# Variational inference

## Partly based on material developed together with Helge Langseth

Andrés Masegosa and Thomas Dyhre Nielsen

June 2019

# Plan for this week

**Day 1:** Probabilistic programming

- Introduction to probabilistic programming
- Probabilistic programming in Pyro

**Day 2:** Variational inference

- Recap of variational inference (variational inference as optimization)
- Derivation and implementation of selected examples
  - Bayesian linear regression
  - Factor analysis
  - ...

**Day 3:** Variational inference – cont'd

- Black box variational inference
- Variational inference in Pyro
- Variational auto-encoders

## Black Box Variational Inference

## VI inference as optimization

We can minimize (improve the variational approximation)

$$\mathrm{KL}(q_\lambda(z), p(z \mid \mathbf{x}))$$

by maximizing the ELBO

$$\mathcal{L}(q) = \mathbb{E}_q \left[ \log \frac{p(\mathbf{z}, \mathbf{x})}{q(\mathbf{z})} \right]$$

## VI inference as optimization

We can minimize (improve the variational approximation)

$$\mathrm{KL}(q_\lambda(z), p(z \mid \mathbf{x}))$$

by maximizing the ELBO

$$\mathcal{L}(q) = \mathbb{E}_q \left[ \log \frac{p(\mathbf{z}, \mathbf{x})}{q(\mathbf{z})} \right]$$

## The mean field assumption

We will often use the mean field assumption, which states that $\mathcal{Q}$ consists of all distributions that *factorizes* according to the equation

$$q(\mathbf{z}) = \prod_i q_i(z_i)$$

$\rightsquigarrow$ we can treat the variables independently.

### 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;
- Repeat until negligible improvement in terms of $\mathcal{L}(q)$:
  - $t \leftarrow t + 1$;
  - $\hat{\boldsymbol{\lambda}}_t \leftarrow \hat{\boldsymbol{\lambda}}_{t-1} + \rho\, \nabla_\lambda \mathcal{L}(q)\big|_{\hat{\boldsymbol{\lambda}}_{t-1}}$;

# BBVI - calculating the gradient

The algorithm requires that we can find

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

With a bit of pencil pushing it follows that

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

# BBVI - calculating the gradient

The algorithm requires that we can find

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

With a bit of pencil pushing it follows that

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

## Properties used for derivation

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 $q_\lambda(\mathbf{z} \mid \boldsymbol{\lambda})$

# Calculating the gradient – Things to notice

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

- 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 $p_\theta(\mathbf{z} \mid \mathbf{x})$.

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

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

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

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

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

## Example

If $q_{\boldsymbol{\lambda}}(z)$ follows a normal distribution ($\boldsymbol{\lambda} = (\mu, \sigma)$):

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right),$$

then

$$\nabla_\mu \log q_{\boldsymbol{\lambda}}(z) = \frac{1}{\sigma^2}(z - \mu)$$

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

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

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

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

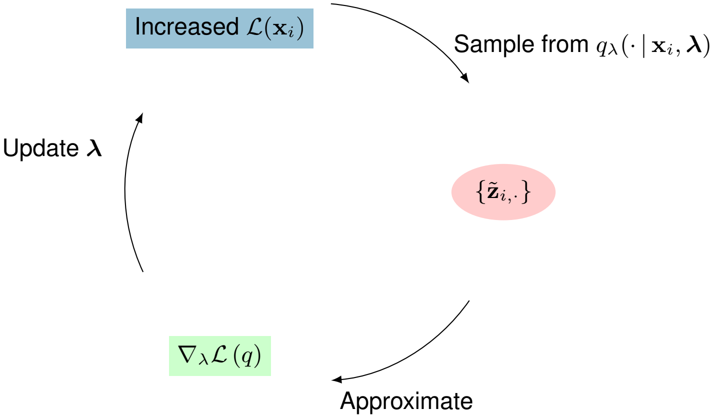
- $q_\lambda(\mathbf{z})$ factorizes under MF, s.t. we can optimize per variable: $q_{\lambda_i}(z_i)$.
- We must calculate $\nabla_\lambda \log q(\mathbf{z} \mid \boldsymbol{\lambda})$, which is also known as the "score function". This depends on the distributional family of $q(\cdot)$; can be precomputed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

