Lecture 24

VARIATIONAL INFERENCE

AM 207

Latent variables

- instead of bayesian vs frequentist, think hidden vs not hidden
- key concept: full data likelihood vs partial data likelihood
- probabilistic model is a *joint distribution* $p(\mathbf{x}, \mathbf{z})$
- observed variables $\mathbf{x}$ corresponding to data, and latent variables $\mathbf{z}$
- can be both "global" parameters and per-data-point cluster belonging

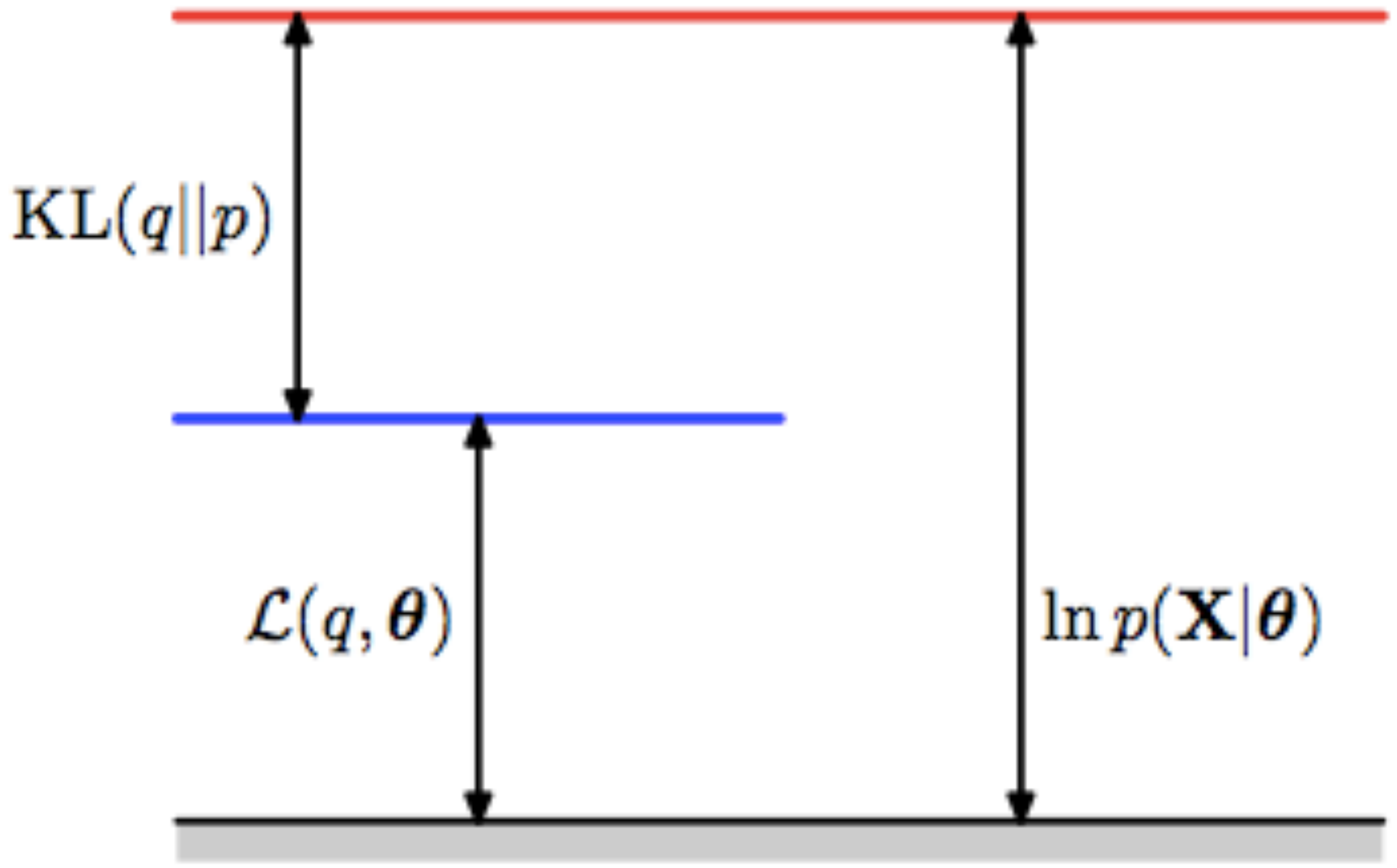
x-data likelihood

$$log\, p(x|\theta) = E_q[log \frac{p(x,z|\theta)}{q}] + D_{KL}(q,p)$$

If we define the ELBO or Evidence Lower bound as:

$$\mathcal{L}(q,\theta) = E_q[log \frac{p(x,z|\theta)}{q}]$$

then $log\, p(x|\theta)$ = ELBO + KL-divergence


$\mathrm{KL}(q||p)$
$\mathcal{L}(q,\boldsymbol{\theta})$
$\ln p(\mathbf{X}|\boldsymbol{\theta})$

- KL divergence only 0 when $p = q$ exactly everywhere
- minimizing KL means maximizing ELBO
- ELBO $\mathcal{L}(q, \theta)$ is a lower bound on the log-likelihood.
- ELBO is average full-data likelihood minus entropy of q:

$$\mathcal{L}(q, \theta) = E_q[log \frac{p(x, z|\theta)}{q}] = E_q[log p(x, z|\theta)] - E_q[log\, q]$$

