

Probabilistic Programming

Marius Popescu

popescunmarius@gmail.com

2019 - 2020

Variational Bayesian Methods

From Variational Inference to Variational Autoencoders

Variational Bayesian Methods

- Variational Bayesian methods are a family of techniques for approximating *intractable integrals* arising in Bayesian inference and machine learning. They are typically used in complex statistical models consisting of observed variables (usually termed "data") as well as unknown parameters and latent variables, with various sorts of relationships among the three types of random variables.
- Variational Bayesian methods are primarily used for two purposes:
 - To provide an analytical approximation to the posterior probability of the unobserved variables, in order to do statistical inference over these variables.
 - To derive a lower bound for the marginal likelihood (sometimes called the "evidence") of the observed data (i.e. the marginal probability of the data given the model, with marginalization performed over unobserved variables). This is typically used for performing model selection, the general idea being that a higher marginal likelihood for a given model indicates a better fit of the data by that model and hence a greater probability that the model in question was the one that generated the data.

Variational Inference

Approximate inference

Probabilistic model: $p(x, \theta) = p(x|\theta)p(\theta)$

Variational Inference

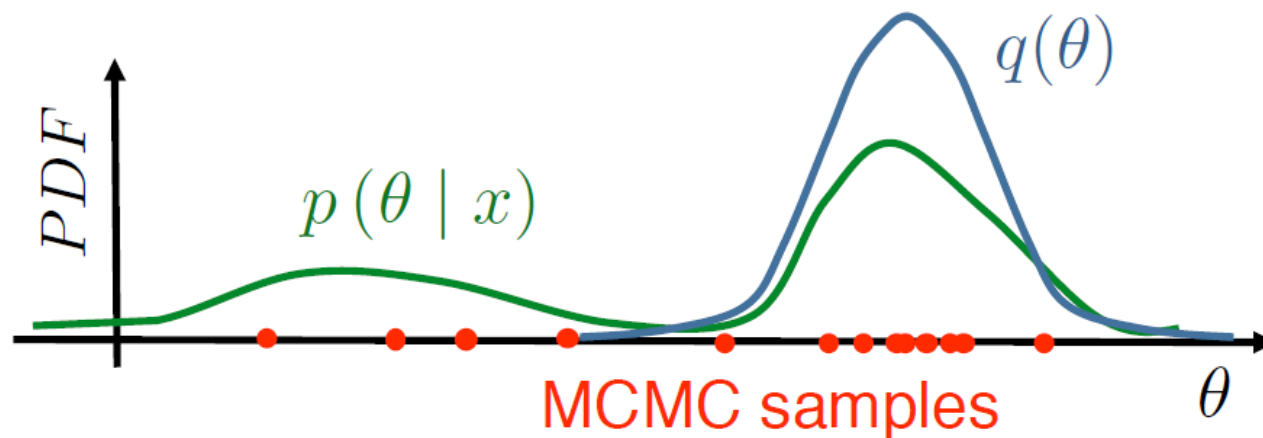
Approximate $p(\theta|x) \approx q(\theta) \in \mathcal{Q}$

- Biased
- Faster and more scalable

MCMC

Samples from unnormalized $p(\theta|x)$

- Unbiased
- Need a lot of samples



Variational Inference

In variational inference, the posterior distribution over a set of unobserved variables $\mathbf{Z} = \{Z_1, \dots, Z_n\}$ given some data \mathbf{X} is approximated by a variational distribution, $q(\mathbf{Z})$:

$$p(\mathbf{Z}|\mathbf{X}) \approx q(\mathbf{Z})$$

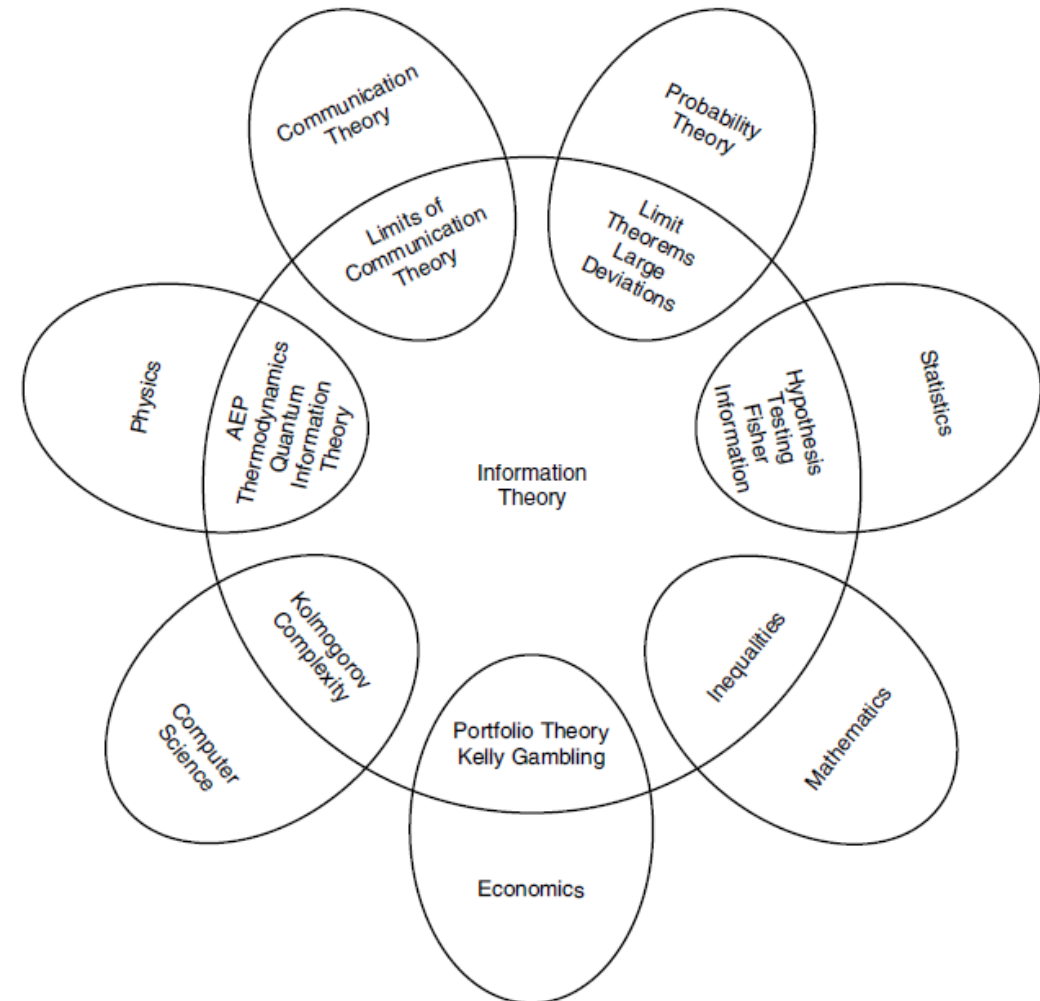
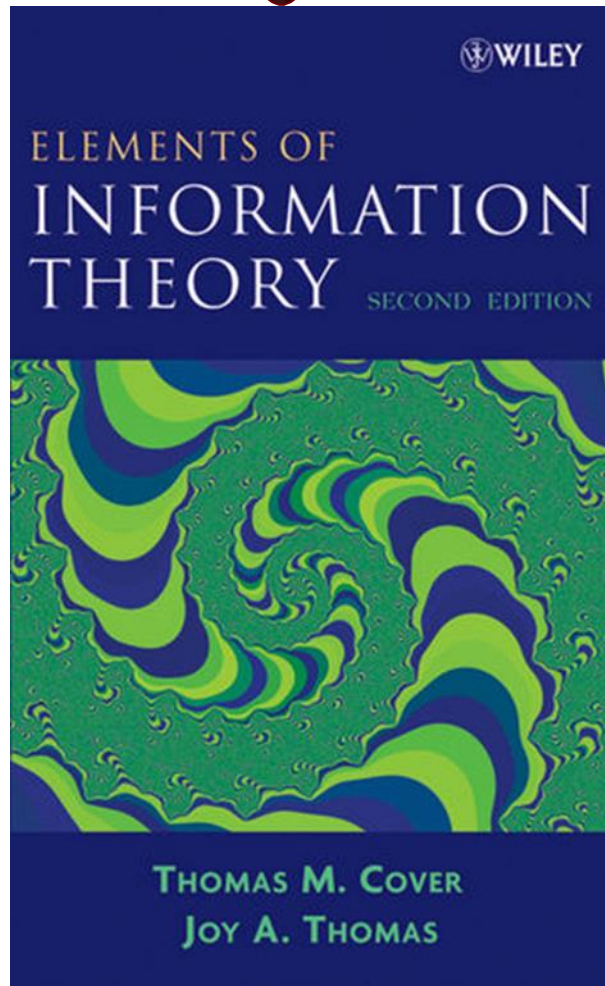
The distribution $q(\mathbf{Z})$ is restricted to belong to a family of distributions (with its own *variational parameters* λ) of simpler form than $p(\mathbf{Z}|\mathbf{X})$ (typically, the true posterior is not in the variational family).

The aim is to find the variational parameters λ that makes q close to the posterior of interest and use q with the fitted parameters as a proxy for the posterior, e.g., to form predictions about future data or to investigate the posterior distribution of the hidden variables.

The most common type of variational inference uses the Kullback–Leibler divergence (KL-divergence) of the two distributions as measure of the closeness.