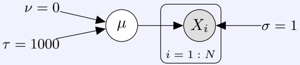
$$\nabla_{\lambda_i} \mathcal{L}(q)|_{\lambda=\hat{\lambda}_i} \approx \frac{1}{M} \sum_{j=1}^{M} \log \frac{p(z_{i,j}, \mathbf{x})}{q(z_{i,j} \mid \hat{\lambda}_i)} \cdot \nabla_{\lambda_i} \log q_i(z_{i,j} \mid \hat{\lambda}_i).$$

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


Increased $\mathcal{L}(\mathbf{x}_i)$
Sample from $q_\lambda(\cdot \mid \mathbf{x}_i, \boldsymbol{\lambda})$
$\{\tilde{\mathbf{z}}_{i,\cdot}\}$
Approximate
$\nabla_\lambda \mathcal{L}(q)$
Update $\boldsymbol{\lambda}$

## Exercise: BBVI in Python

Consider the simple generative model:

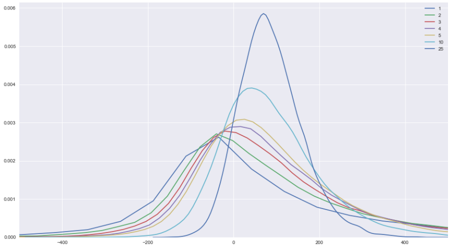



- Derive the BBVI estimate of the gradient for the variational parameters of $q(\mu) = \mathcal{N}(\lambda, 1)$.
- Implement the gradient estimate in the notebook

  `students_BBVI.ipynb`

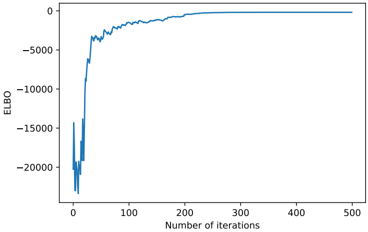
- Perform gradient ascent using your gradient implementation by running the notebook.

## Density of gradient estimates




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

## Evolution of ELBO




Based on gradient estimates using $1$ sample

BBVI-full.ipynb

- 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: Variational inference in Pyro

### `Pyro`

Pyro (`pyro.ai`) is a Python library for probabilistic modeling, inference, and criticism, integrated with PyTorch.

**Modeling:**
- Directed graphical models
- Neural networks (via `nn.Module`)
- ...

**Inference:**
- Variational inference – including BBVI, SVI
- Monte Carlo – including Importance sampling and Hamiltonian Monte Carlo
- ...

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

### ... and there are also many other possibilities

`Tensorflow` is integrating probabilistic thinking into its core, `InferPy` is a local alternative, etc.

## Pyro models in general

- observations ⇔ `pyro.sample` with the `obs` argument
- latent random variables ⇔ `pyro.sample`
- parameters ⇔ `pyro.param`

## Simple example

```python
#The observatons
obs = {'sensor': torch.tensor(18.0)}

def model(obs):
    temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
    sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
```

## Guides

**Definition:**

- Guides are **arbitrary stochastic functions**.
- Guides produces samples for those variables of the model which are **not observed**.

## Guides

### Definition:

- Guides are **arbitrary stochastic functions**.
- Guides produces samples for those variables of the model which are **not observed**.

### Guides are used for:

- Define the $q$ **distributions** in variational settings.
- Define **inference networks** as in VAEs.
- Build **proposal distributions** in importance sampling, MCMC.
- ...

## Guide requirements

Guide functions must satisfy these two criteria to be valid approximations for a particular model:

1. all unobserved (i.e., not conditioned) sample statements that appear in the model appear in the guide.
2. the guide has the same input signature as the model (i.e., takes the same arguments)

## Example

```python
#The observatons
obs = {'sensor': torch.tensor(18.0)}

def model(obs):
    temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
    sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
```

```python
#The guide
def guide(obs):
    a = pyro.param("mean", torch.tensor(0.0))
    b = pyro.param("scale", torch.tensor(1.), constraint=constraints.positive)
    temp = pyro.sample('temp', dist.Normal(a, b))
```

# Pyro example

**Bayesian_linear_regression.ipynb**

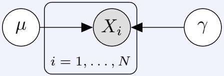
**FA.ipynb**

### Exercise 1: Explore existing models

Go through and explore the notebooks

- `Bayesian_linear_regression.ipynb`
- `FA.ipynb`

### Exercise 2: Pyro implementation for a simple Gaussian model




- $X_i \mid \{\mu, \gamma\} \sim \mathcal{N}(\mu, 1/\gamma)$
- $\mu \sim \mathcal{N}(0, \tau)$
- $\gamma \sim \text{Gamma}(\alpha, \beta)$

In this task you should implement a pyro model and guide for the graphical model above. This involves specifying appropriate parameters for the model (e.g. reflecting prior knowledge) as well as coming up with a suitable variational approximation in the form of the Pyro guide. Make your implementation in the notebook

`Day3/student_simple_model.ipynb`

which also contains a data generation component as well as the framework for the learning procedure.

