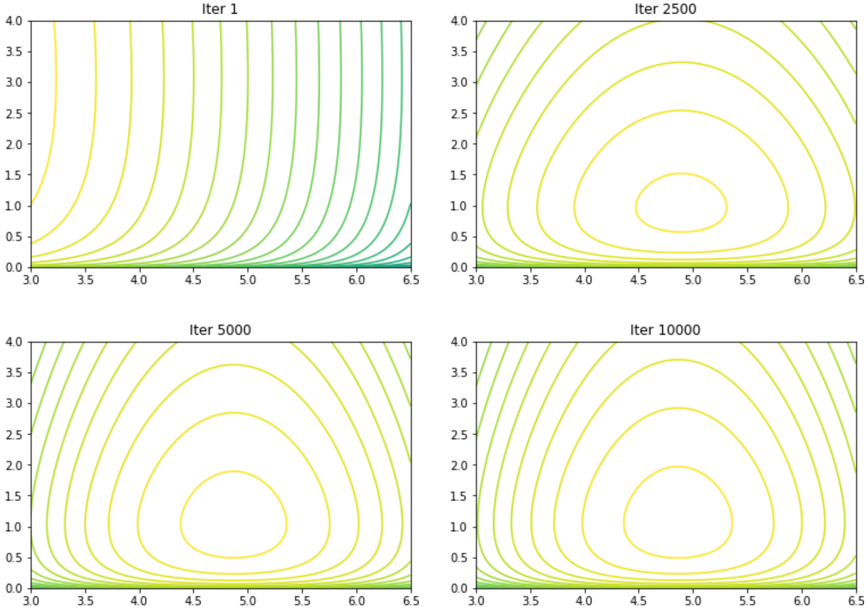
Example model

We assume a simple generative model:

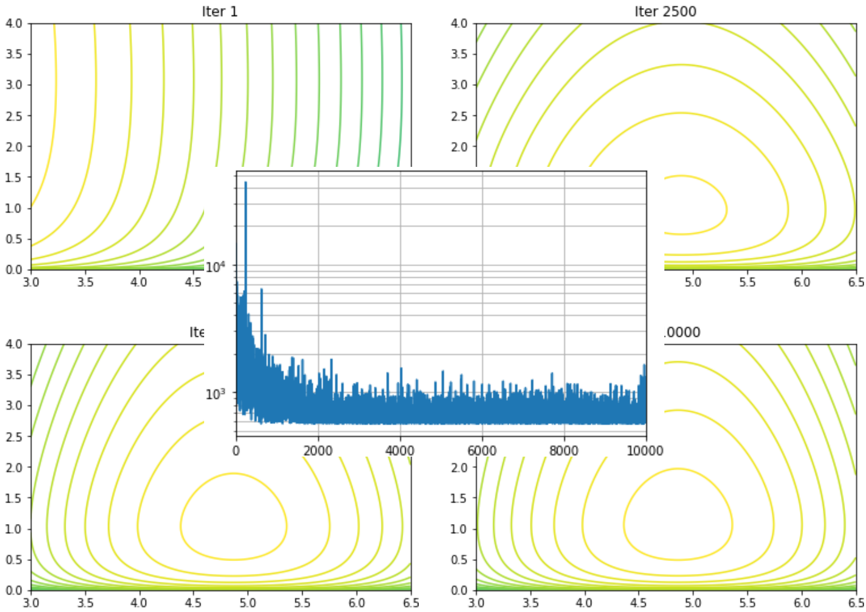


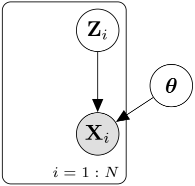
Edward-simple-Gaussian.ipynb

Posterior variational distribution over $(\mu, 1/\Sigma)$



Posterior variational distribution over $(\mu, 1/\Sigma)$

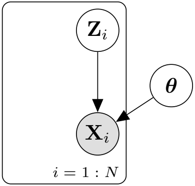




Generative model: $\mathbf{Z} \rightsquigarrow \mathbf{X}$

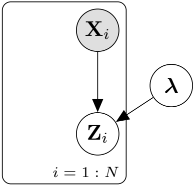
```
z = ed.models.Normal(
    loc=tf.zeros([batch_size, z_dim]),
    scale=tf.ones([batch_size, z_dim]))
hidden_gen = tf.layers.dense(z, 64,
    activation=tf.nn.relu)
hidden_gen = tf.layers.dense(
    hidden_gen, 256, activation=tf.nn.relu)
x = ed.models.Bernoulli(
    logits=tf.layers.dense(hidden_gen, 28 * 28))
```