Variational inference can be seen as an extension of the maximum a posteriori estimation (MAP) of the single most probable value of each parameter to fully Bayesian estimation which computes (an approximation to) the entire posterior distribution of the latent variables.

KL-divergence comes from Information Theory



Information Theory



In the early 1940s it was thought to be impossible to send information at a positive rate with negligible probability of error. Shannon surprised the communication theory community by proving that the probability of error could be made nearly zero for all communication rates below channel capacity. The capacity can be computed simply from the noise characteristics of the channel. Shannon further argued that random processes such as music and speech have an irreducible complexity below which the signal cannot be compressed. This he named *the entropy*, in deference to the parallel use of this word in thermodynamics, and argued that if the entropy of the source is less than the capacity of the channel, asymptotically error-free communication can be achieved.

Information theory answers two fundamental questions in communication theory: What is the ultimate data compression (answer: the entropy H), and what is the ultimate transmission rate of communication (answer: the channel capacity C).

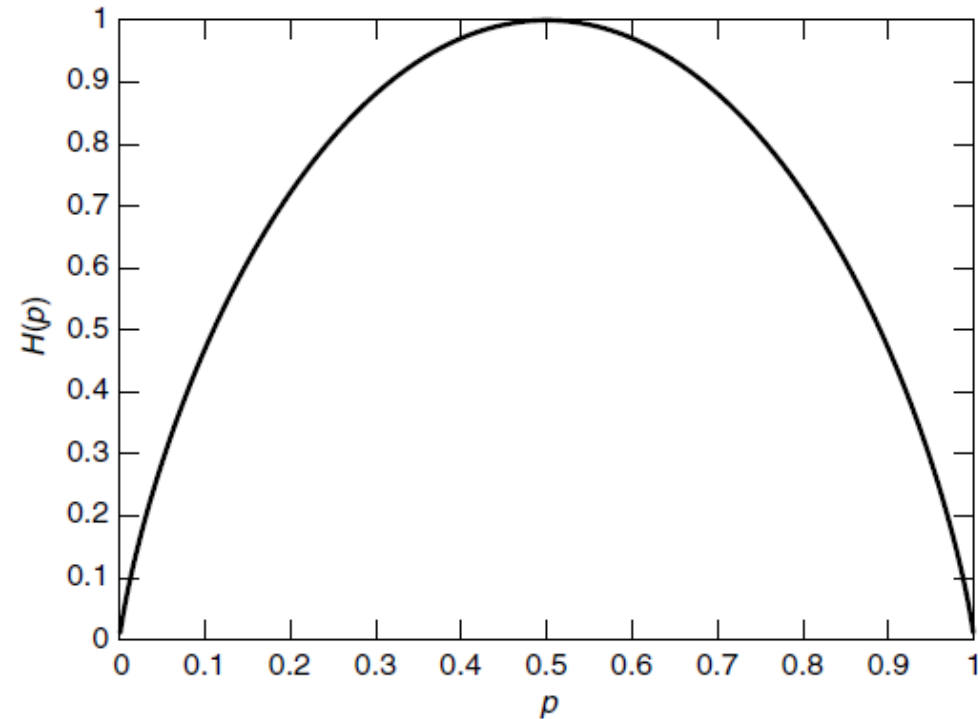
The Entropy

The entropy $H(X)$ of a discrete random variable X is defined by:

$$H(X) = - \sum_{x \in X} p(x) \log_2 p(x) = E_p \log_2 \frac{1}{p(x)}$$

Intuitively, the entropy $H(X)$ of a discrete random variable X is a measure of the amount of uncertainty associated with the value of X when only its distribution is known; it is a measure of the amount of information required on the average to describe the random variable (we could construct a code with average description length $H(X)$).

The entropy is expressed in bits. For example, the entropy of a fair coin toss is 1 bit.



Kullback–Leibler Divergence

For discrete probability distributions p and q defined on the same probability space, the KL-divergence between p and q is defined as:

$$D_{KL}(p||q) = - \sum_{x \in X} p(x) \log_2 \frac{q(x)}{p(x)} = \sum_{x \in X} p(x) \log_2 \frac{p(x)}{q(x)}$$

For distributions p and q of a continuous random variable, the KL-divergence is defined to be the integral:

$$D_{KL}(p||q) = \int_{-\infty}^{\infty} p(x) \log_2 \frac{p(x)}{q(x)} dx$$

More generally:

$$D_{KL}(p||q) = E_p \log_2 \frac{p(x)}{q(x)}$$

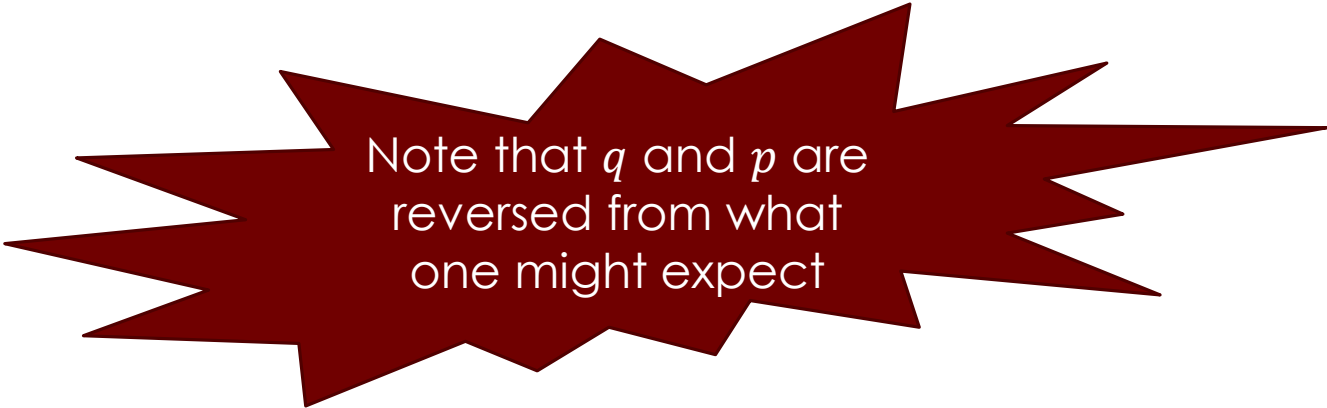
Kullback–Leibler Divergence

$D_{KL}(p||q)$ is a measure of the inefficiency of assuming that the distribution is q when the true distribution is p . For example, if we knew the true distribution p of the random variable, we could construct a code with average description length $H(p)$. If, instead, we used the code for a distribution q , we would need $H(p) + D_{KL}(p||q)$ bits on the average to describe the random variable.

Intuitively, there are three cases:

- If p is high and q is high then we are happy.
- If p is high and q is low then we pay a price.
- If p is low then we don't care (because of the expectation).

Coming Back to Variational Inference



Note that q and p are reversed from what one might expect

We want:

$$\min D_{KL}(q(\mathbf{Z})||p(\mathbf{Z}|\mathbf{X}))$$

The Evidence Lower Bound (ELBO)

We actually can't minimize the KL-divergence exactly, but we can minimize a function that is equal to it up to a constant. This is the evidence lower bound (ELBO).

$$D_{KL}(q(\mathbf{Z})||p(\mathbf{Z}|\mathbf{X})) = E_q \log_2 \frac{q(\mathbf{Z})}{p(\mathbf{Z}|\mathbf{X})} = - \underbrace{\left(E_q(\log_2 p(\mathbf{Z}, \mathbf{X})) - E_q(\log_2 q(\mathbf{Z})) \right)}_{\text{ELBO}} + \log_2 p(\mathbf{X})$$

Notice that $\log_2 p(\mathbf{X})$ does not depend on q . So, as a function of the variational distribution, minimizing the KL-divergence is the same as maximizing the ELBO.

Variational Inference

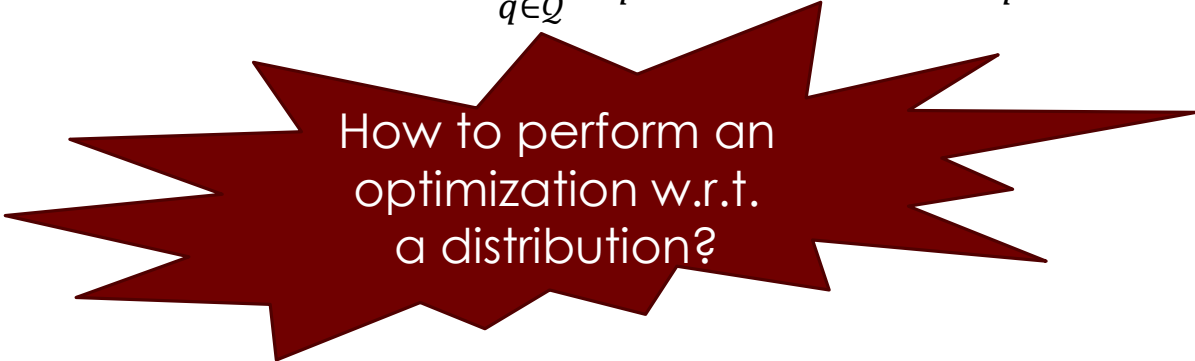
Final optimization problem:

$$\begin{aligned} & \max_{q \in \mathcal{Q}} E_q(\log_2 p(\mathbf{Z}, \mathbf{X})) - E_q(\log_2 q(\mathbf{Z})) \\ & E_q(\log_2 p(\mathbf{Z}, \mathbf{X})) - E_q(\log_2 q(\mathbf{Z})) = E_q \left(\log_2 \frac{p(\mathbf{Z}, \mathbf{X})}{q(\mathbf{Z})} \right) = E_q \left(\log_2 \frac{p(\mathbf{X}|\mathbf{Z})p(\mathbf{Z})}{q(\mathbf{Z})} \right) = \\ & = \underbrace{E_q(\log_2 p(\mathbf{X}|\mathbf{Z}))}_{\text{data term}} - \underbrace{D_{KL}(q(\mathbf{Z})||p(\mathbf{Z}))}_{\text{regularizer}} \end{aligned}$$

Variational Inference

Final optimization problem:

$$\max_{q \in \mathcal{Q}} E_q(\log_2 p(\mathbf{Z}, \mathbf{X})) - E_q(\log_2 q(\mathbf{Z}))$$



How to perform an
optimization w.r.t.
a distribution?