# 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

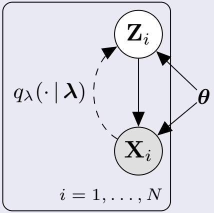
- Recall that a Bayesian network specification includes the conditional probability distribution $p(x_i \mid \mathrm{pa}(x_i))$ for each variable $X_i$.
- Typically the CPD is assumed to belong to some distributional family out of convenience — e.g., to obtain conjugacy.
- Deep Bayesian models opens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

# The Variational Auto Encoder (VAE)

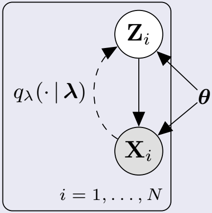
## Model of interest




- 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.

## 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 $\boldsymbol{\lambda}$.

**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

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

**The two terms penalizes:**

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

# Calculating the ELBO terms

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

- The KL-term is dependent on the distributional families of $p_\theta(\mathbf{z})$ and $q_\lambda(\mathbf{z} \mid \mathbf{x}_i, \boldsymbol{\lambda})$.
  - One can assume a simple shape, like:
    - $p_\theta(\mathbf{z})$ being Gaussian with zero mean and isotropic covariance;
    - $q_\lambda(z_\ell \mid \mathbf{x}_i, \boldsymbol{\lambda})$ 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

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

- 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).
- 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

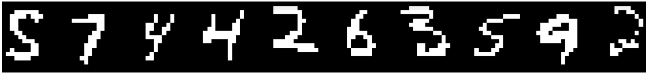
$$\nabla_{\lambda,\theta}\mathcal{L}\left(\mathcal{D}, \boldsymbol{\theta}, \boldsymbol{\lambda}\right) \approx \nabla_{\lambda,\theta}\sum_{i=1}^{N}\tilde{\mathcal{L}}(\mathbf{x}_i).$$

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

## Simple implementation

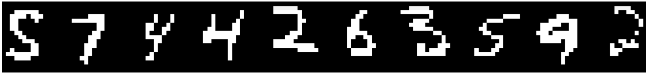
Notice that variational learning is casted as a gradient ascent procedure. We can therefore utilize Pyro and Tensorflow or other similar tools.

- 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 $28 \times 28$ array, each $\mathbf{x}_i$ is a picture of a handwritten digit (“0” – “9”)

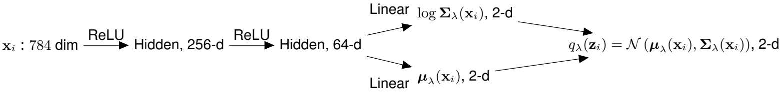


# Fun with MNIST – The model

- 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 $28 \times 28$ array, each $\mathbf{x}_i$ is a picture of a handwritten digit ("0" – "9")

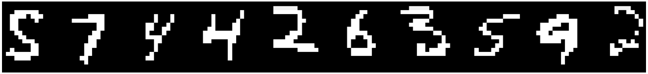


- Encoding is done in **two** dimensions. A priori $\mathbf{Z}_i \sim p_\theta(\mathbf{z}_i) = \mathcal{N}(\mathbf{0}_2, \mathbf{I}_2)$.
- The approximate expectation in the ELBO is calculated using $M = 1$ sample per data-point.
- The **encoder network** $\mathbf{X} \rightsquigarrow \mathbf{Z}$ is a $256 + 64$ 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, $q_\lambda(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\lambda}) = \mathcal{N}(\boldsymbol{\mu}_\lambda(\mathbf{x}_i), \boldsymbol{\Sigma}_\lambda(\mathbf{x}_i))$.



# 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 $28 \times 28$ 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, $q_\lambda(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\lambda}) = \mathcal{N}(\boldsymbol{\mu}_\lambda(\mathbf{x}_i), \boldsymbol{\Sigma}_\lambda(\mathbf{x}_i))$.
- 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 $p_\theta(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta})$ is Bernoulli with parameters $\mathbf{p}_\theta(\mathbf{z}_i)$.