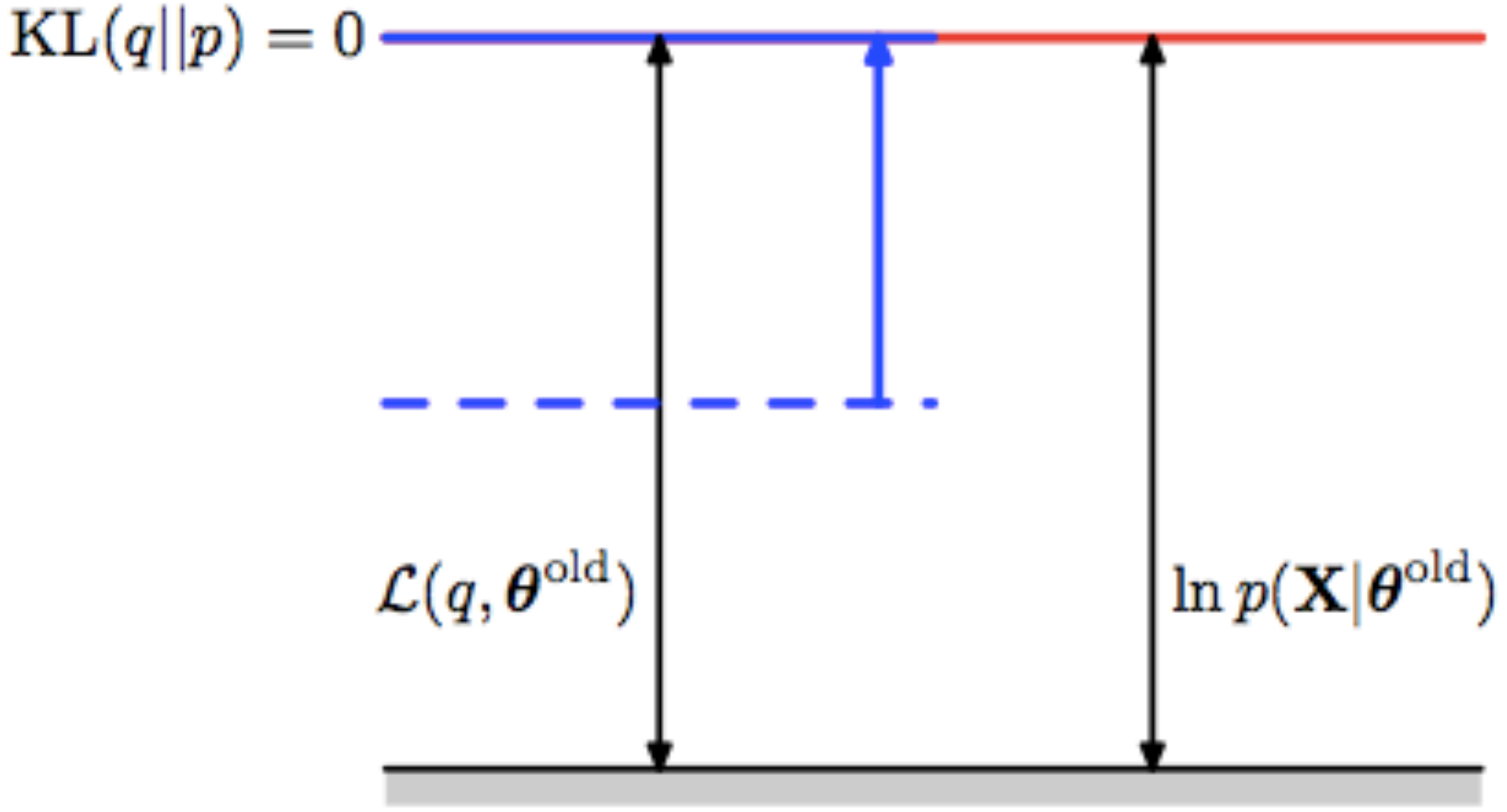
E-step conceptually

Choose at some (possibly initial) value of the parameters θ_{old},

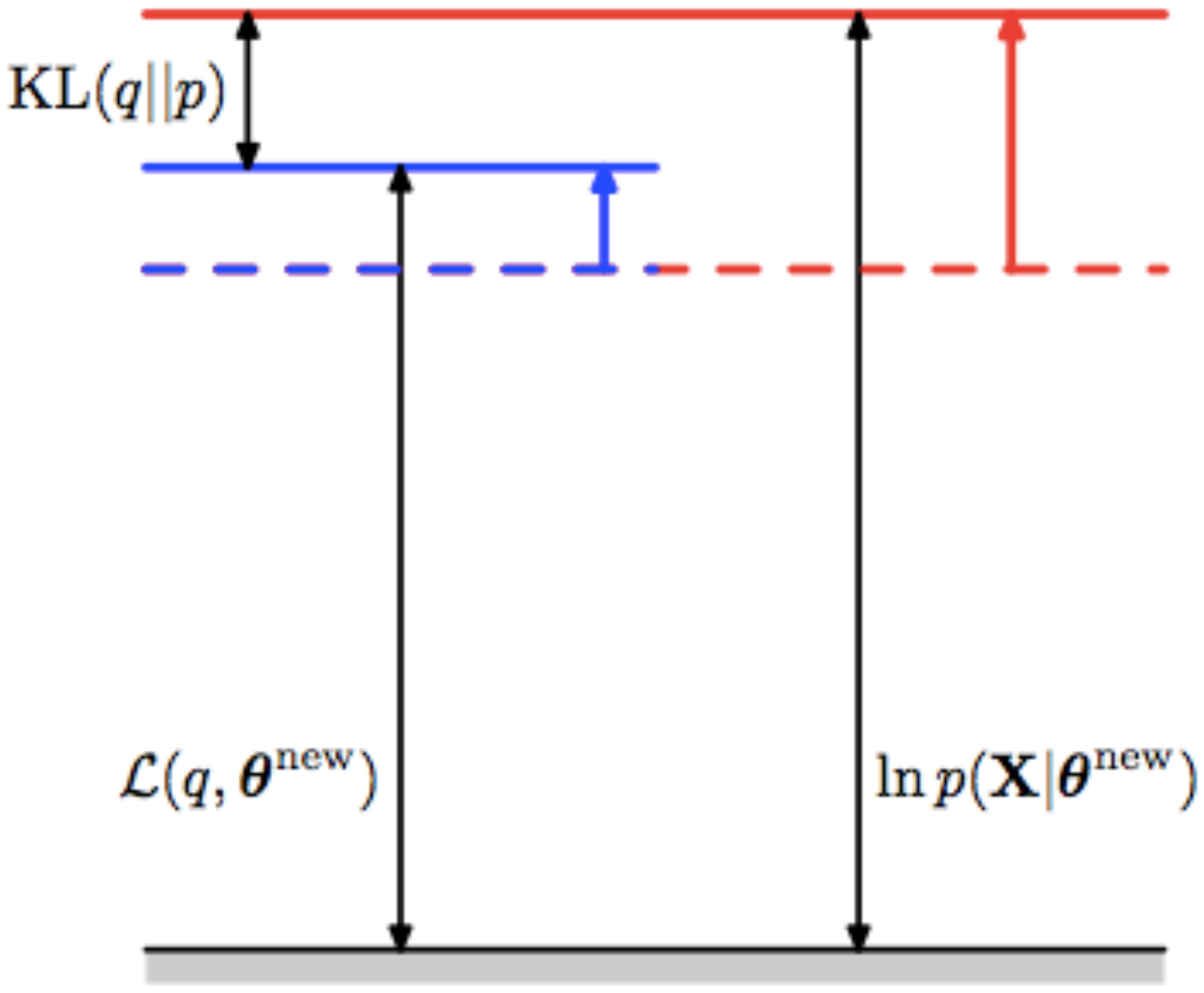
$$q(z) = p(z|x, \theta_{old}),$$

then KL divergence = 0, and thus $\mathcal{L}(q, \theta)$ = log-likelihood at θ_{old}, maximizing the ELBO.

Conditioned on observed data, and θ_{old}, we use q to **conceptually** compute the expectation of the missing data.



$\mathrm{KL}(q||p) = 0$
$\mathcal{L}(q, \boldsymbol{\theta}^{\mathrm{old}})$
$\ln p(\mathbf{X}|\boldsymbol{\theta}^{\mathrm{old}})$


$\mathrm{KL}(q||p)$
$\mathcal{L}(q, \boldsymbol{\theta}^{\mathrm{new}})$
$\ln p(\mathbf{X}|\boldsymbol{\theta}^{\mathrm{new}})$

M-step

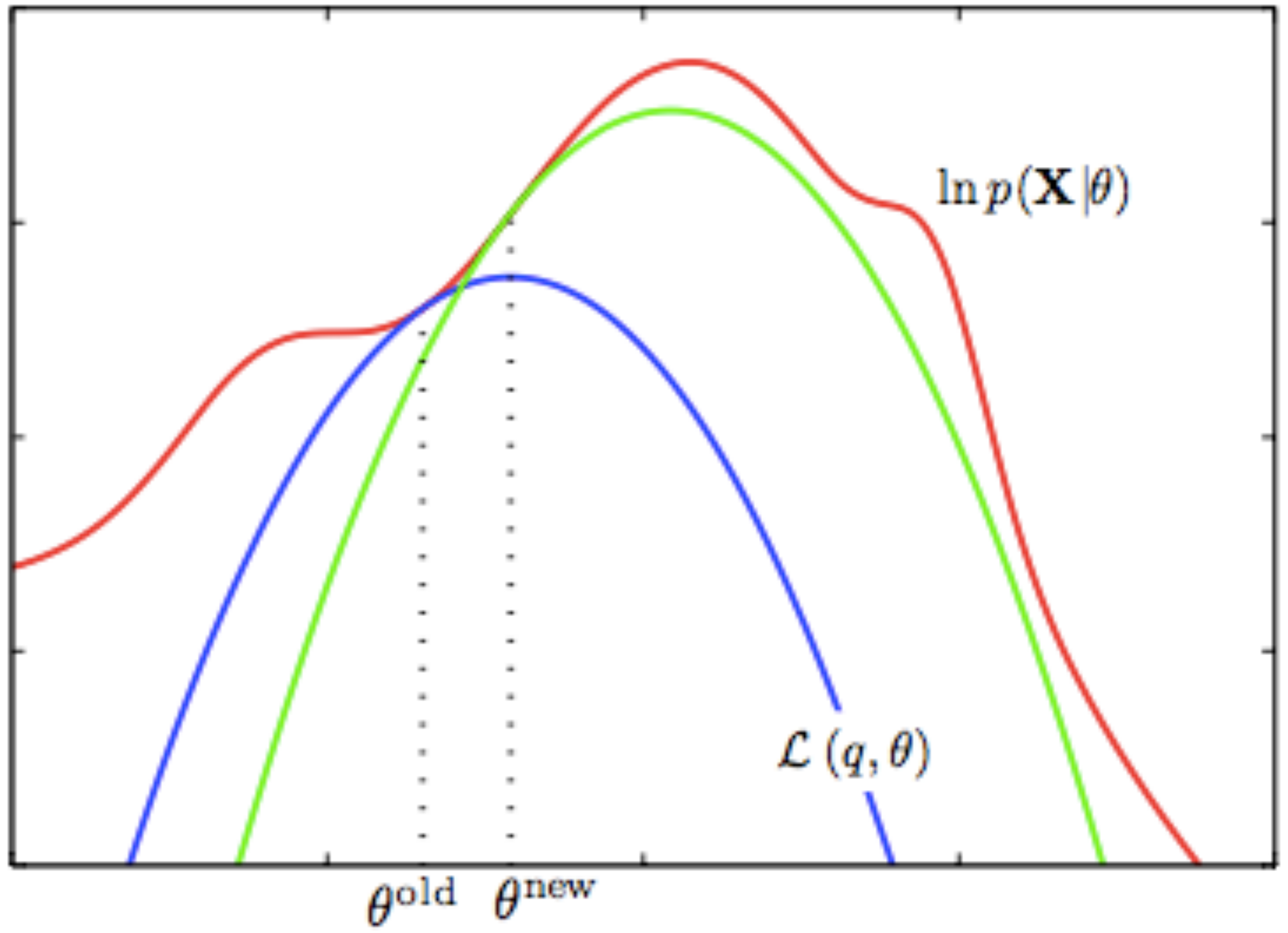
After E-step, ELBO touches $\ell(x|\theta)$, any maximization wrt θ will also “push up” on likelihood, thus increasing it.

Thus hold $q(z)$ fixed at the z-posterior calculated at θ_{old}, and maximize ELBO $\mathcal{L}(q, \theta, \theta_{old})$ or $Q(q, \theta, \theta_{old})$ wrt θ to obtain new θ_{new}.

In general $q(\theta_{old}0 \neq p(z|x, \theta_{new})$, hence KL $\neq 0$. Thus increase in $\ell(x|\theta) \geq$ increase in ELBO.

Process

1. Start with $p(x|\theta)$(red curve), θ_{old}.
2. Until convergence:
    1. E-step: Evaluate $q(z, \theta_{old}) = p(z|x, \theta_{old})$ which gives rise to $Q(\theta, \theta_{old})$ or $ELBO(\theta, \theta_{old})$(blue curve) whose value equals the value of $p(x|\theta)$ at θ_{old}.
    2. M-step: maximize Q or $ELBO$ wrt θ to get θ_{new}.
    3. Set $\theta_{old} = \theta_{new}$



In EM we know $p(z|x, \theta_{old})$, but what if you did not? You know pdf $p(x, z)$, and thus an un-normalized $p(z \mid x)$

When z are the parameters of some posterior, this is the standard bayesian inference problem!!