- Mean field approximation (factorized family)
- Parametric approximation (parametric family)

Example: Probabilistic PCA

Probabilistic PCA

Probabilistic principal components analysis (PCA) is a dimensionality reduction technique that analyzes data via a lower dimensional latent space. Probabilistic PCA generalizes classical PCA. It is often used when there are missing values in the data or for multidimensional scaling.

The Model

Consider a data set $X = \{x_n\}$ of N data points, where each data point is d -dimensional, $x_n \in \mathbb{R}^d$. We aim to represent each x_n under a latent variable $z_n \in \mathbb{R}^k$ with lower dimension, $k < d$. The set of principal axes \mathbf{W} relates the latent variables to the data.

Specifically, we assume that each latent variable is normally distributed:

$$z_n \sim N(\mathbf{0}, \mathbf{I})$$

The corresponding data point is generated via a projection:

$$x_n | z_n \sim N(\mathbf{W}z_n, \sigma^2 \mathbf{I})$$

where the matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$ are known as the principal axes. In probabilistic PCA, we are typically interested in estimating the principal axes \mathbf{W} and the noise term σ^2 .

Probabilistic PCA generalizes classical PCA. Marginalizing out the latent variable, the distribution of each data point is:

$$x_n \sim N(\mathbf{0}, \mathbf{W}\mathbf{W}' + \sigma^2 \mathbf{I})$$

Classical PCA is the specific case of probabilistic PCA when the covariance of the noise becomes infinitesimally small: $\sigma^2 \rightarrow 0$. In our analysis, we assume σ is known.

The Model in TFP

```
def probabilistic_pca(data_dim, latent_dim, num_datapoints, stddv_datapoints): # (unmodeled) data
    w = ed.Normal(loc=tf.zeros([data_dim, latent_dim]),
                  scale=2.0 * tf.ones([data_dim, latent_dim]),
                  name="w") # parameter
    z = ed.Normal(loc=tf.zeros([latent_dim, num_datapoints]),
                  scale=tf.ones([latent_dim, num_datapoints]),
                  name="z") # parameter
    x = ed.Normal(loc=tf.matmul(w, z),
                  scale=stddv_datapoints * tf.ones([data_dim, num_datapoints]),
                  name="x") # (modeled) data
    return x, (w, z)
```

```
log_joint = ed.make_log_joint_fn(probabilistic_pca)
```

Using the Edward2 Model to Generate Data

```
num_datapoints = 5000
data_dim = 2
latent_dim = 1
stddev_datapoints = 0.5

model = probabilistic_pca(data_dim=data_dim,
                          latent_dim=latent_dim,
                          num_datapoints=num_datapoints,
                          stddev_datapoints=stddev_datapoints)

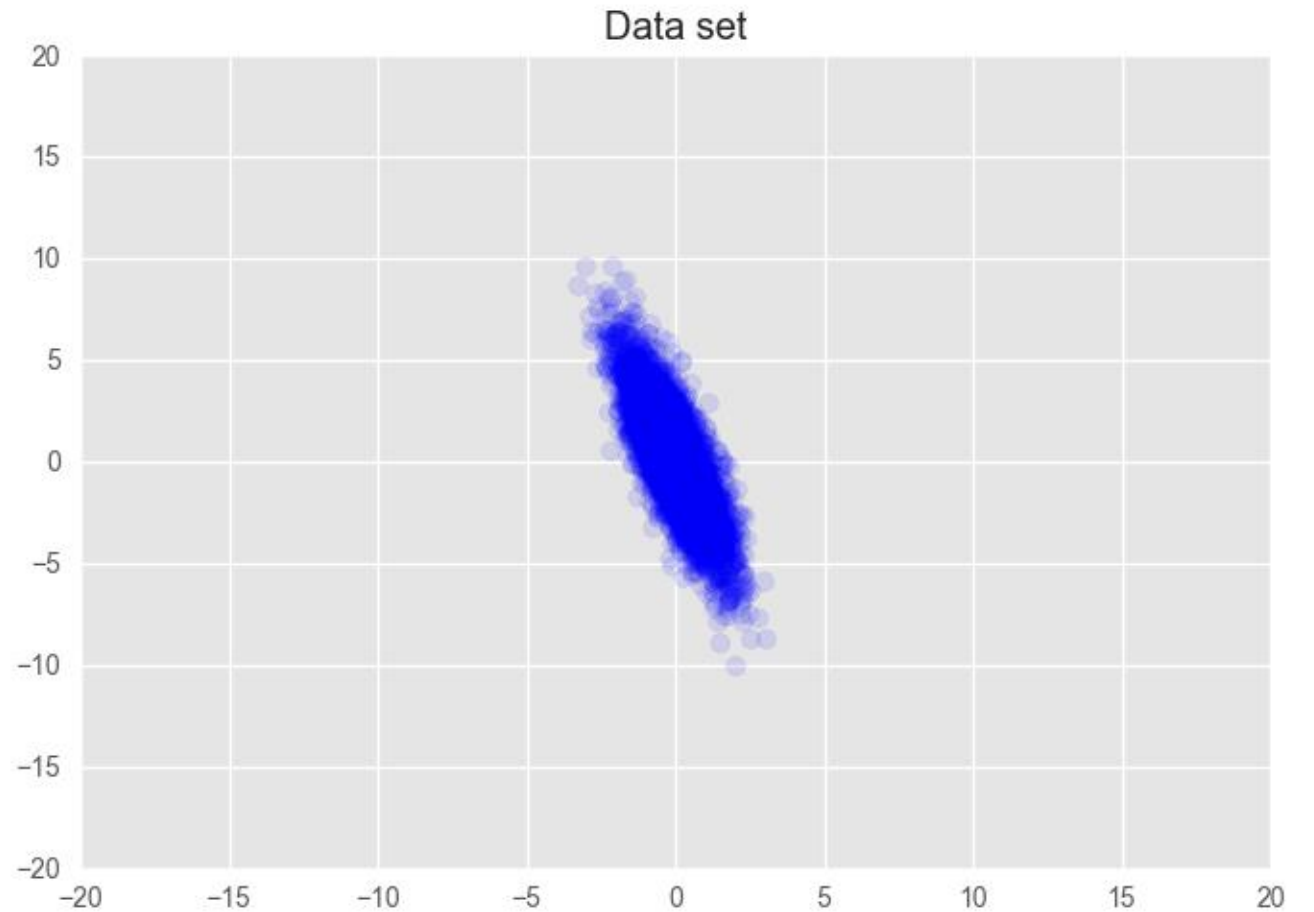
with tf.Session() as sess:
    x_train, (actual_w, actual_z) = sess.run(model)

print("Principal axes:")
print(actual_w)
```

The Data

Principal axes:

$\begin{bmatrix} -0.7300116 \\ 2.5360086 \end{bmatrix}$



MAP Inference

We first use MAP for the point estimate of latent variables.

MAP inference is done by calculating the values of **W** and **Z** that maximize the posterior density:

$$p(\mathbf{W}, \mathbf{Z} | \mathbf{X}) \propto p(\mathbf{W}, \mathbf{Z}, \mathbf{X})$$

```
tf.reset_default_graph()

w = tf.Variable(np.ones([data_dim, latent_dim]), dtype=tf.float32)
z = tf.Variable(np.ones([latent_dim, num_datapoints]), dtype=tf.float32)

def target(w, z):
    """Unnormalized target density as a function of the parameters."""
    return log_joint(data_dim=data_dim,
                     latent_dim=latent_dim,
                     num_datapoints=num_datapoints,
                     stddv_datapoints=stddv_datapoints,
                     w=w, z=z, x=x_train)

energy = -target(w, z)

optimizer = tf.train.AdamOptimizer(learning_rate=0.05)
train = optimizer.minimize(energy)
```