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

# Pyro specification of an encoder

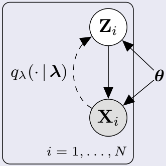
```python
class Decoder(nn.Module):
    def __init__(self, z_dim, hidden_dim):
        super(Decoder, self).__init__()
        # Setup the two linear transformations used
        self.fc1 = nn.Linear(z_dim, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, 784)
        # Setup the non-linearities
        self.softplus = nn.Softplus()
        self.sigmoid = nn.Sigmoid()

    def forward(self, z):
        # Define the forward computation on the latent z
        # First compute the hidden units
        hidden = self.softplus(self.fc1(z))
        # Return the parameter for the output Bernoulli
        # Each is of size batch_size x 784
        loc_img = self.sigmoid(self.fc21(hidden))
        return loc_img

# define the model p(x|z)p(z)
def model(self, x):
    # register PyTorch module `decoder` with Pyro
    pyro.module("decoder", self.decoder)
    with pyro.plate("data", x.shape[0]):
        # setup hyperparameters for prior p(z)
        z_loc = x.new_zeros(torch.Size((x.shape[0], self.z_dim)))
        z_scale = x.new_ones(torch.Size((x.shape[0], self.z_dim)))
        z = pyro.sample("latent", dist.Normal(z_loc, z_scale).to_event(1))
        # decode the latent code z
        loc_img = self.decoder.forward(z)
        # score against actual images
        pyro.sample("obs", dist.Bernoulli(loc_img).to_event(1),
                    obs=x.reshape(-1, 784))
```

## Notes

- The PYRO.MODULE call registers the parameters in the decoder network with Pyro.
- The decoder network is a subclass of NN.MODULE; the class inherits methods such as PARAMETERS() and BACKWARD for calculating gradients.



# Pyro specification of a decoder

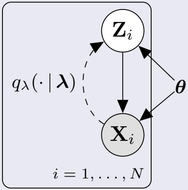
```python
class Encoder(nn.Module):
    def __init__(self, z_dim, hidden_dim):
        super(Encoder, self).__init__()
        # Setup the three linear transformations used
        self.fc1 = nn.Linear(784, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, z_dim)
        self.fc22 = nn.Linear(hidden_dim, z_dim)
        # Setup the non-linearities
        self.softplus = nn.Softplus()

    def forward(self, x):
        # Define the forward computation on the image x
        # First shape the mini-batch to have pixels in
        # the rightmost dimension
        x = x.reshape(-1, 784)
        # then compute the hidden units
        hidden = self.softplus(self.fc1(x))
        # Return a mean vector and a (positive) square
        # root covariance each of size batch_size x z_dim
        z_loc = self.fc21(hidden)
        z_scale = torch.exp(self.fc22(hidden))
        return z_loc, z_scale

# define the guide (i.e. variational distribution) q(z|x)
def guide(self, x):
    # register PyTorch module `encoder` with Pyro
    pyro.module("encoder", self.encoder)
    with pyro.plate("data", x.shape[0]):
        # use the encoder to get the parameters used to define q(z|x)
        z_loc, z_scale = self.encoder.forward(x)
        # sample the latent code z
        pyro.sample("latent", dist.Normal(z_loc, z_scale).to_event(1))
```

## Notes

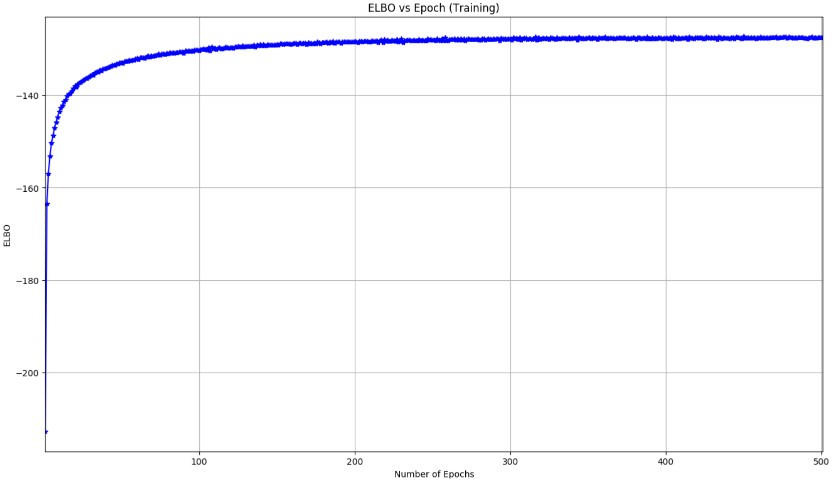
- The encoder and guide follow the same structure as the encoder and model