What is the data size is too large? And our samplers take too long. And MCMC has sampler fidles. We'll need to use:

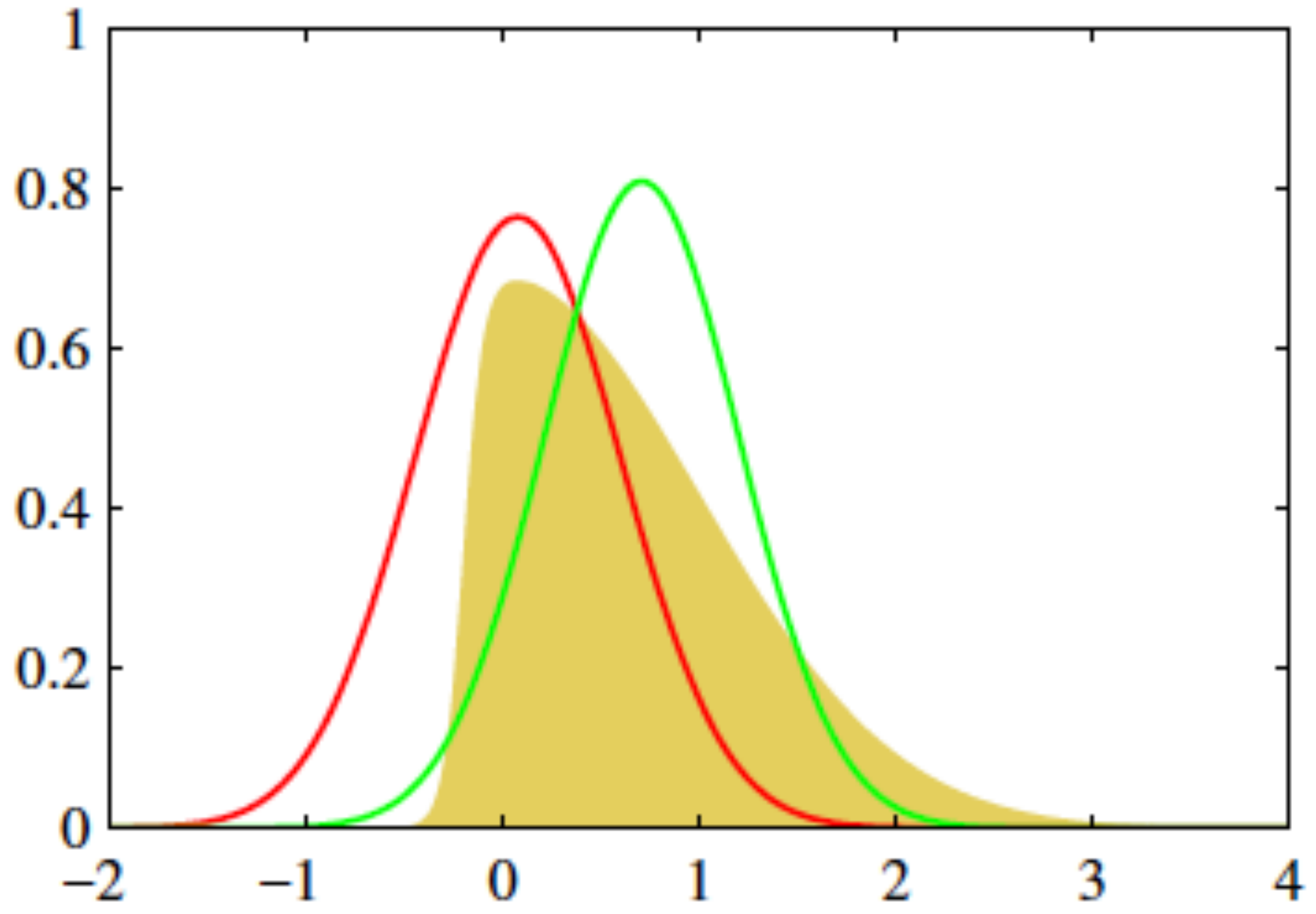
VARIATIONAL INFERENCE

Core Idea

z is now all parameters. Dont distinguish from θ.

Restricting to a family of approximate distributions D over z, find a member of that family that minimizes the KL divergence to the exact posterior. An optimization problem over functions:

$$q^*(z) = \underset{q(z)\in D}{\arg\min} \quad KL(q(z)||p(z|x))$$



VI vs MCMC

MCMC	**VI**
More computationally intensive	Less intensive
Guarantees producing asymptotically exact samples from target distribution	No such guarantees
Slower	Faster, especially for large data sets and complex distributions
Best for precise inference	Useful to explore many scenarios quickly or large data sets

Mean Field: Find a q such that:

$KL + ELBO = log(p(x))$: KL minimized means ELBO maximized.

Choose a "mean-field" q such that:

$$q(z) = \prod_{j=1}^{m} q_j(z_j)$$

Each individual latent factor can take on any paramteric form corresponding to the latent variable.

Optimization: CAVI

Coordinate ascent mean-field variational inference

maximizes ELBO by iteratively optimizing each variational factor of the mean-field variational distribution, while holding the others fixed.

Define Complete Conditional of $z_j = p(z_j | \boldsymbol{z_{-j}}, \boldsymbol{x})$

Algorithm

Input: $p(x, z)$ with data set x, **Output**: $q(z) = \prod_j q_j(z_j)$

Initialize: $q_j(z_j)$

```
while ELBO has not converged (or z have not converged):`
    for each j:

$$q_j \propto exp(E_{-j}[logp(z_j|z_{-j}, x])$$

    compute ELBO
```

where the expectations above are with respect to the variational distribution over $\boldsymbol{z_{-j}}$:

$$\prod_{\boldsymbol{l \neq j}} q_l(z_l)$$

Assertion: $q_j^*(z_j) \propto \exp\{E_{-j}[log(p(z_j|\boldsymbol{z_{-j}}, \boldsymbol{x}))]\}$
$\implies q_j^*(z_j) \propto \exp\{E_{-j}[log(p(z_j, \boldsymbol{z_{-j}}, \boldsymbol{x}))]\}$

(because the mean-field family assumes that all the latent variables are independent)

Example: "Fake :-) Gaussian"

```
data = np.random.randn(100)
with pm.Model() as model:
    mu = pm.Normal('mu', mu=0, sd=1)
    sd = pm.HalfNormal('sd', sd=1)
    n = pm.Normal('n', mu=mu, sd=sd, observed=data)
```

Assume Gaussian posteriors for `mu` and `log(sd)`. So, for e.g.,

$$\mu \sim N(\mu_\mu, \sigma_\mu^2), log(\sigma) \sim N(\mu_\sigma, \sigma_\sigma^2)$$

For the second term below, we have only retained what depends on $q_j(z_j)$

$$ELBO(q) = E_q[(log(p(z,x))] - E_q[log(q(z))]$$
$$\implies ELBO(q_j) = E_j[E_{-j}[log(p(z_j, \boldsymbol{z_{-j}}, \boldsymbol{x}))]] - E_j[log(q_j(z_j))] + constants$$
$$\implies ELBO(q_j) = E_j[A] - E_j[log(q_j(z_j))] + constants$$

Upto an added constant, $RHS = -D_{KL}(q_j, exp(A))$. Thus, maximizing $ELBO(q_j)$ same as minimizing KL divergence.

This occurs when $q_j = exp(A)$. Thus CAVI locally maximizes ELBO.

Example of 1D Gaussian

$$p(\mathcal{D}|\mu,\tau) = \left(\frac{\tau}{2\pi}\right)^{N/2} \exp\left\{-\frac{\tau}{2}\sum_{n=1}^{N}(x_n-\mu)^2\right\}.$$

We now introduce conjugate prior distributions for μ and τ given by

$$\begin{aligned} p(\mu|\tau) &= \mathcal{N}\left(\mu|\mu_0,(\lambda_0\tau)^{-1}\right) \\ p(\tau) &= \mathrm{Gam}(\tau|a_0,b_0) \end{aligned}$$

Consider a variational posterior $q(\mu,\tau) = q_\mu(\mu)\,q_\tau(\tau)$.

Using our formulae we have for the approximate μ posterior:

$$\begin{aligned}\ln q^\star_\mu(\mu) &= \mathbb{E}_\tau\left[\ln p(\mathcal{D}|\mu,\tau) + \ln p(\mu|\tau)\right] + \text{const} \\ &= -\frac{\mathbb{E}[\tau]}{2}\left\{\lambda_0(\mu-\mu_0)^2 + \sum_{n=1}^{N}(x_n-\mu)^2\right\} + \text{const.}\end{aligned}$$

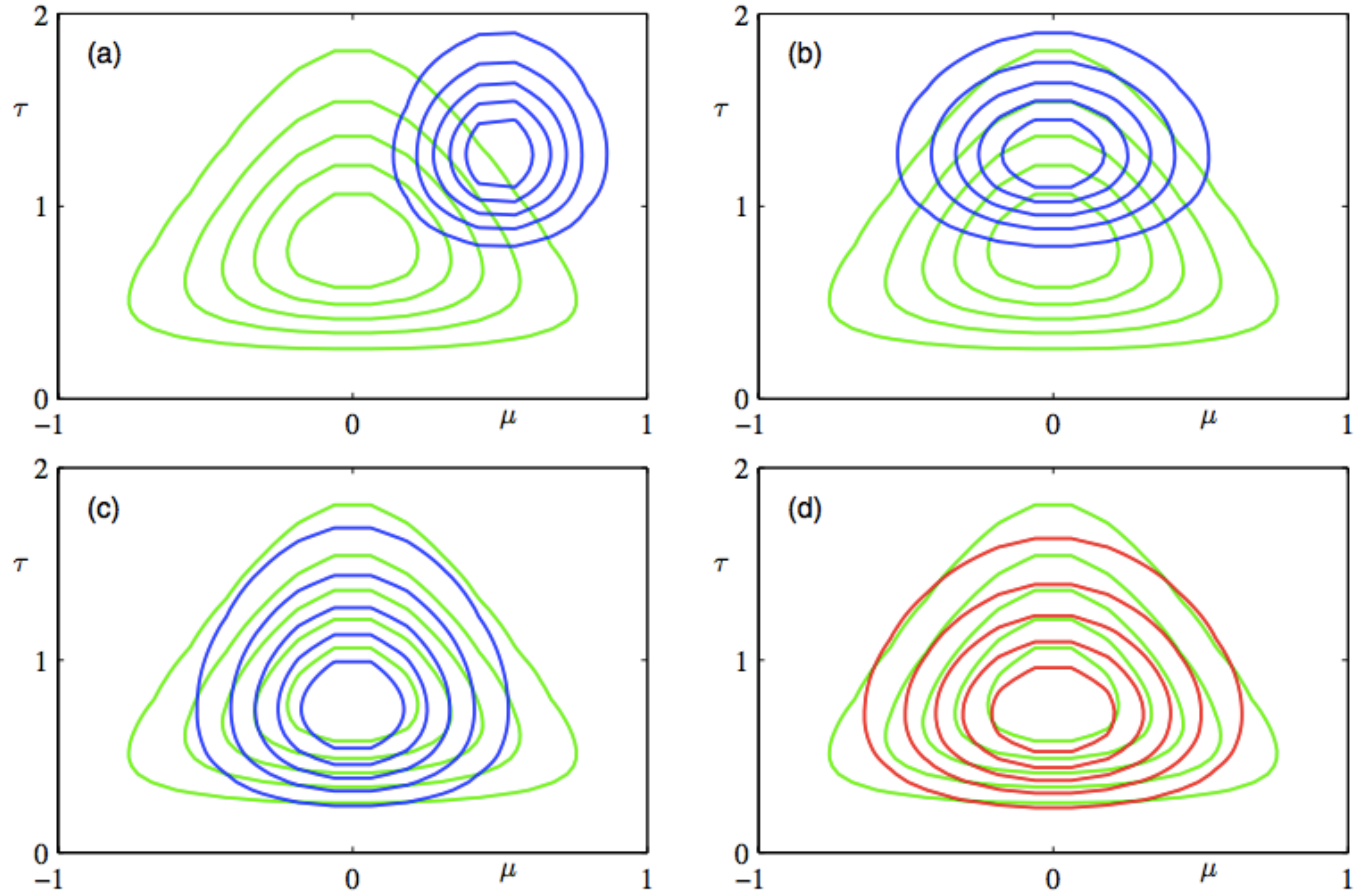
and thus $q_\mu(\mu) \sim N(\mu \mid \mu_N, \lambda_n^{-1})$