MAP Inference: Training

```
init = tf.global_variables_initializer()

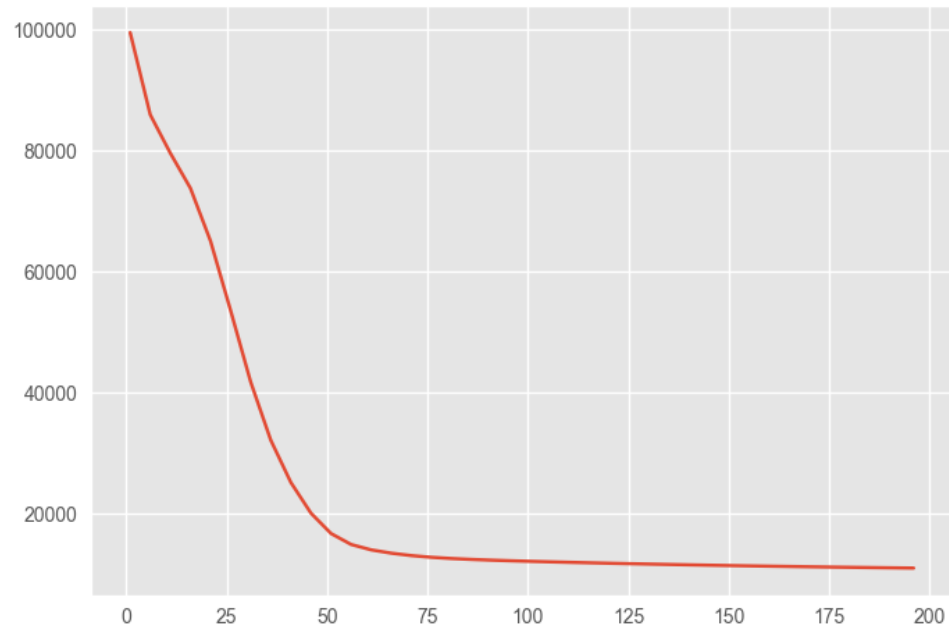
t = []
num_epochs = 200

with tf.Session() as sess:
    sess.run(init)

    for i in range(num_epochs):
        sess.run(train)
        if i % 5 == 0:
            cE, cw, cz = sess.run([energy, w, z])
            t.append(cE)

w_inferred_map = sess.run(w)
z_inferred_map = sess.run(z)
```

MAP Inference: Training



Model & Inference Criticism

- Assessing how model predictions match the true data
- Assessing how data generated from the model matches the true data

Manipulating Model Computation: Interceptors

An interceptor is a function that acts on another function f and its arguments $*args, **kwargs$. It performs various computations before returning an output (typically $f(*args, **kwargs)$: the result of applying the function itself). The `ed.interception` context manager pushes interceptors onto a stack, and any interceptable function is intercepted by the stack. All random variable constructors are interceptable.

In particular, we make predictions with a model learned posterior means rather than with its priors.

Criticism

We can use the Edward2 model to sample data for the inferred values for **W** and **Z** and compare to the actual dataset we conditioned on.

```
def replace_latents(w=actual_w, z=actual_z):
```

```
    def interceptor(rv_constructor, *rv_args, **rv_kwargs):
```

```
        """Replaces the priors with actual values to generate samples from."""
```

```
        name = rv_kwargs.pop("name")
```

```
        if name == "w":
```

```
            rv_kwargs["value"] = w
```

```
        elif name == "z":
```

```
            rv_kwargs["value"] = z
```

```
        return rv_constructor(*rv_args, **rv_kwargs)
```

```
    return interceptor
```

```
with ed.interception(replace_latents(w_inferred_map, z_inferred_map)):
```

```
    generate = probabilistic_pca(
```

```
        data_dim=data_dim, latent_dim=latent_dim,
```

```
        num_datapoints=num_datapoints, stddv_datapoints=stddv_datapoints)
```

```
with tf.Session() as sess:
```

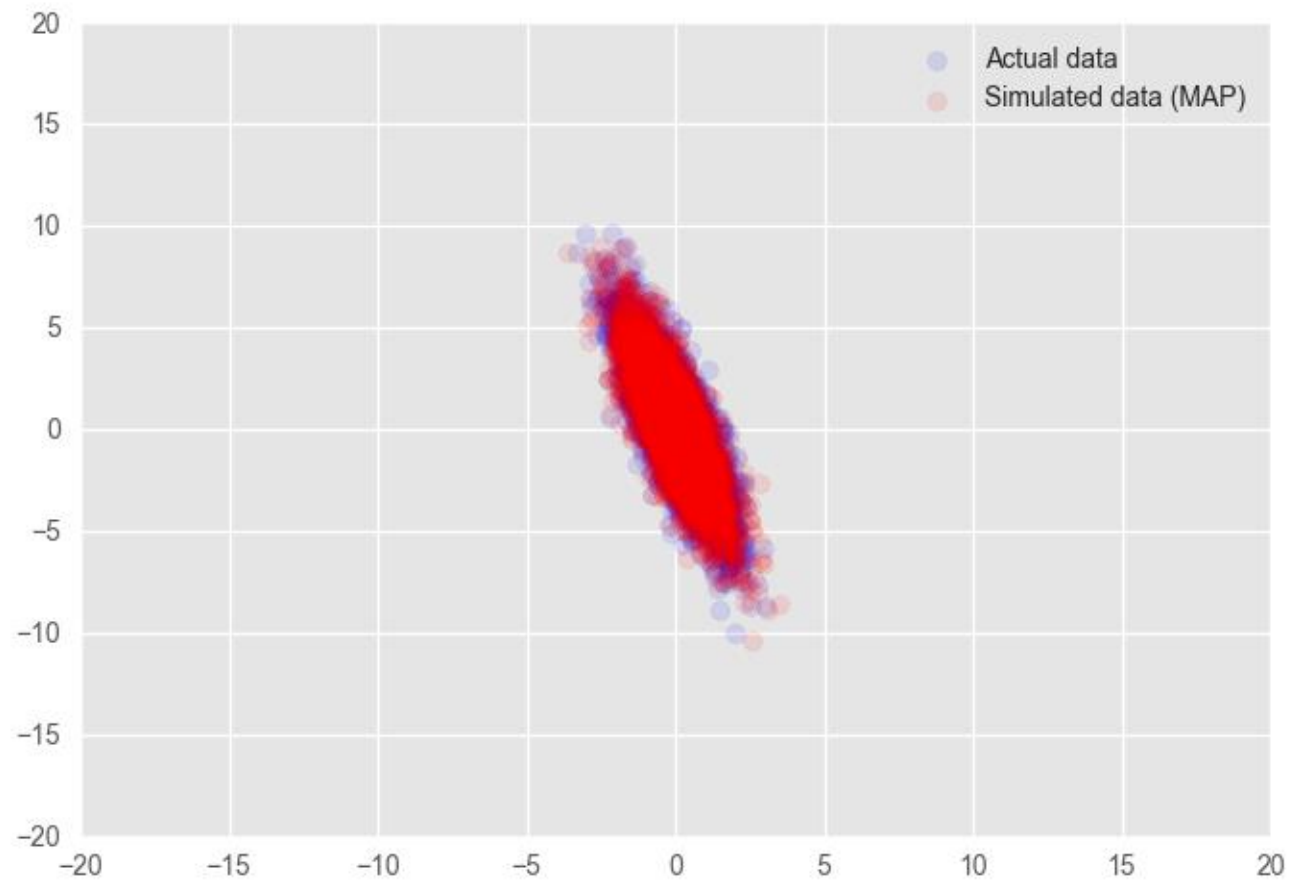
```
    x_generated, _ = sess.run(generate)
```

The Results

```
print("MAP-estimated axes:")  
print(w_inferred_map)
```

MAP-estimated axes:

```
[[ -0.8976625]  
 [ 3.1588674]]
```



Variational Inference

We next use variational inference, where the posterior distribution $p(\mathbf{W}, \mathbf{Z}|\mathbf{X})$ is approximated using a variational distribution $q(\mathbf{W}, \mathbf{Z})$ parametrized by λ .

The aim is to find the variational parameters λ that minimize the KL-divergence between q and the posterior: $D_{KL}(q(\mathbf{W}, \mathbf{Z})||p(\mathbf{W}, \mathbf{Z}|\mathbf{X}))$, or equivalently, that maximize the ELBO:

$$E_q(\log_2 p(\mathbf{W}, \mathbf{Z}, \mathbf{X})) - E_q(\log_2 q(\mathbf{W}, \mathbf{Z}))$$

```
tf.reset_default_graph()
```

```
def variational_model(qw_mean, qw_stddv, qz_mean, qz_stddv):  
    qw = ed.Normal(loc=qw_mean, scale=qw_stddv, name="qw")  
    qz = ed.Normal(loc=qz_mean, scale=qz_stddv, name="qz")  
    return qw, qz
```

```
log_q = ed.make_log_joint_fn(variational_model)
```

```
def target_q(qw, qz):  
    return log_q(qw_mean=qw_mean, qw_stddv=qw_stddv,  
                 qz_mean=qz_mean, qz_stddv=qz_stddv,  
                 qw=qw, qz=qz)
```

```
qw_mean = tf.Variable(np.ones([data_dim, latent_dim]), dtype=tf.float32)  
qz_mean = tf.Variable(np.ones([latent_dim, num_datapoints]), dtype=tf.float32)  
qw_stddv = tf.nn.softplus(tf.Variable(-4 * np.ones([data_dim, latent_dim]), dtype=tf.float32))  
qz_stddv = tf.nn.softplus(tf.Variable(-4 * np.ones([latent_dim, num_datapoints]), dtype=tf.float32))
```

```
qw, qz = variational_model(qw_mean=qw_mean, qw_stddv=qw_stddv,  
                           qz_mean=qz_mean, qz_stddv=qz_stddv)
```

```
energy = target(qw, qz)  
entropy = -target_q(qw, qz)
```

```
elbo = energy + entropy
```


Variational Inference: Training

```
optimizer = tf.train.AdamOptimizer(learning_rate = 0.05)
train = optimizer.minimize(-elbo)
```

```
init = tf.global_variables_initializer()
```

```
t = []
```