Generative model: $\mathbf{Z} \rightsquigarrow \mathbf{X}$




```
z = ed.models.Normal(
    loc=tf.zeros([batch_size, z_dim]),
    scale=tf.ones([batch_size, z_dim]))
hidden_gen = tf.layers.dense(z, 64,
    activation=tf.nn.relu)
hidden_gen = tf.layers.dense(
    hidden_gen, 256, activation=tf.nn.relu)
x = ed.models.Bernoulli(
    logits=tf.layers.dense(hidden_gen, 28 * 28))
```

Variational model: $\mathbf{X} \rightsquigarrow \mathbf{Z}$




```
x_ph = tf.placeholder(
    tf.int32, [batch_size, 28 * 28])
hidden_vb = tf.layers.dense(
    tf.cast(x_ph, tf.float32), 256,
    activation=tf.nn.relu)
hidden_vb = tf.layers.dense(hidden_vb, 64,
    activation=tf.nn.relu)
qz = ed.models.Normal(
    loc=tf.layers.dense(hidden_vb, z_dim),
    scale=tf.layers.dense(hidden_vb, z_dim,
        activation=tf.nn.softplus))
```

- Inference is done by binding distributions together. Here `z` and `qz` are bound, and relate to the same data `x` – which is fed into the system using the placeholder `x_ph`.
- Notice how we can choose among different inference engines. Here we do `KLqp`, which is standard BBVI.
- Everything is built on top of Tensorflow, hence we have access to the standard optimization routines for training, here we use `RMSprop`

Code to define the optimization:

```
# Bind p(x, z) and q(z | x) to the same TensorFlow placeholder for x.
inference = ed.KLqp({z: qz}, data={x: x_ph})
optimizer = tf.train.RMSPropOptimizer(0.01, epsilon=1.0)
inference.initialize(optimizer=optimizer)
```

Code to do the actual training:

```
for epoch in range(n_epoch):
    print("Epoch: {:3d}: ".format(epoch), end='')
    loss = 0.0

    for t in range(n_iter_per_epoch):
        x_batch = next(x_train_generator)
        info_dict = inference.update(feed_dict={x_ph: x_batch})
        loss  += info_dict['loss']
```

Example: Variational Auto Encoder in `Edward` – cont'd

- Inference is done by binding distributions together. Here `z` and `qz` are bound, and relate to the same data `x` – which is fed into the system using the placeholder `x_ph`.
- Notice how we can choose among different inference engines. Here we do `KLqp`, which is standard BBVI.
- Everything is built on top of Tensorflow, hence we have access to the standard optimization routines for training, here we use `RMSprop`

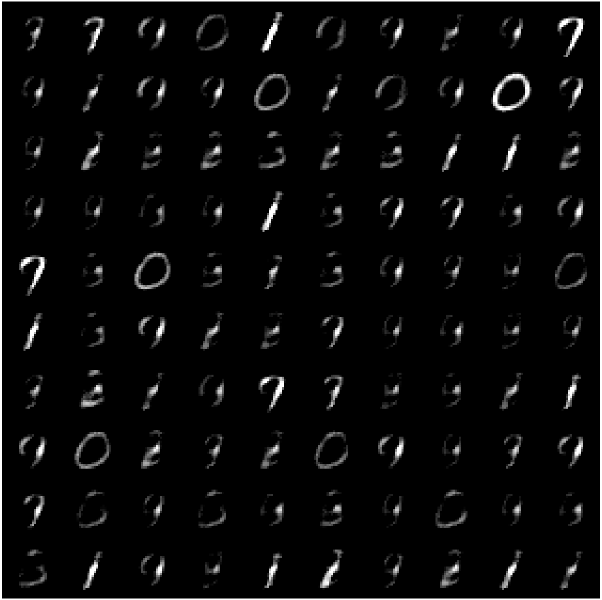
Example – Python notebook

Show notebook: **`Edward-VAE.ipynb`**

```
print("Epoch: {:3d}: ".format(epoch), end='')
loss = 0.0

for t in range(n_iter_per_epoch):
    x_batch = next(x_train_generator)
    info_dict = inference.update(feed_dict={x_ph: x_batch})
    loss  += info_dict['loss']
```