with $\mu_N = \dfrac{\lambda_0\mu_0 + N\bar{x}}{\lambda_0 + N}$ and $\lambda_N = (\lambda_0 + N)E_\tau[\tau]$

$$\begin{aligned}
\ln q_\tau^\star(\tau) &= \mathbb{E}_\mu\left[\ln p(\mathcal{D}|\mu,\tau) + \ln p(\mu|\tau)\right] + \ln p(\tau) + \text{const} \\
&= (a_0 - 1)\ln\tau - b_0\tau + \frac{N}{2}\ln\tau \\
&\quad -\frac{\tau}{2}\mathbb{E}_\mu\left[\sum_{n=1}^{N}(x_n-\mu)^2 + \lambda_0(\mu-\mu_0)^2\right] + \text{const}
\end{aligned}$$

with the approximate τ posterior $q_\tau(\tau) \sim Gamma(\tau \mid a_N, b_N)$, and

$$a_N = a_0 + N/2,\;\; b_N = b_0 + \frac{1}{2}E_\mu\left[\sum_{n=1}^{N}(x_n-\mu)^2 + \lambda_0(\mu-\mu_0)^2\right]$$



(a)
(b)
(c)
(d)
τ
μ
2
1
0
−1
0
1

ADVI

Core Idea:

- Use gradient based optimization, do it on less data
- do it automatically

Problem with CAVI

- does not scale
- ELBO must be painstakingly calculated
- optimized with custom CAVI updates for each new model
- If you choose to use a gradient based optimizer then you must supply gradients.

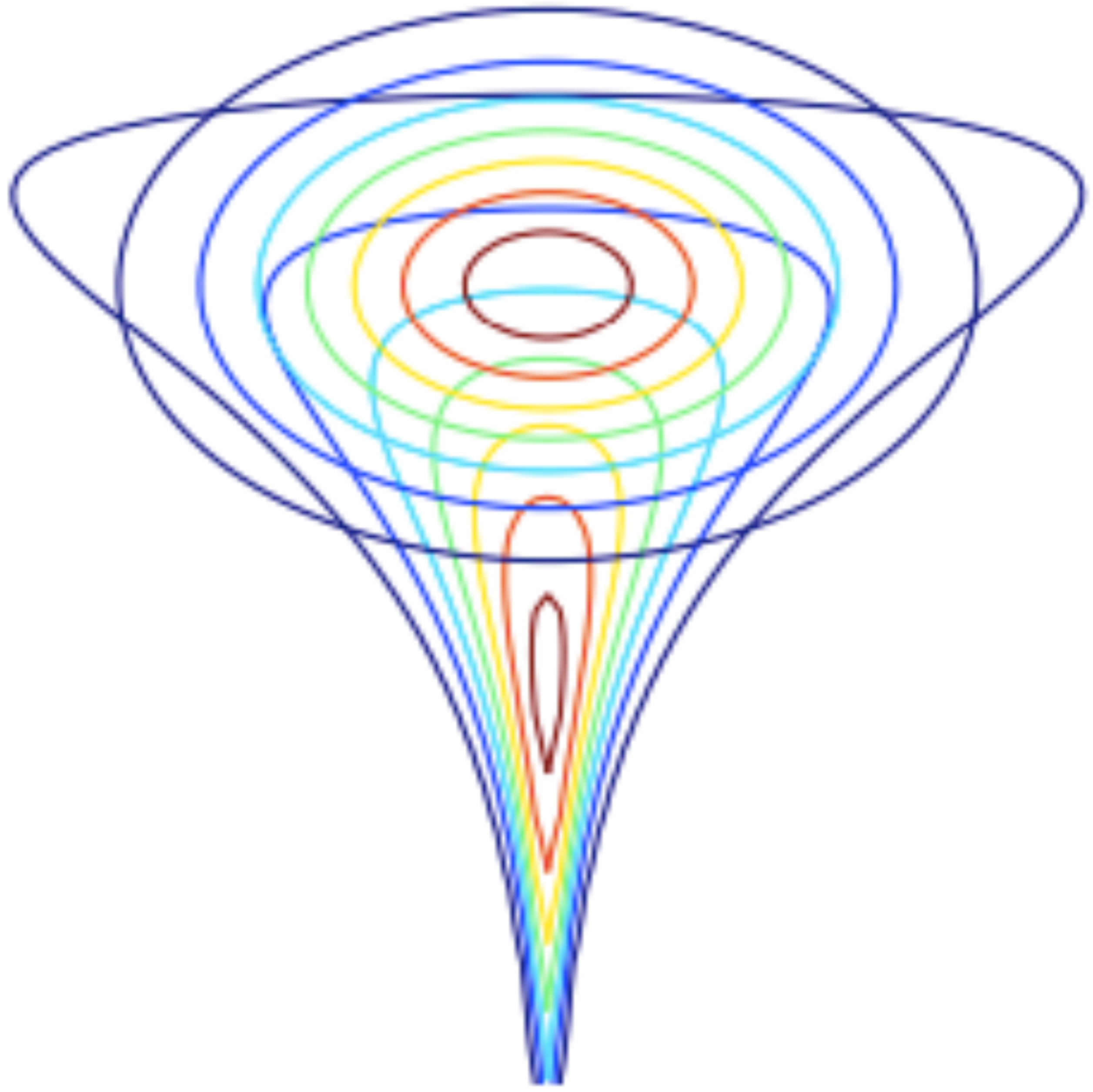
ADVI solves this problem automatically. The user specifies the model, expressed as a program, and ADVI automatically generates a corresponding variational algorithm. The idea is to first automatically transform the inference problem into a common space and then to solve the variational optimization. Solving the problem in this common space solves variational inference for all models in a large class.

-ADVI Paper

Basic Idea: BBVI

```
def lower_bound(variational_params, logprob_func, D, num_samples):
    # variational_params, mean and covariance of q.
    # logprob_func: model unnormalized log-probability.
    # D: number of parameters
    # num_samples: number of Monte Carlo samples.
    mu, cov = variational_params[:D], np.exp(variational_params[D:])
    # Sample from MVN using eparameterization trick.
    samples = npr.randn(num_samples, D) * np.sqrt(cov) + mu
    # ELBO = exact entropy plus Monte Carlo estimate of energy.
    return mvn.entropy(mu, np.diag(cov)) + np.mean(logprob(samples))
    # Gradient wrt variational params using autograd.
    gradient_func = grad(lower_bound)
    #then use Adam, or RMSpropo SGD
```

```
def log_density(x):
    # An example unnormalized 2D density
    mu, log_sigma = x[:, 0], x[:, 1]
    sigma_density = norm.logpdf(log_sigma, 0, 1.35)
    mu_density = norm.logpdf(mu, 0, np.exp(log_sigma))
    return sigma_density + mu_density
```



from Duvenaud and Adams

What does ADVI do?