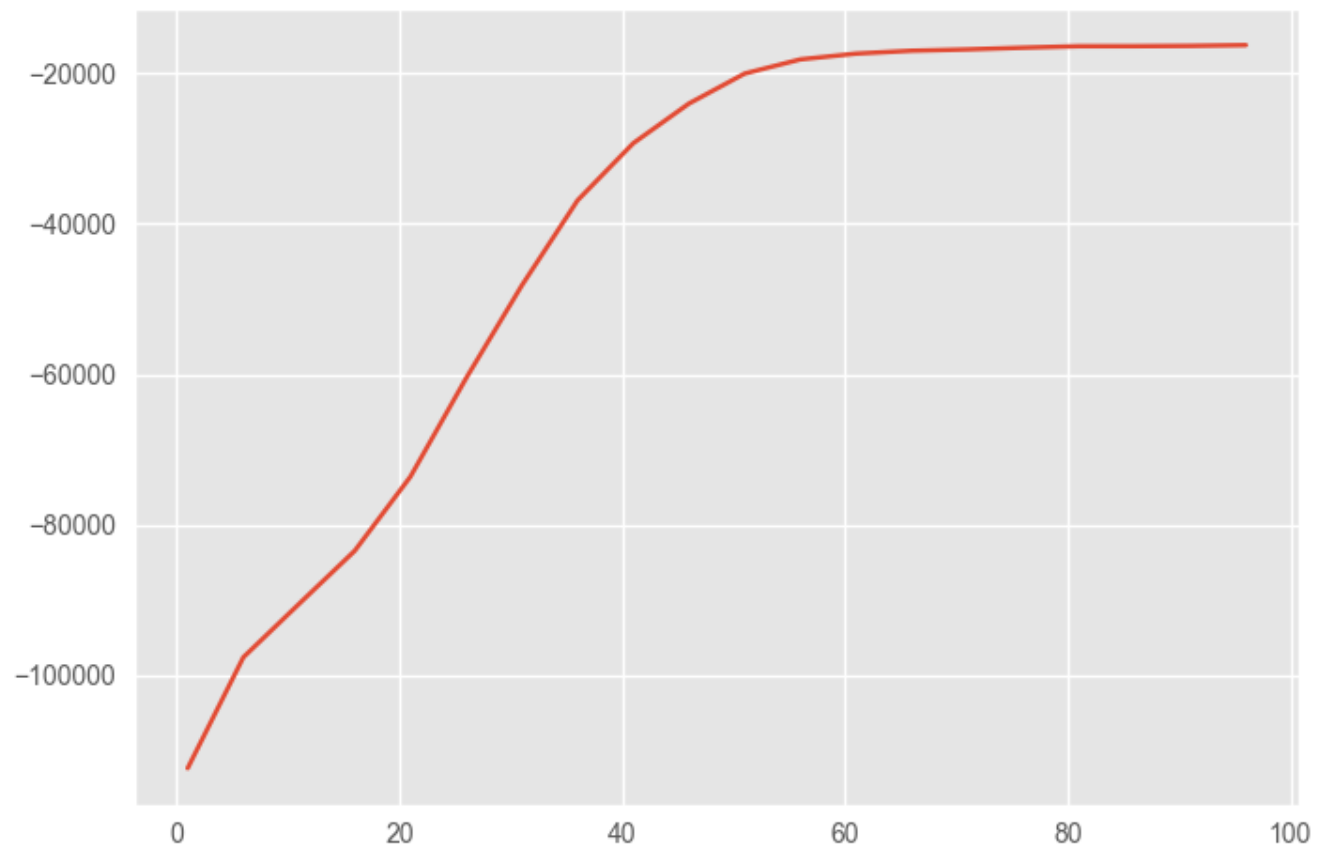
```
num_epochs = 100
```

```
with tf.Session() as sess:
    sess.run(init)
```

```
    for i in range(num_epochs):
        sess.run(train)
        if i % 5 == 0:
            t.append(sess.run([elbo]))
```

```
w_mean_inferred = sess.run(qw_mean)
w_stddev_inferred = sess.run(qw_stddev)
z_mean_inferred = sess.run(qz_mean)
z_stddev_inferred = sess.run(qz_stddev)
```

Variational Inference: Training



Criticism

We can use the Edward2 model to sample data for the inferred values for **W** and **Z** and compare to the actual dataset we conditioned on.

```
print("Inferred axes:")
print(w_mean_inferred)
print("Standard Deviation:")
print(w_stdv_inferred)

with ed.interception(replace_latents(w_mean_inferred, z_mean_inferred)):
    generate = probabilistic_pca(
        data_dim=data_dim, latent_dim=latent_dim,
        num_datapoints=num_datapoints, stdv_datapoints=stdv_datapoints)

with tf.Session() as sess:
    x_generated, _ = sess.run(generate)
```

The Results

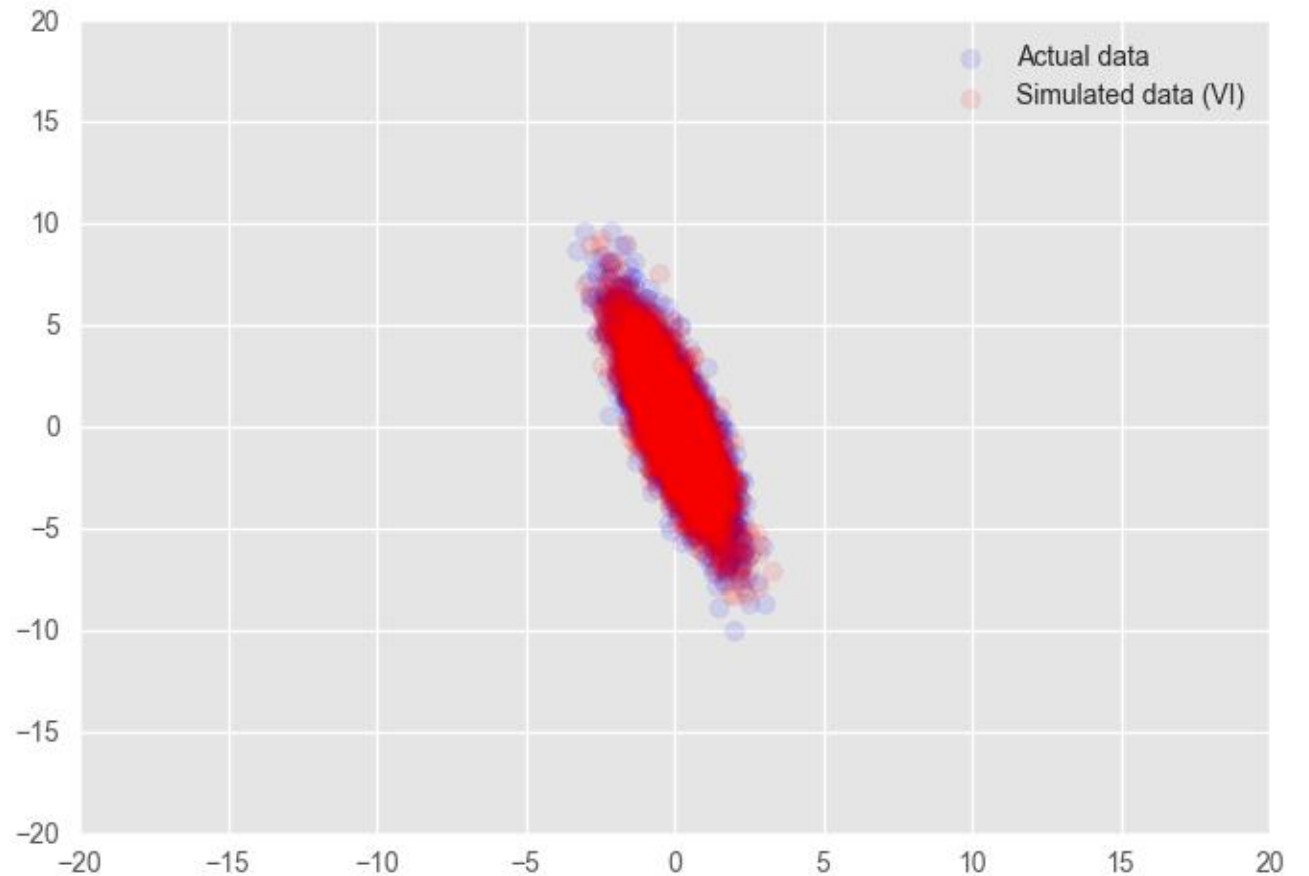
```
print("Inferred axes:")  
print(w_mean_inferred)  
print("Standard Deviation:")  
print(w_stddv_inferred)
```