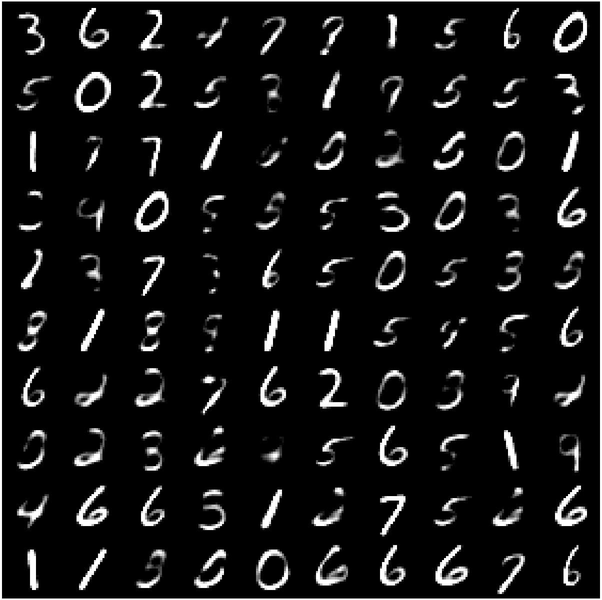
Generated pictures from Edward: $\mathbb{E}_{p_\theta}\left[\mathbf{X} \mid \mathbf{z}\right]$ for sampled $\mathbf{Z}$-values



Examples after 1 epoch

Examples after 250 epoch

Generated pictures from Edward: $\mathbb{E}_{p_\theta}[\mathbf{X} \mid \mathbf{z}]$ for sampled $\mathbf{Z}$-values



Manifold after 500 epochs

Conclusions

Variational Bayes: VB is a deterministic alternative to sampling for **approximate inference in Bayesian models**.

- VB seeks the model $q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda}_\mathbf{x})$ inside a family of applicable models $\mathcal{Q}$ that is closest to the (unattainable) posterior $p(\mathbf{z} \mid \mathbf{x})$ in terms of a Kullback Leibler divergence.
- Depending on $\mathcal{Q}$, we can assert efficient inference – a very common choice is the **mean field assumption**:

$$q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda}) = \prod_i q_{\lambda_i}(z_i \mid \boldsymbol{\lambda}_i).$$

Variational Bayes: VB is a deterministic alternative to sampling for **approximate inference in Bayesian models**.

VB message passing: Variational Bayes for **Exponential Family** models.

Variational Bayes: VB is a deterministic alternative to sampling for **approximate inference in Bayesian models**.

VB message passing: Variational Bayes for **Exponential Family** models.

Stochastic VB: **Mini-batching** in VMP. Leaning heavily on stochastic approximation theory.

Variational Bayes

Variational Bayes: VB is a deterministic alternative to sampling for **approximate inference in Bayesian models**.

VB message passing: Variational Bayes for **Exponential Family** models.

Stochastic VB: **Mini-batching** in VMP. Leaning heavily on stochastic approximation theory.

VB outside Exponential Family models: VB for general distribution families.

- The goal is to obtain **efficient** inference in **unconstrained** Bayesian network models – any structure, any combination of distributional families.
- **Variational Auto-Encoders** efficiently implements **bipartite latent-variable models** with flexible conditional probability distributions.
- **Black-Box** Variational Inference promises VB inference in any model, but at the cost that inference relies on sampling – and it sometimes shows poor converge properties in practice.

Variational Bayes: VB is a deterministic alternative to sampling for **approximate inference in Bayesian models**.

VB message passing: Variational Bayes for **Exponential Family** models.

Stochastic VB: **Mini-batching** in VMP. Leaning heavily on stochastic approximation theory.

VB outside Exponential Family models: VB for general distribution families.

Probabilistic Programming Languages: PPLs are programming languages to describe probabilistic models and perform inference in them.

- `Edward` is a PPL built on top of `Tensorflow`, and which supports several inference techniques, including BBVI, SVI, MCMC, and exact inference.
- Several other alternatives exist as well (`Pyro`, `Stan`, `JAGS`, `InferPy` ...)