1. Transformation of latent parameters
2. Standardization transform for posterior to push gradient inside expectation
3. Monte-Carlo estimate of expectation
4. Hill-climb using automatic differentiation

Start with: $\mathcal{L} = E_q[logp(x, \theta)] + H(q)$

where q is a function of θ, parametrized by variational params ϕ:

$$q(\theta; \phi).$$

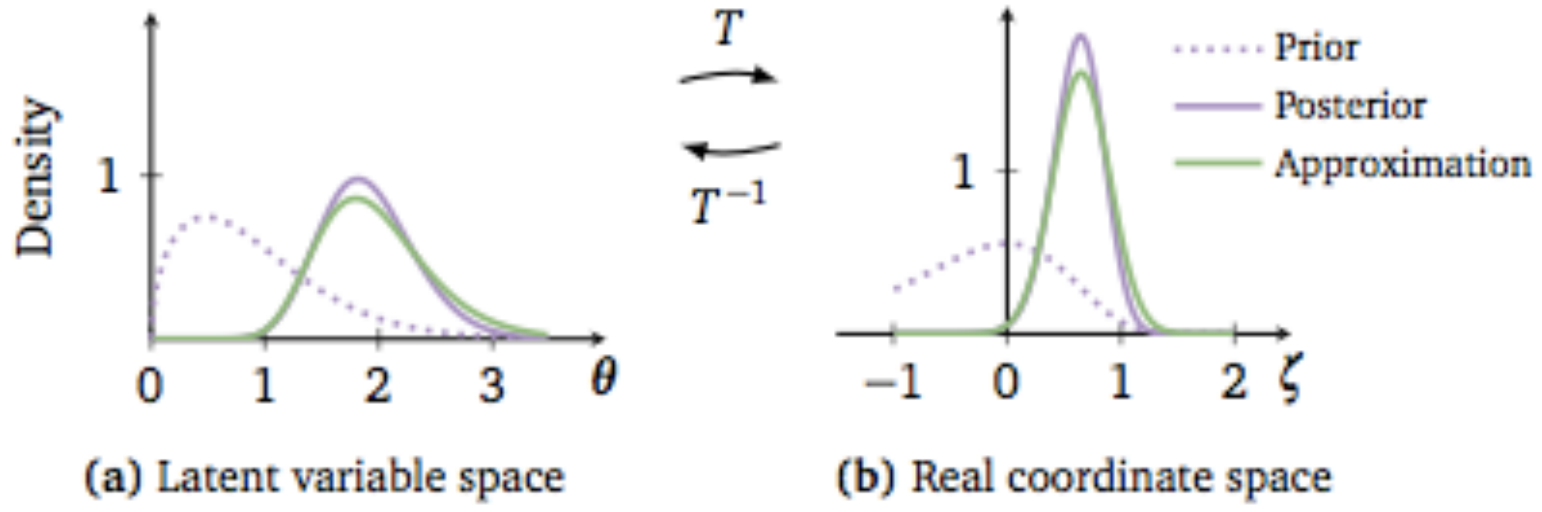
We'll choose a family, say normals as the distribution for each postrior parameter θ. Then ϕ is all the means and standard deviations of the normals. But before that we must ensure all θ live on the real line, If they dont, we'll transform or marginalize.

(1) T-Transformation

- Latent parameters are transformed to representations where the 'new" parameters are unconstrained on the real-line. Specifically the joint $p(x, \theta)$ transforms to $p(x, \zeta)$ where ζ is un-constrained.
- This is done for *ALL* latent variables. Thus use the same variational family for ALL parameters, and indeed for ALL models

$$\mathcal{L} = E_{\zeta}[logp(x, T^{-1}(\zeta)) + log(det(J_{T^{-1}}(\zeta)))] + H(q)$$

- Discrete parameters must be marginalized out.
- Optimizing the KL-divergence implicitly assumes that the support of the approximating density lies within the support of the posterior. These transformations make sure that this is the case
- First choose as our family of approximating densities mean-field normal distributions. We'll transform the always positive σ params by simply taking their logs.



(2) S-transformation (a function of ϕ)

- we must maximize our suitably transformed ELBO, so want to autodiff wrt ϕ.
- we are optimizing an expectation value with respect to the transformed approximate posterior. This posterior contains our transformed latent parameters so the gradient of this expectation is not simply defined.
- we want to push the gradient inside the expectation. For this, the distribution we use to calculate the expectation must be free of parameters

Another transformation takes the approximate 1-D gaussian q and standardizes it. The determinant of the jacobian of this transform is 1.

(3) Calculate the gradients to maximize the ELBO and

Standardize the ζ by the means and standard deviations (choleskies) of the q approximations, to get η.

$$\mathcal{L} = E_{N(0,I)}[logp(x, T^{-1}(S_{\phi}^{-1}(\eta))) + log(det(J_{T^{-1}}(S_{\phi}^{-1}(\eta))))] + H(q)$$

We dont bother to standardize the entropy, as the entropy of the gaussian is a well known beast and we can compute it.

Now we can move the gradient inside the expectation (integral) to boot. This means that our job now becomes the calculation of the gradient of the full-data joint.

(4) Calculate the gradients and compute the expectation

As a result of this, we can now compute the expectation as a monte-carlo estimate over a standard Gaussian--superfast!

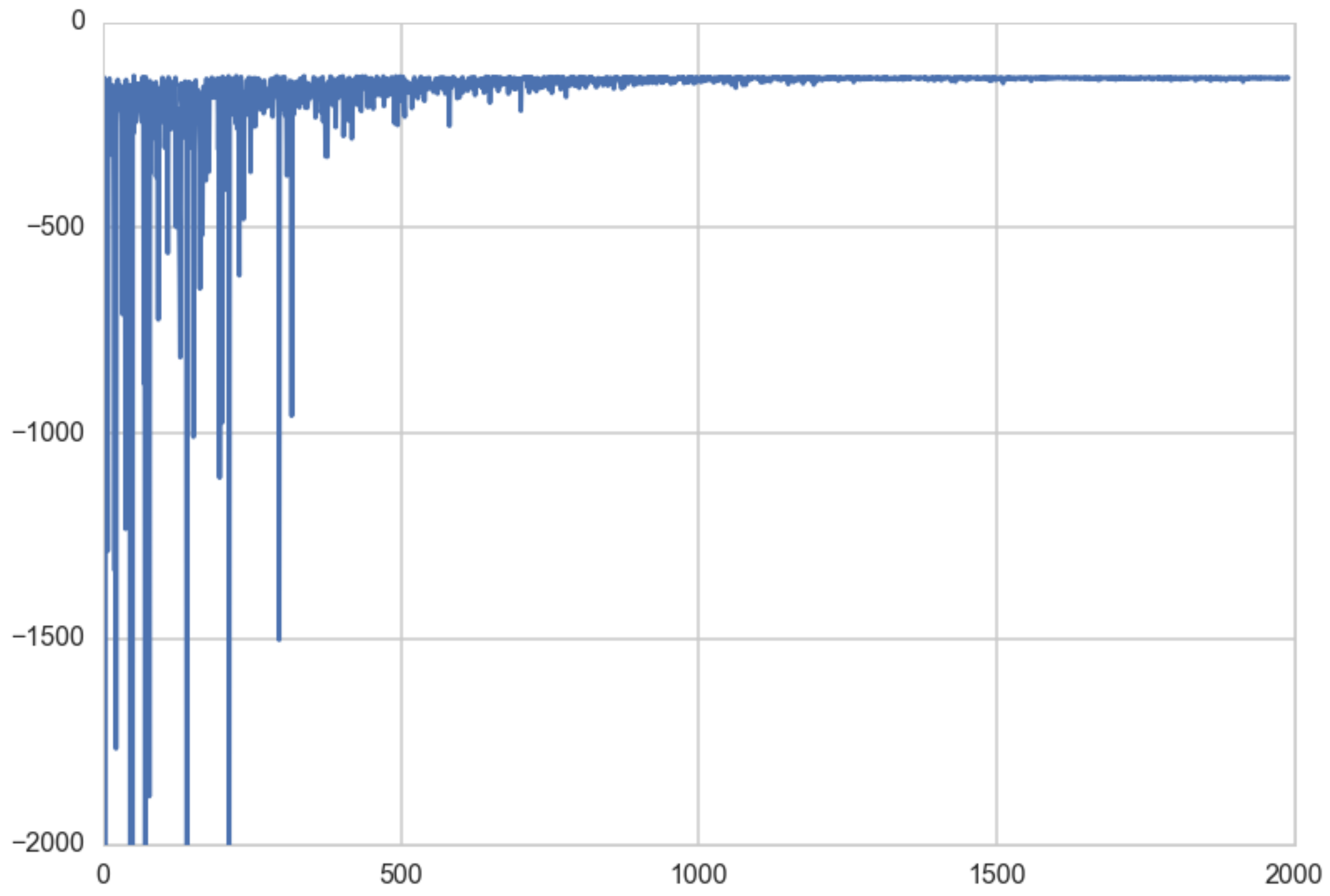
We can replace full x data by just one point (SGD) or mini-batch (some-x) and thus use noisy gradients to optimize the variational distribution.

An adaptively tuned step-size is used to provide good convergence.