Inferred axes:

```
[[ -0.6105784]  
 [  2.141177 ]]
```

Standard Deviation:

```
[[0.01481778]  
 [0.0108717 ]]
```



VAE: Variational Autoencoders

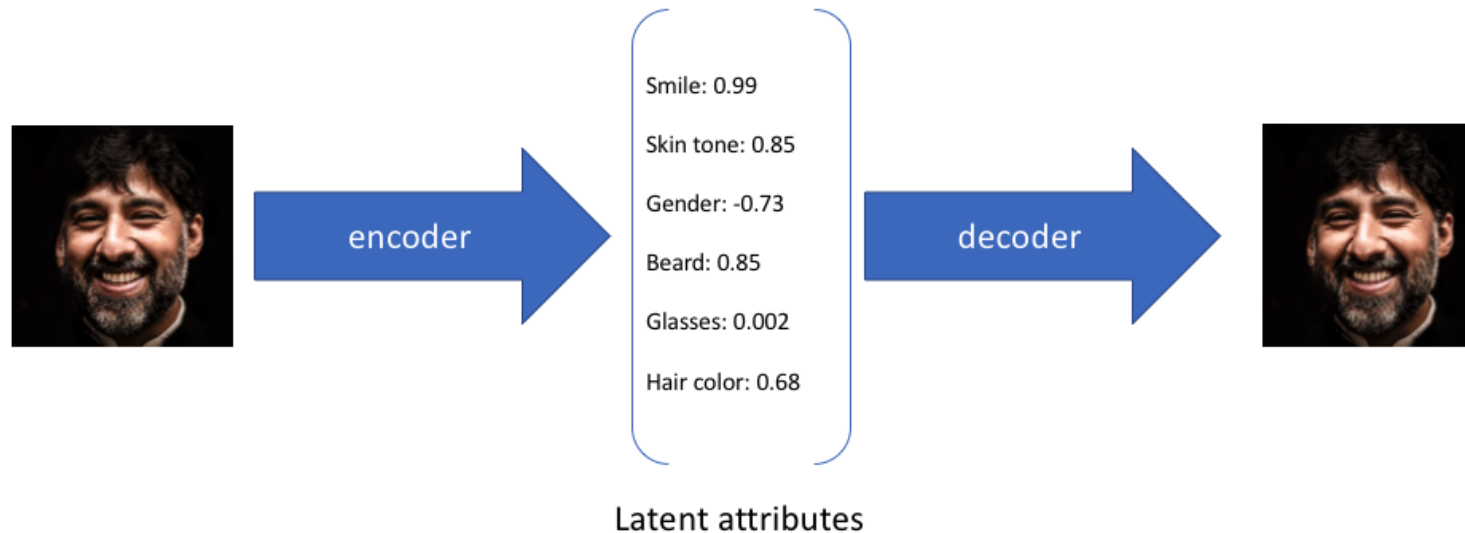
Diederik Kingma and Max Welling. Auto-Encoding Variational Bayes. In International Conference on Learning Representations, 2014. (<https://arxiv.org/abs/1312.6114>)

Autoencoders

- An *autoencoder* is a machine learning model which uses one learned system to represent data in some low-dimensional space and a second learned system to restore the low-dimensional representation to what would have otherwise been the input.
- The *encoder* converts the input data into an *encoding vector* where each dimension represents some learned attribute about the data. The *decoder* then subsequently takes these values and attempts to recreate the original input.

Autoencoders: Intuition

Let's suppose we've trained an autoencoder model on a large dataset of faces with a encoding dimension of 6. An ideal autoencoder will learn descriptive attributes of faces such as skin color, whether or not the person is wearing glasses, etc. in an attempt to describe an observation in some compressed representation:



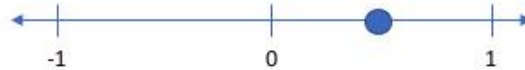
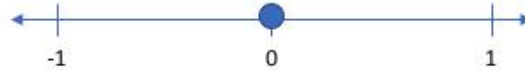
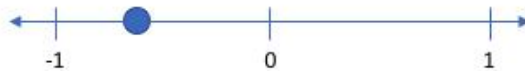
VAE

A variational autoencoder (VAE) provides a *probabilistic manner* for describing an observation in latent space. Thus, rather than building an encoder which outputs a single value to describe each latent state attribute, we'll formulate our encoder to describe a *probability distribution* for each latent attribute.

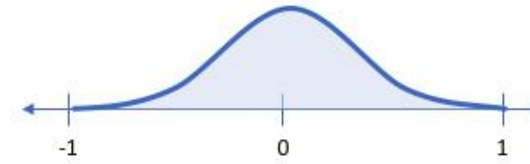
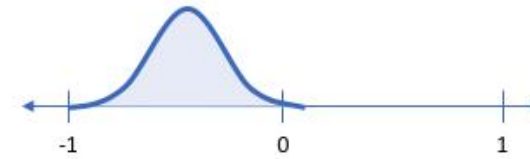
What single value would you assign for the smile attribute if you feed in a photo of the Mona Lisa?



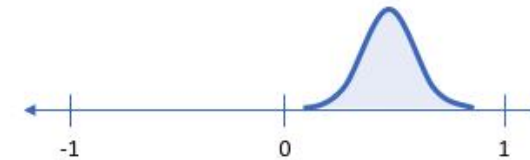
Smile (discrete value)



Smile (probability distribution)

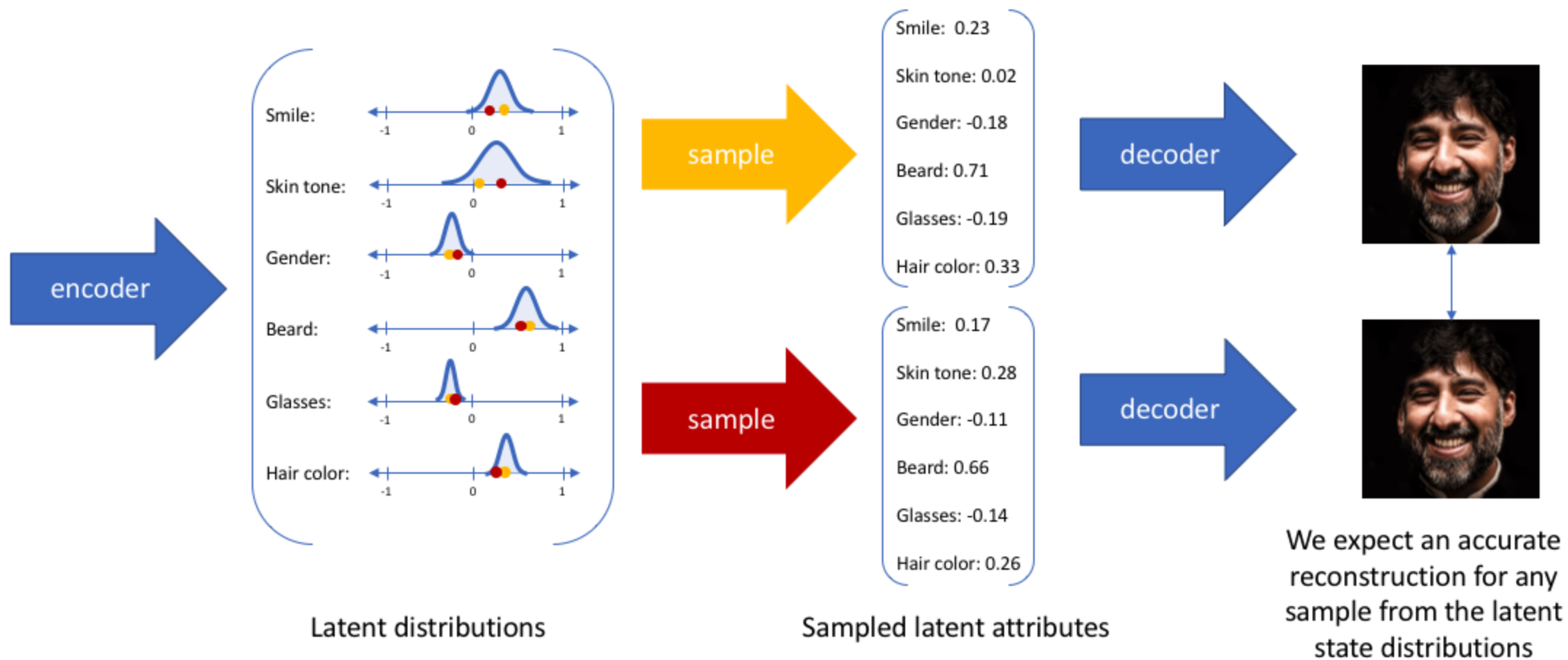


vs.



VAE

By constructing our encoder model to output a range of possible values (a statistical distribution) from which we'll randomly sample to feed into our decoder model, we're essentially enforcing a continuous, smooth latent space representation. For any sampling of the latent distributions, we're expecting our decoder model to be able to accurately reconstruct the input. Thus, values which are nearby to one another in latent space should correspond with very similar reconstructions.



VAE

- VAE comprises a probabilistic model over data and a variational model designed to approximate the former's posterior.
- The VAE defines a generative model in which a latent code Z is sampled from a prior $p(Z)$, then used to generate an observation X by way of a decoder $p(X|Z)$. The full reconstruction follows:

$X \sim p(X)$ # A random image from some dataset.

$Z \sim q(Z|X)$ # A random encoding of the original image ("encoder").

$\hat{X} \sim p(\hat{X}|Z)$ # A random reconstruction of the original image ("decoder").

VAE Loss Function

To fit the VAE, we assume an approximate representation of the posterior in the form of an encoder $q(Z|X)$. We minimize the KL divergence between $q(Z|X)$ and the true posterior $p(Z|X)$:

$$\min D_{KL}(q(\mathbf{Z}|\mathbf{X})||p(\mathbf{Z}|\mathbf{X}))$$

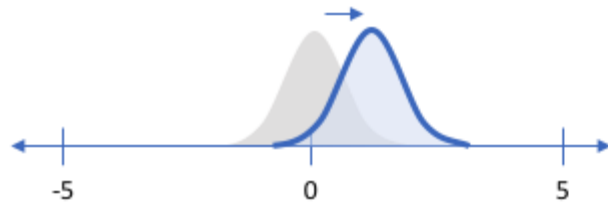
this is equivalent to maximizing the evidence lower bound (ELBO):

$$E_{q(\mathbf{z}|\mathbf{x})}(\log_2 p(\mathbf{X}|\mathbf{Z})) - D_{KL}(q(\mathbf{Z}|\mathbf{X})||p(\mathbf{Z}))$$

The first term represents the reconstruction likelihood (an expected reconstruction loss) and the second term is a kind of distributional regularizer (ensures that our learned distribution q is similar to the true prior distribution p).