# Wrapping things up

**VAE.ipnyb**

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



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

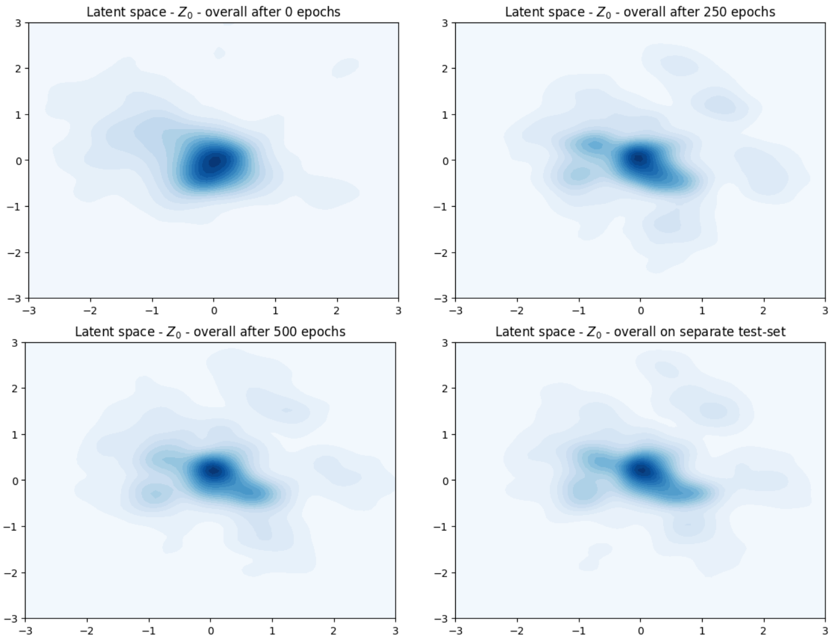
After 1 epoch

After 250 epochs

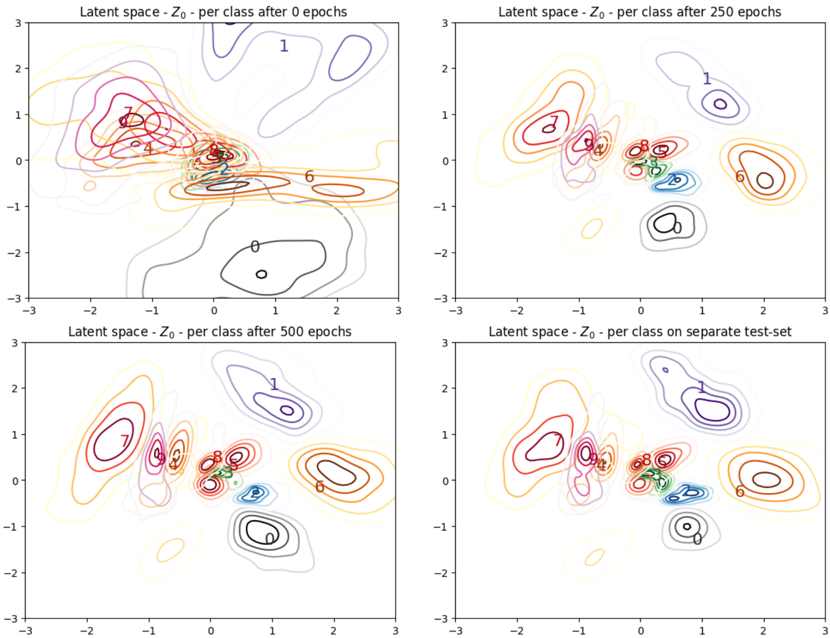
After 500 epoch

Using separate test-set

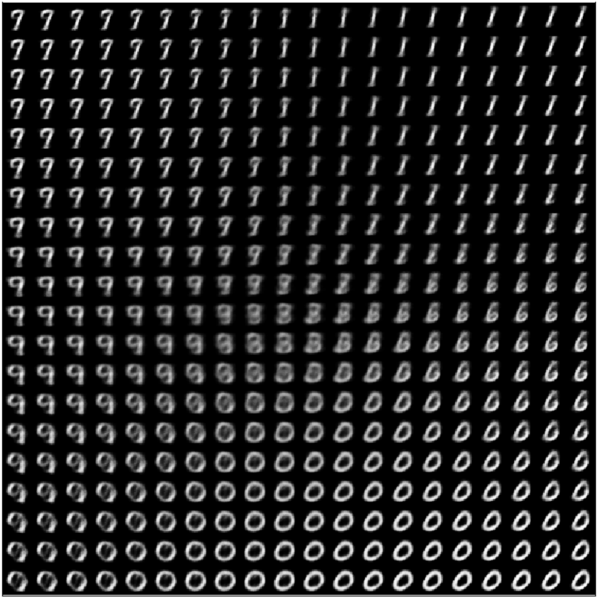
# Averaged distribution over $\mathbf{Z}$