ADVI in pymc3

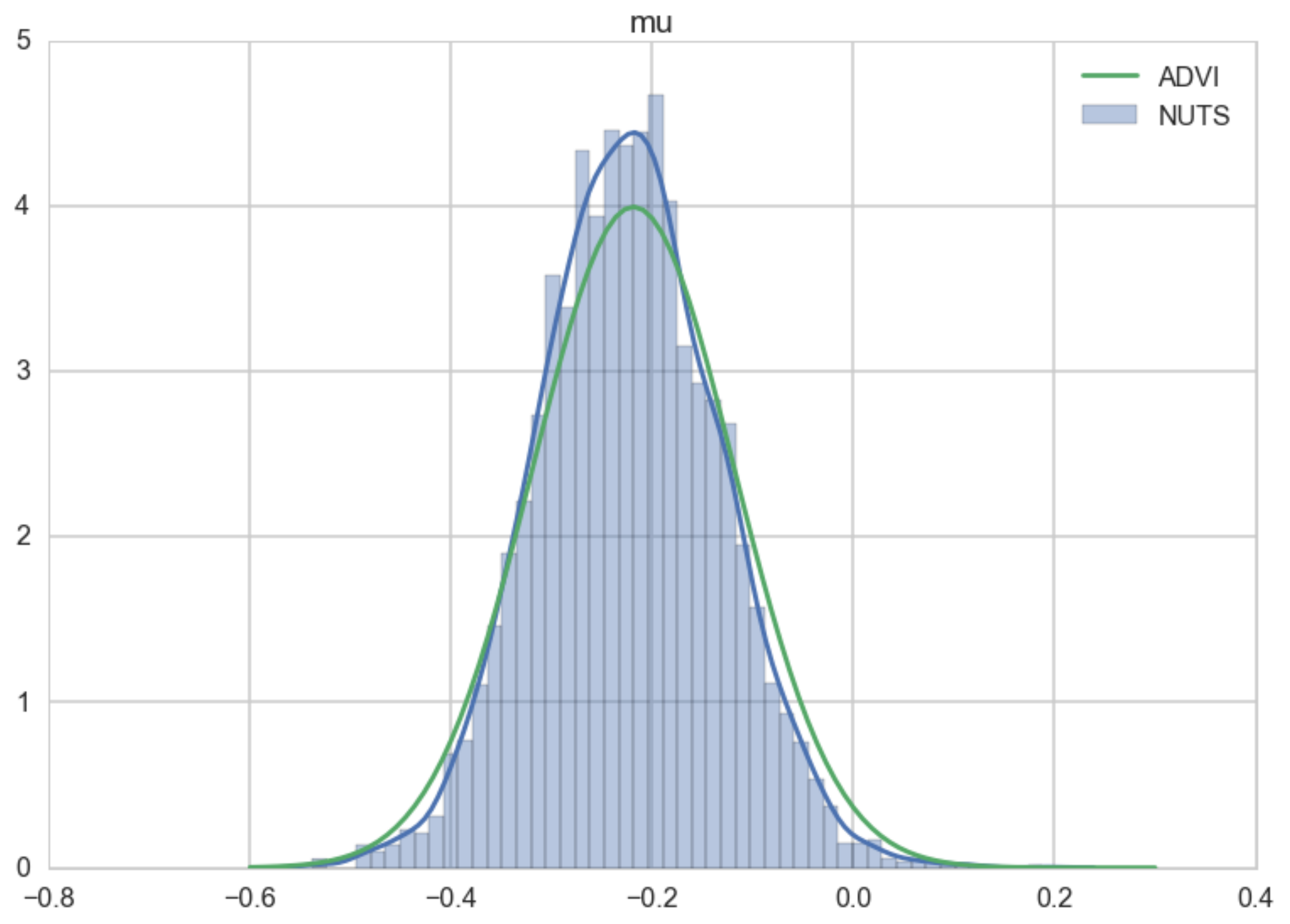
```
data = np.random.randn(100)
with pm.Model() as model:
    mu = pm.Normal('mu', mu=0, sd=1, testval=0)
    sd = pm.HalfNormal('sd', sd=1)
    n = pm.Normal('n', mu=mu, sd=sd, observed=data)
advifit = pm.ADVI( model=model)
advifit.fit(n=50000)
elbo = -advifit.hist
plt.plot(elbo[::10]);
```

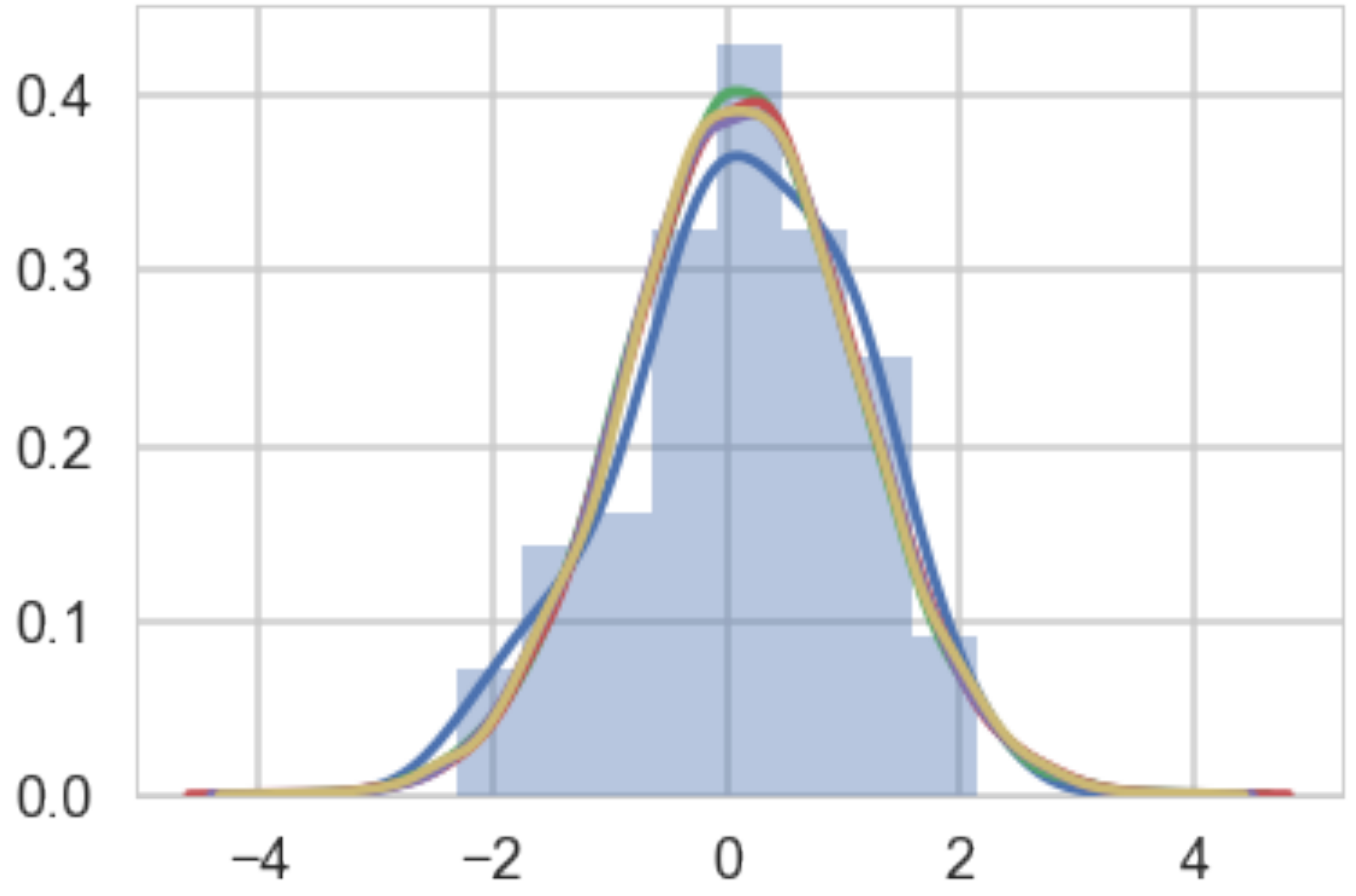
Elbo:



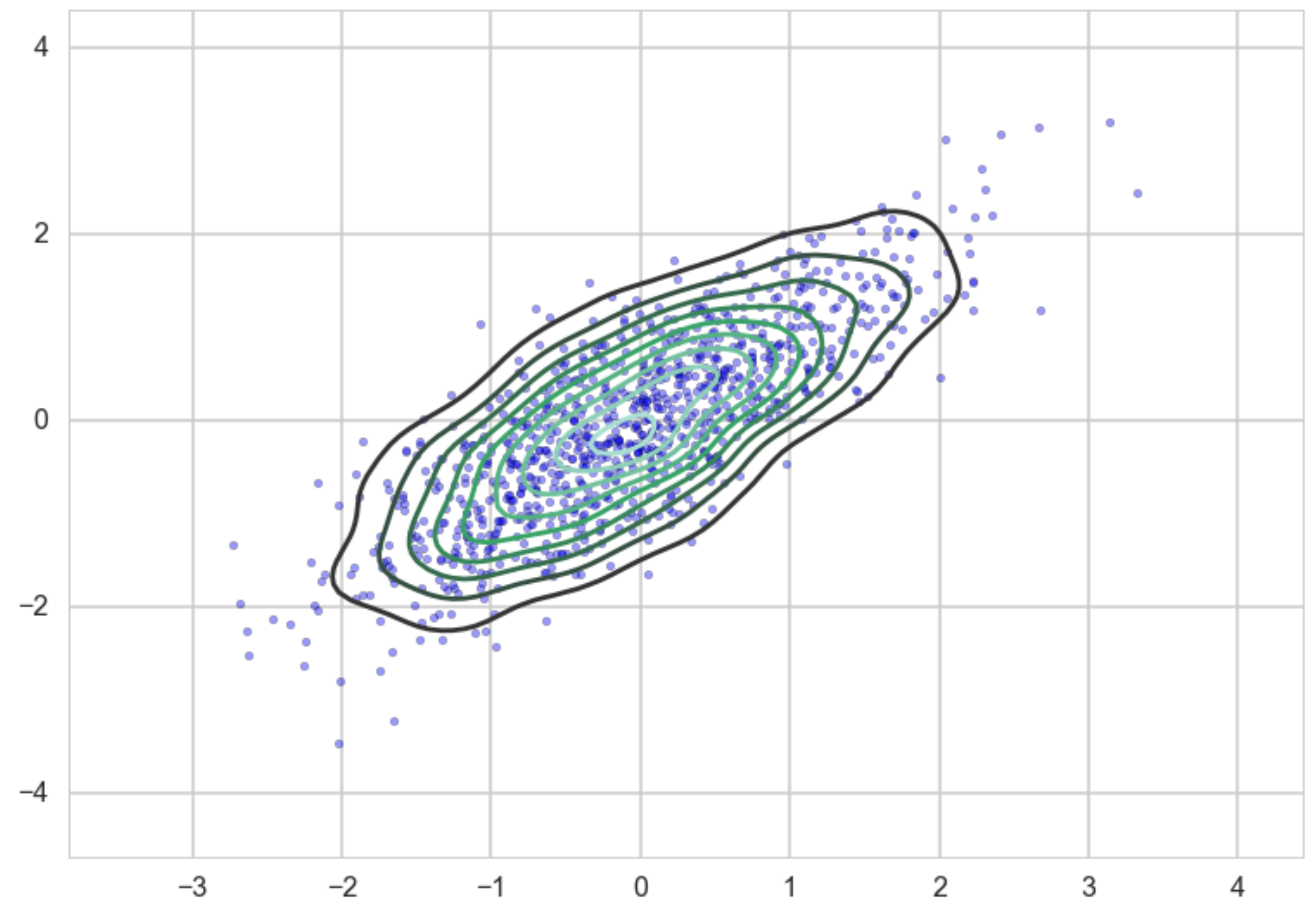
```
trace = advifit.approx.sample(10000)
pred = pm.sample_ppc(trace, 10000, model=model)
sns.distplot(trace_nuts['mu'], label='NUTS')
sns.kdeplot(trace['mu'], label='ADVI')
sns.distplot(data)
sns.kdeplot(pred['n'][:,0])
...
```