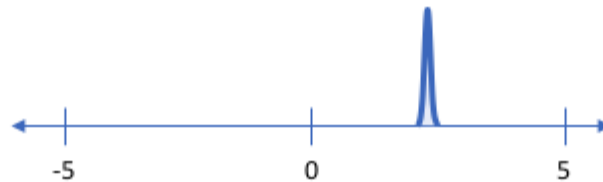
VAE Loss Function

Penalizing reconstruction loss encourages the distribution to describe the input



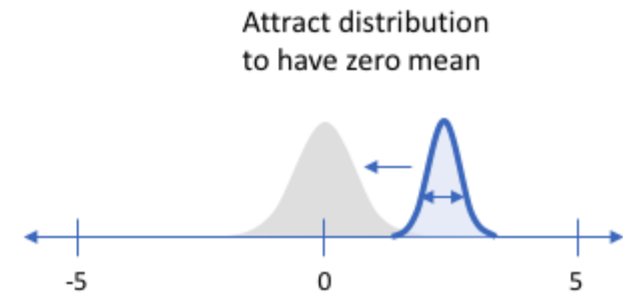
Our distribution deviates from the prior to describe some characteristic of the data

Without regularization, our network can “cheat” by learning narrow distributions



With a small enough variance, this distribution is effectively only representing a single value

Penalizing KL divergence acts as a regularizing force



Attract distribution to have zero mean

Ensure sufficient variance to yield a smooth latent space

VAE Loss Function

$$D_{KL}(p||q) = E_q \log_2 \frac{p(x)}{q(x)}$$

$$E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 p(\mathbf{X}|\mathbf{Z})) - D_{KL}(q(\mathbf{Z}|\mathbf{X})||p(\mathbf{Z}))$$

$$E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 p(\mathbf{X}|\mathbf{Z})) - E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 \frac{q(\mathbf{Z}|\mathbf{X})}{p(\mathbf{Z})})$$

$$E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 p(\mathbf{X}|\mathbf{Z})) - E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 q(\mathbf{Z}|\mathbf{X})) + E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 p(\mathbf{Z}))$$

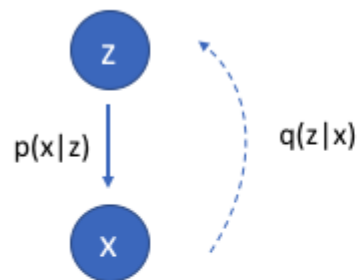
$$E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 p(\mathbf{X}|\mathbf{Z})p(\mathbf{Z})) - E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 q(\mathbf{Z}|\mathbf{X}))$$

$$E_{q(\mathbf{Z}|\mathbf{X})}(\log_2 p(\mathbf{X},\mathbf{Z})) + H(q(\mathbf{Z}|\mathbf{X}))$$

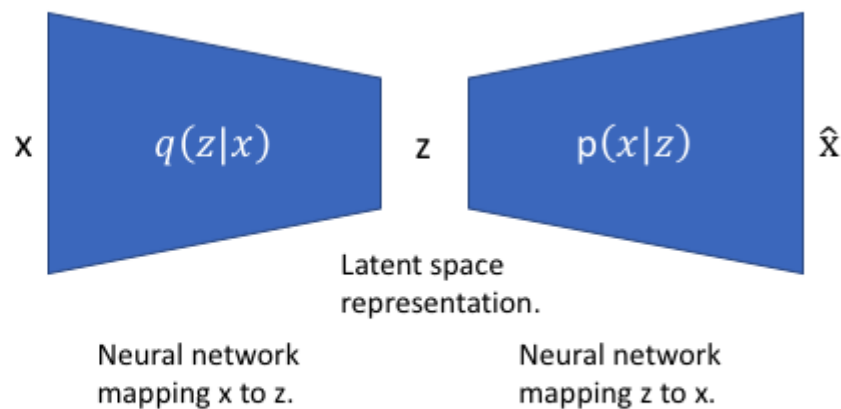
$$H(X) = - \sum_{x \in X} p(x) \log_2 p(x)$$

VAE

We can further construct the VAE probabilistic model into a neural network architecture where the encoder model learns a mapping from \mathbf{X} to \mathbf{Z} and the decoder model learns a mapping from \mathbf{Z} back to \mathbf{X} .



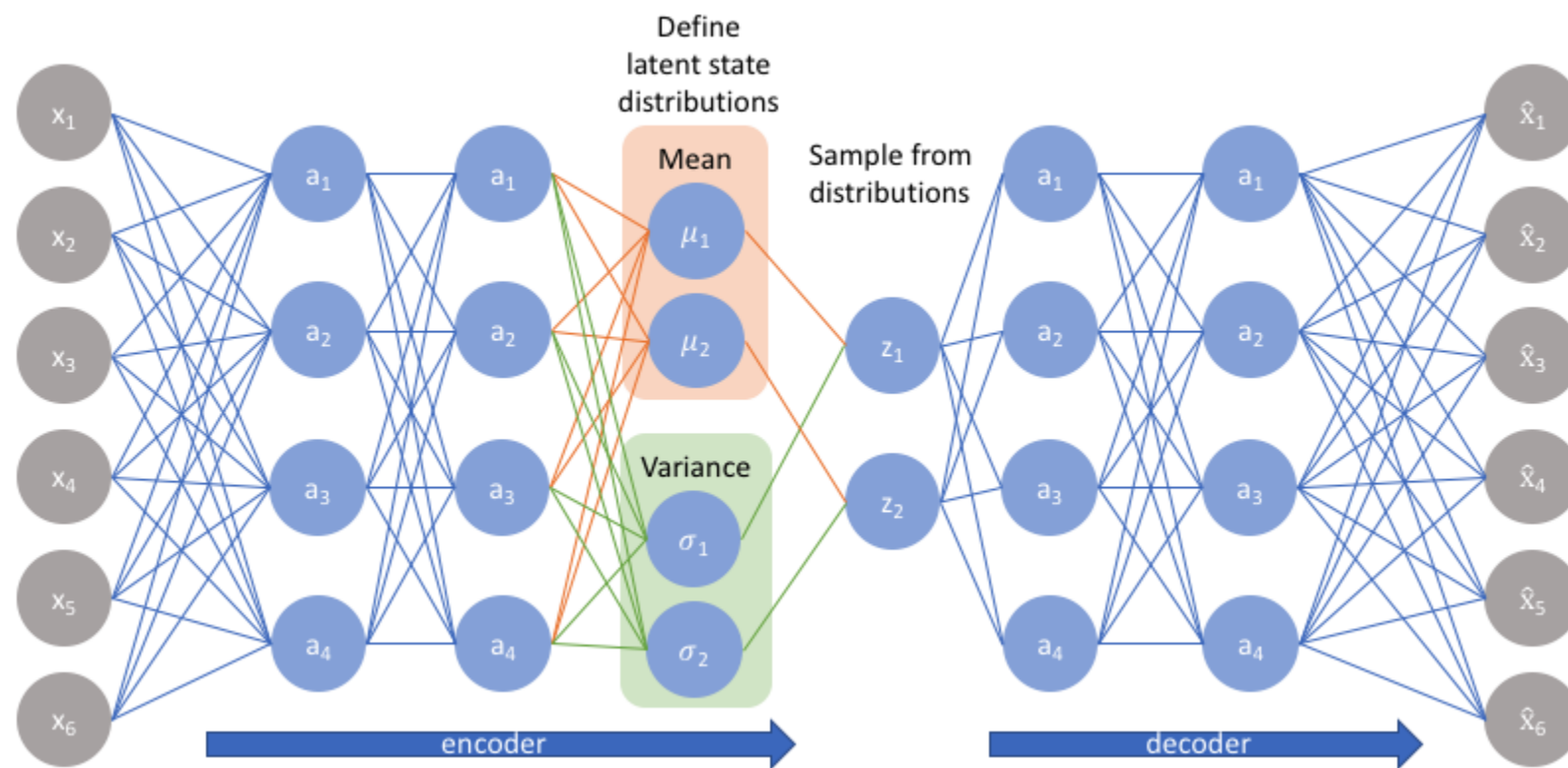
We'd like to use our observations to understand the hidden variable.



VAE

Rather than directly outputting values for the latent state as we would in a standard autoencoder, the encoder model of a VAE will output parameters describing a distribution for each dimension in the latent space. If we're assuming that our prior follows a normal distribution, we'll output two vectors describing the mean and variance of the latent state distributions. If we were to build a true multivariate Gaussian model, we'd need to define a covariance matrix describing how each of the dimensions are correlated. However, we'll make a simplifying assumption that our covariance matrix only has nonzero values on the diagonal, allowing us to describe this information in a simple vector.