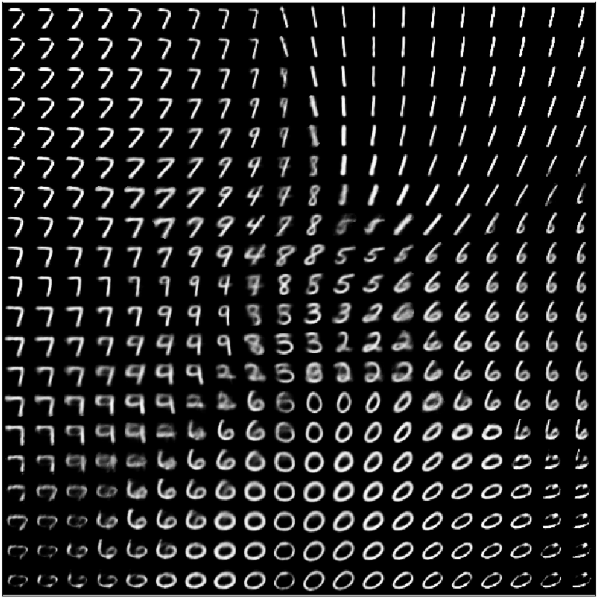
# Averaged distribution over $\mathbf{Z}$ – per class



# The picture manifold – $\mathbb{E}_{p_\theta}[\mathbf{X} \mid \mathbf{z}]$ for different values of $\mathbf{z}$

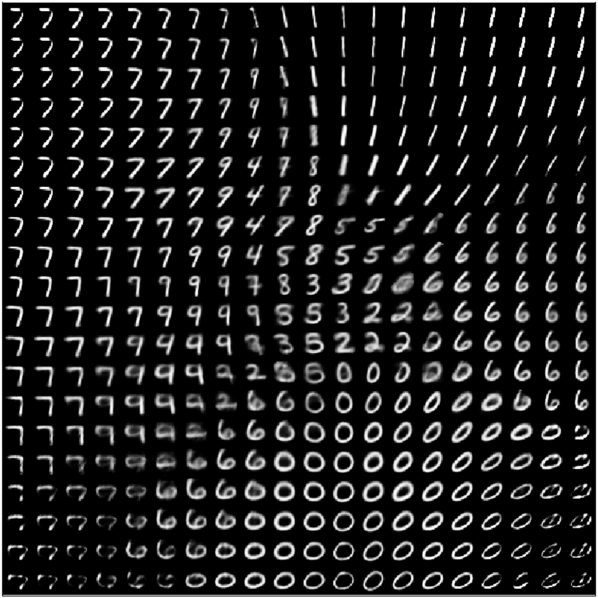


Manifold after 1 epoch



Manifold after 250 epochs

# The picture manifold – $\mathbb{E}_{p_\theta}[\mathbf{X} \mid \mathbf{z}]$ for different values of $\mathbf{z}$



Manifold after 500 epochs

# Conclusions

- **PPLs are the right tool for probabilistic modeling.**
  - Enormous expressibility.
  - Powerful inference engines (BlackBox Variational Inference)

- **PPLs are the right tool for probabilistic modeling.**
  - Enormous expressibility.
  - Powerful inference engines (BlackBox Variational Inference)

- **Conjugate Exponential Models.**
  - Variational Inference is very efficient and stable.
  - Requires manual derivation of updating equations.
  - There are tools (variational message passing) that avoid that (Infer.net, Amidst Toolbox, etc).

# Conclusions

- **PPLs are the right tool for probabilistic modeling.**
  - Enormous expressibility.
  - Powerful inference engines (BlackBox Variational Inference)

- **Conjugate Exponential Models.**
  - Variational Inference is very efficient and stable.
  - Requires manual derivation of updating equations.
  - There are tools (variational message passing) that avoid that (Infer.net, Amidst Toolbox, etc).

- **Beyond Conjugate Exponential Models.**
  - Combine deep learning and probabilistic modeling.
  - Black-Box VI is not so efficient and stable.
  - But it works well in many cases.