left, parameters; right, pp





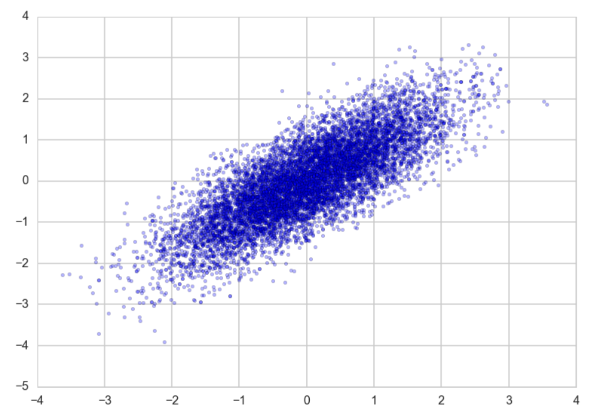
ADVI problems: 2D gaussian example



High correlation gaussian with sampler

```
cov=np.array([[0,0.8],[0.8,0]], dtype=np.float64)
data = np.random.multivariate_normal([0,0], cov, size=1000)
sns.kdeplot(data);
with pm.Model() as mdensity:
    density = pm.MvNormal('density', mu=[0,0],
    cov=tt.fill_diagonal(cov,1), shape=2)
with mdensity:
    mdtrace=pm.sample(10000)
```

Trace:



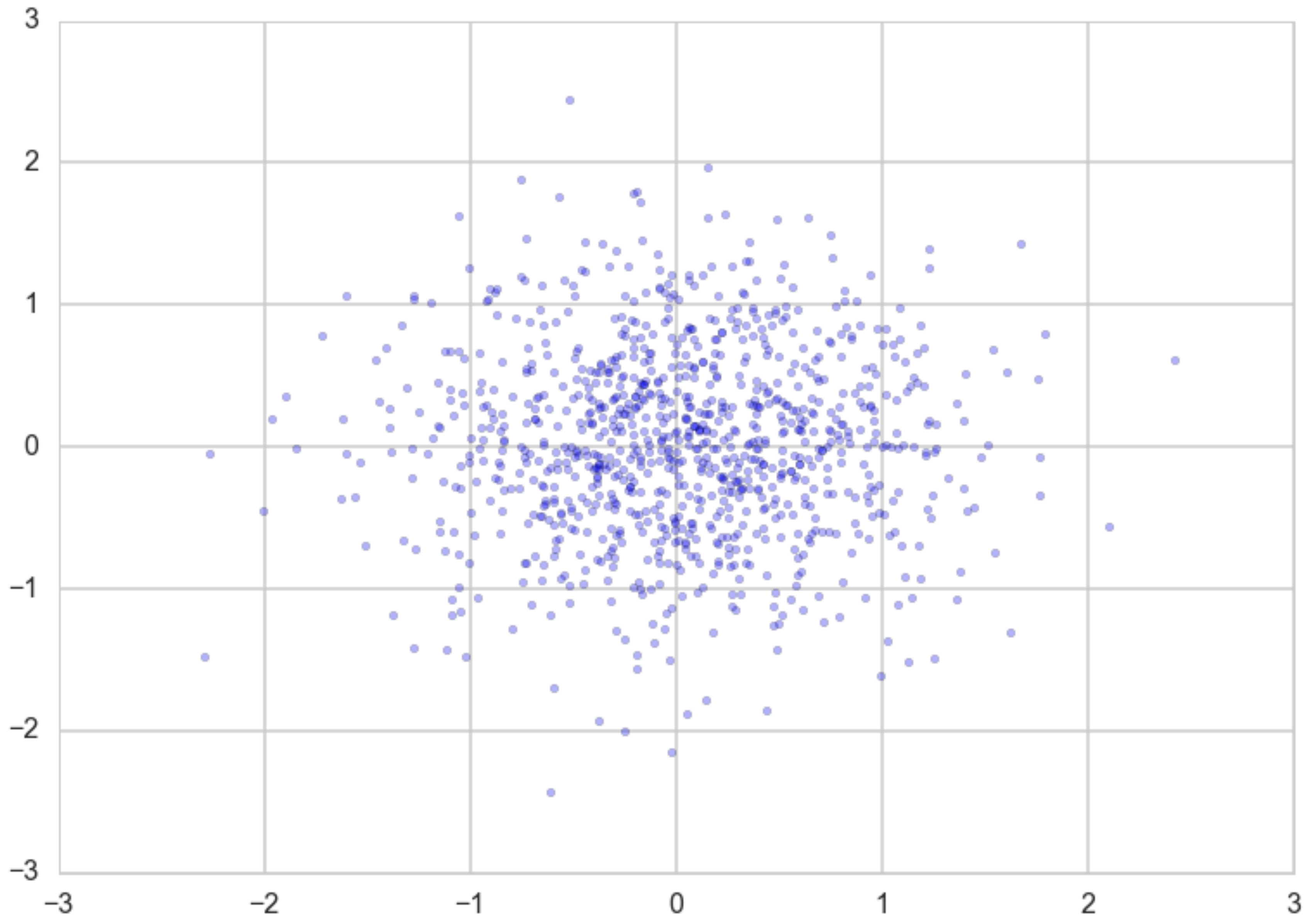
Sampling with ADVI

```
mdvar = pm.ADVI(model=mdensity)
mdvar.fit(n=40000)
samps=mdvar.approx.sample(5000)
plt.scatter(samps['density'][:,0],
    samps['density'][:,1], s=5, alpha=0.3)
```

ADVI cannot find the correlational structure.

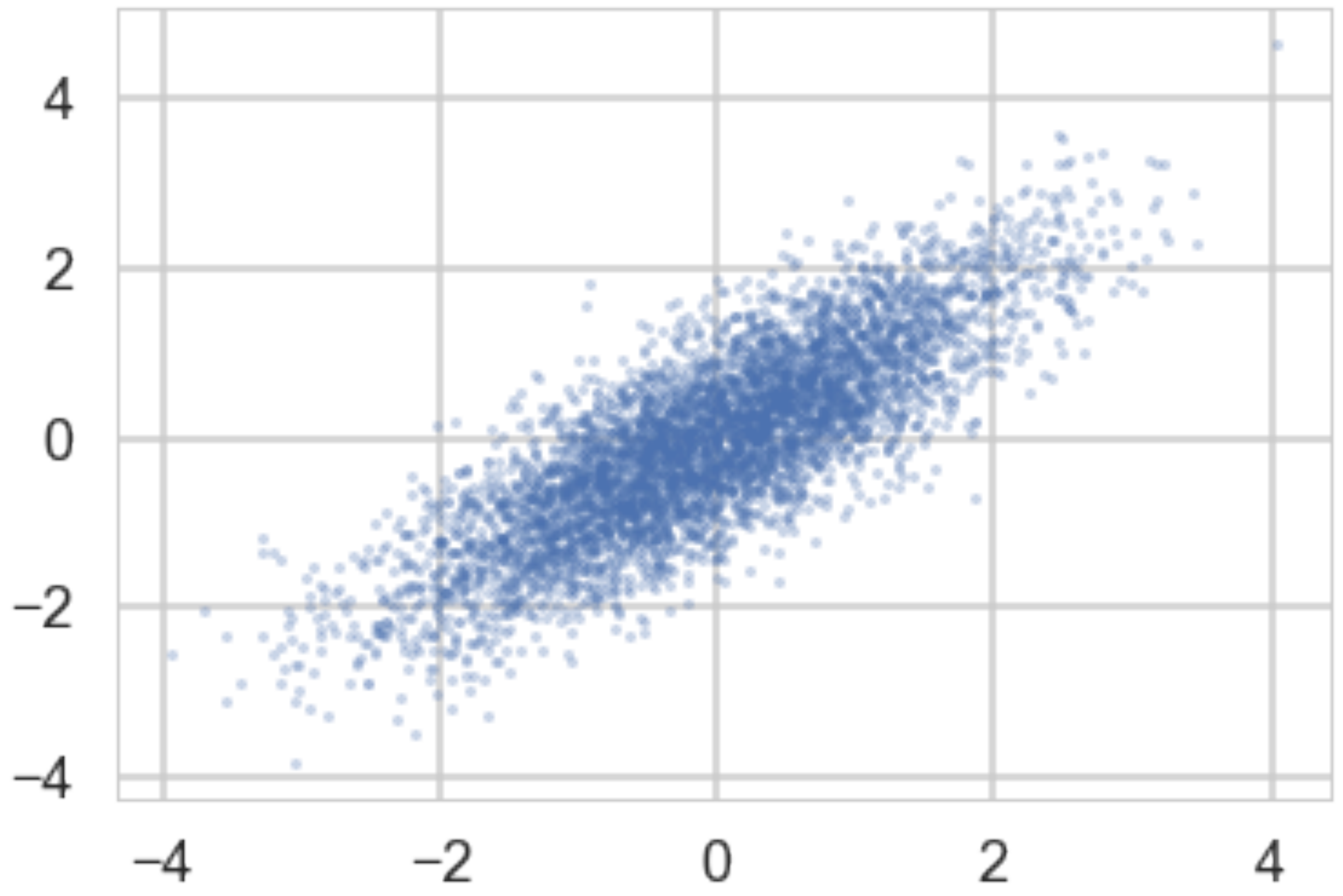
Transform to de-correlate to use ADVI.

You have been doing this for NUTS anyways. Or use Full Rank.