VAE



VAE in TFP and Keras

```
xdim, d, zdim = 784, 200, 2
gen_net = tf.keras.Sequential([tf.keras.layers.Dense(d, activation=tf.nn.elu),
                               tf.keras.layers.Dense(d, activation=tf.nn.elu),
                               tf.keras.layers.Dense(xdim, activation=None)])
inference_net = tf.keras.Sequential([tf.keras.layers.Dense(d, activation=tf.nn.elu),
                                     tf.keras.layers.Dense(d, activation=tf.nn.elu),
                                     tf.keras.layers.Dense(2*zdim, activation=None)])
```

VAE in TFP and Keras

```
def generative_model(batch_size=None):  
    z = ed.MultivariateNormalDiag(loc=tf.zeros([zdim]), scale_identity_multiplier=1.,  
                                  sample_shape=batch_size, name='z')  
    x = ed.Bernoulli(gen_net(z), name='x')  
    return x
```

```
def variational_model(x):  
    outs = inference_net(x)  
    _loc, _scale = outs[:, :zdim], 1e-3 + tf.nn.softplus(outs[:, zdim:])  
    z = ed.MultivariateNormalDiag(loc=_loc, scale_diag=_scale, name='z_posterior')  
    return z
```

```
log_joint = ed.make_log_joint_fn(generative_model)
```

VAE in TFP and Keras

```
def get_loss(inputs):  
    z = variational_model(inputs)  
    energy = log_joint(z=z, x=inputs)  
    entropy = tf.reduce_sum(z.distribution.entropy())  
    return (-energy - entropy)
```

VAE in TFP and Keras

```
mnist = tf.keras.datasets.mnist
(x_train, _), _ = mnist.load_data()
x_train = np.rint((x_train / 255.).reshape(-1, xdim)).astype(np.float32)
assert x_train.max() <= 1 and x_train.min() >= 0 #assert correctly
```

```
bs = 64
dataset = tf.data.Dataset.from_tensor_slices(x_train)
dataset = dataset.shuffle((len(x_train)))
dataset = dataset.batch(bs, drop_remainder=True)
dataset = dataset.prefetch(16)
data_iterator = dataset.make_initializable_iterator()
batch_x = data_iterator.get_next()
```

```
loss = get_loss(batch_x) / bs # divide by bs since loss is summed.
opt = tf.contrib.opt.AdamWOptimizer(0, learning_rate=1e-3)
train = opt.minimize(loss)
```

VAE in TFP and Keras

```
init = tf.global_variables_initializer()

sess = tf.Session()
sess.run(init)
for i in range(50): # 50 epochs; takes <2m on my GPU.
    sess.run(data_iterator.initializer)
    try:
        while True:
            _, _loss = sess.run([train, loss])
    except tf.errors.OutOfRangeError:
        print(_loss, ', end of epoch {}'.format(i))
```


Training

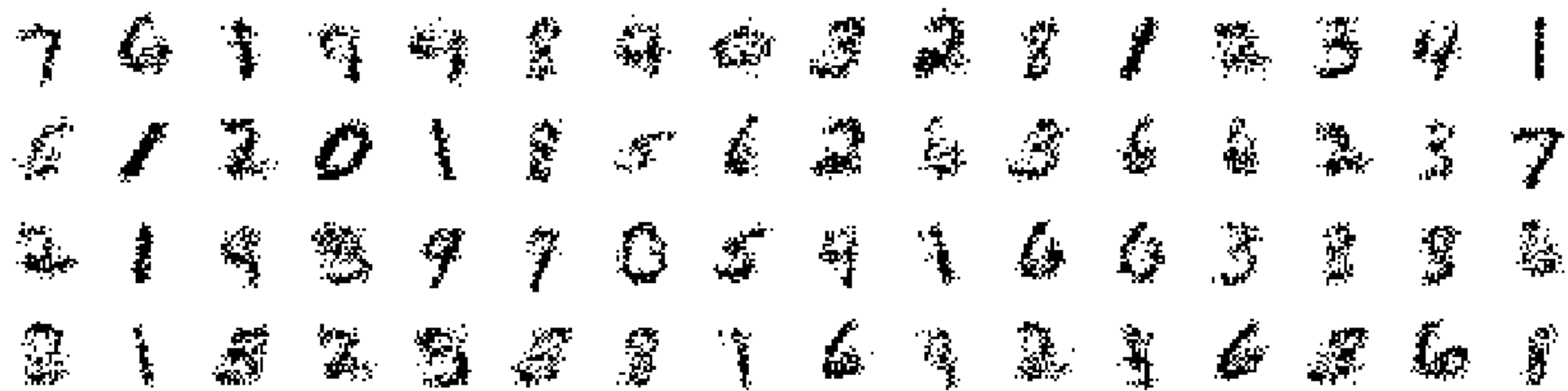
```
mpopescu@turing: ~/My/TFP
2019-01-08 08:13:02.672977: I tensorflow/core/common_runtime/gpu/gpu_device.cc:115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 10405 MB memory) -> physical GPU (device: 0, name: GeForce GTX 1080 Ti, pci bus id: 0000:02:00.0, compute capability: 6.1)
2019-01-08 08:13:02.673466: I tensorflow/core/common_runtime/gpu/gpu_device.cc:115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:1 with 10405 MB memory) -> physical GPU (device: 1, name: GeForce GTX 1080 Ti, pci bus id: 0000:04:00.0, compute capability: 6.1)
2019-01-08 08:13:02.673810: I tensorflow/core/common_runtime/gpu/gpu_device.cc:115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:2 with 10405 MB memory) -> physical GPU (device: 2, name: GeForce GTX 1080 Ti, pci bus id: 0000:83:00.0, compute capability: 6.1)
2019-01-08 08:13:02.674183: I tensorflow/core/common_runtime/gpu/gpu_device.cc:115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:3 with 10405 MB memory) -> physical GPU (device: 3, name: GeForce GTX 1080 Ti, pci bus id: 0000:84:00.0, compute capability: 6.1)
169.46884 , end of epoch 0
172.11403 , end of epoch 1
156.98259 , end of epoch 2
160.30925 , end of epoch 3
157.5325 , end of epoch 4
142.57921 , end of epoch 5
148.8006 , end of epoch 6
155.15837 , end of epoch 7
```

```
mpopescu@turing: ~/My/TFP
144.15254 , end of epoch 27
142.50539 , end of epoch 28
130.39604 , end of epoch 29
142.58778 , end of epoch 30
127.93375 , end of epoch 31
135.87712 , end of epoch 32
140.75005 , end of epoch 33
135.43109 , end of epoch 34
142.5304 , end of epoch 35
144.01065 , end of epoch 36
138.1394 , end of epoch 37
149.9505 , end of epoch 38
139.58916 , end of epoch 39
139.48537 , end of epoch 40
135.03062 , end of epoch 41
142.47807 , end of epoch 42
130.82309 , end of epoch 43
145.19824 , end of epoch 44
133.09439 , end of epoch 45
141.77513 , end of epoch 46
137.96176 , end of epoch 47
141.08607 , end of epoch 48
130.66998 , end of epoch 49
(tf) mpopescu@turing:~/My/TFP$
```

VAE as a Generative Model

```
# Generate imgs.  
generated = generative_model(batch_size=64)  
img = sess.run(generated).reshape(-1, 28, 28)  
plt.figure(figsize=(16, 4))  
for i in range(64):  
    plt.subplot(4, 16, i+1)  
    plt.imshow(img[i], cmap='Greys')  
    plt.axis('off')  
plt.savefig('vae1.png')  
plt.show()
```

The Results



Reconstruct Images

```
def replace_z(z):
    def interceptor(rv_constructor, *rv_args, **rv_kwargs):
        name = rv_kwargs.pop('name')
        if name == 'z':
            rv_kwargs['value'] = z
        return rv_constructor(*rv_args, **rv_kwargs)
    return interceptor
```

```
# Reconstruct imgs. Left is original, right is reconstructed.
batch = batch_x
var_z = variational_model(batch)
with ed.interception(replace_z(z=var_z)):
    recon = generative_model()
sess.run(data_iterator.initializer)
_batch, _recon = sess.run([batch, recon])
_batch, _recon = _batch.reshape(-1, 28, 28), _recon.reshape(-1, 28, 28)
plt.figure(figsize=(16, 4))
for i in range(32):
    plt.subplot(4, 16, 2*i+1)
    plt.imshow(_batch[i], cmap='Greys')
    plt.axis('off')
    plt.subplot(4, 16, 2*i+2)
    plt.imshow(_recon[i], cmap='Greys')
    plt.axis('off')
plt.savefig('vae2.png')
plt.show()
```

The Results

5	5	8	8	2	2	9	9	6	6	2	2	2	2	0	0
9	9	5	5	5	5	0	0	9	9	7	7	5	5	6	6
8	8	8	8	3	3	0	0	2	2	4	4	0	0	9	9
8	8	3	3	0	0	2	2	8	8	3	3	8	8	8	8

Visualization of Latent Space

```
from scipy.stats import norm
N = 20
invcdf = norm.ppf(np.linspace(.1, .9, num=N))
zs = [[invcdf[i], invcdf[j]] for i in range(N) for j in range(N)]
with ed.interception(replace_z(z=zs)):
    recon = generative_model(N*N)
sess.run(data_iterator.initializer)
_recon = sess.run(recon).reshape(-1, 28, 28)
_batch, _recon = _batch.reshape(-1, 28, 28), _recon.reshape(-1, 28, 28)
plt.figure(figsize=(16, 16))
for i in range(N):
    for j in range(N):
        plt.subplot(N, N, N*j+i+1)
        plt.imshow(_recon[N*j+i], cmap='Greys')
        plt.axis('off')
plt.savefig('vae3.png')
plt.show()
```

[illegible]