ADVI Full Rank

```
mdvar_fr = pm.FullRankADVI(model=mdensity)
```



Relaxing the mean-field approximation

- Full-Rank ADVI: model covariance
- Normalizing Flows
- Operator Variational Inference: allows generalization of many algorithms under one umbrella

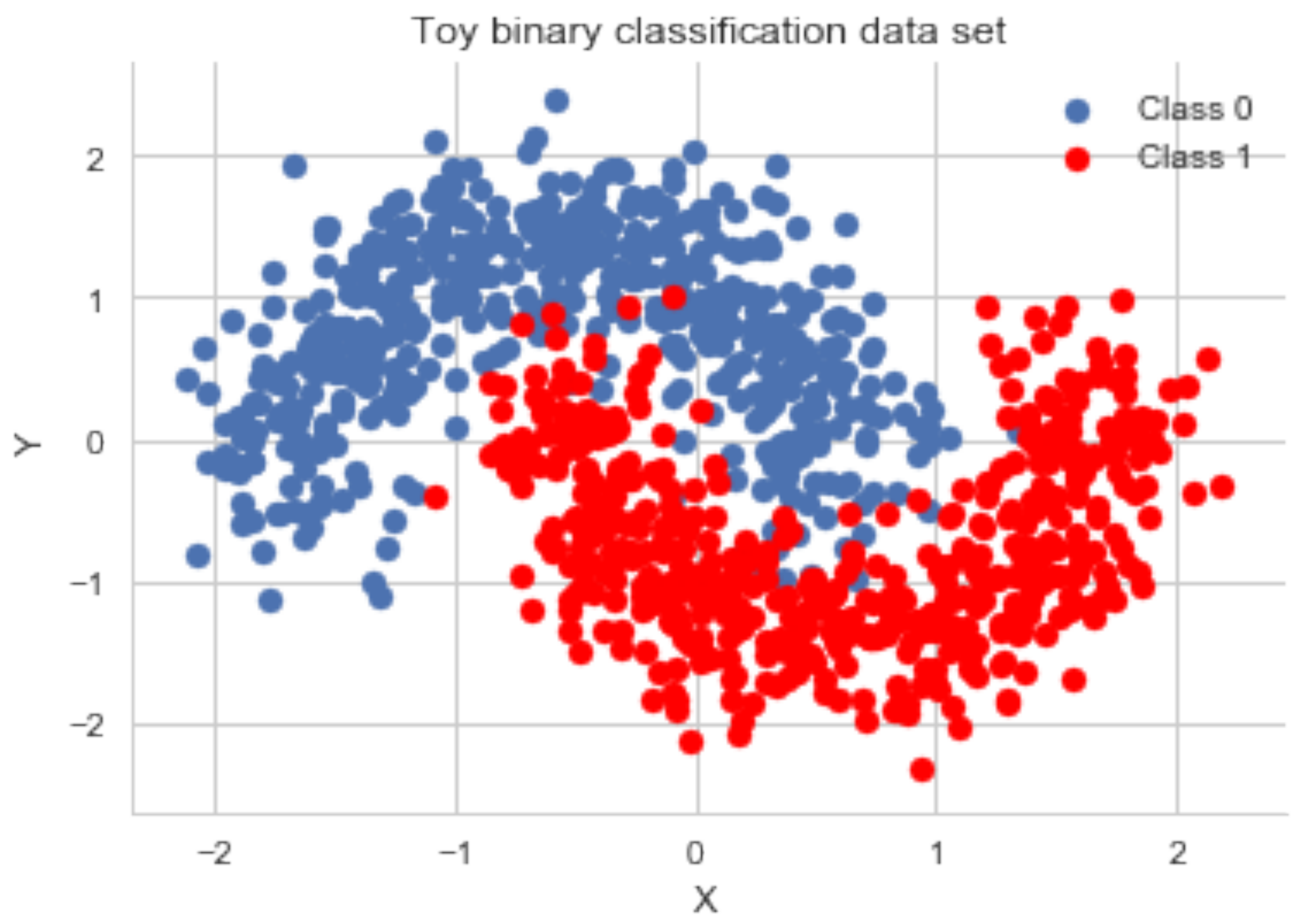
(all implemented in pymc3)

Why use VB

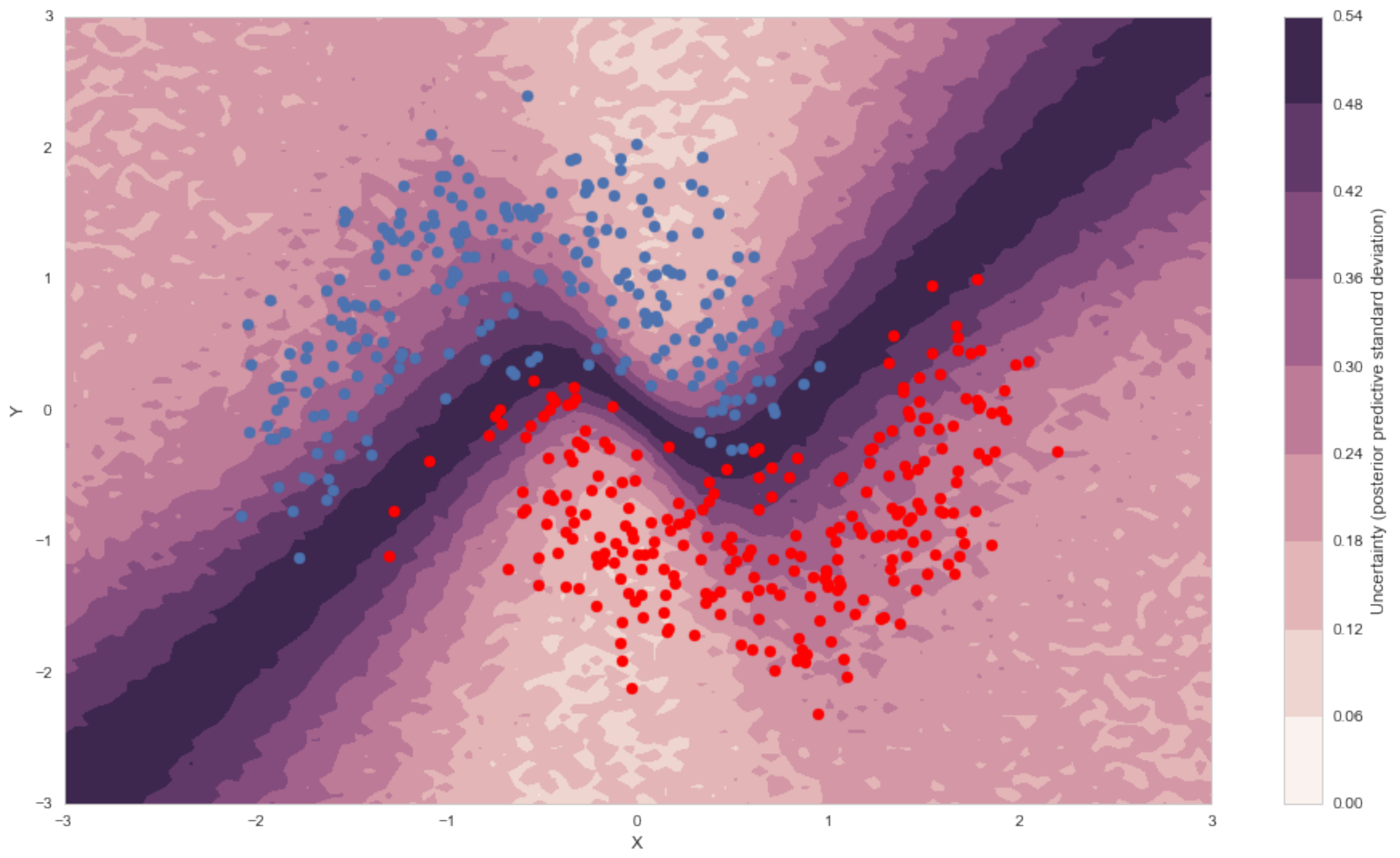
- simply not possible to do inference in large models
- inference in neural networks: understanding robustness, etc
- hierarchical neural networks
- Mixture density networks: mixture parameters are fitted using ANNs
- extension to generative semisupervised learning
- variational autoencoders

Bayesian Neural Network in pymc3

```
def construct_nn(ann_input, ann_output):
    n_hidden = 5
    # Initialize random weights between each layer
    init_1 = np.random.randn(X.shape[1], n_hidden)
    init_2 = np.random.randn(n_hidden, n_hidden)
    init_out = np.random.randn(n_hidden)
    with pm.Model() as neural_network:
        # Weights from input to hidden layer
        weights_in_1 = pm.Normal('w_in_1', 0, sd=1,
                                 shape=(X.shape[1], n_hidden),
                                   testval=init_1)
        # Weights from 1st to 2nd layer
        weights_1_2 = pm.Normal('w_1_2', 0, sd=1,
                                 shape=(n_hidden, n_hidden),
                                   testval=init_2)
        # Weights from hidden layer to output
        weights_2_out = pm.Normal('w_2_out', 0, sd=1,
                                   shape=(n_hidden,),
                                     testval=init_out)
        # Build neural-network using tanh activation function
        act_1 = pm.math.tanh(pm.math.dot(ann_input,
                                         weights_in_1))
        act_2 = pm.math.tanh(pm.math.dot(act_1,
                                         weights_1_2))
        act_out = pm.math.sigmoid(pm.math.dot(act_2,
                                              weights_2_out))
        # Binary classification -> Bernoulli likelihood
        out = pm.Bernoulli('out',
                           act_out,
                           observed=ann_output,
                           total_size=Y_train.shape[0] # IMPORTANT for minibatches
                          )
    return neural_network
```



Fitting with uncertainty



```
ann_input = theano.shared(X_train)
ann_output = theano.shared(Y_train)
neural_network = construct_nn(ann_input, ann_output)
with neural_network:
    nutstrace = pm.sample(2000, tune=1000)
```

232+120 divergences, bad accepance probability, high R^2, 5 mins.

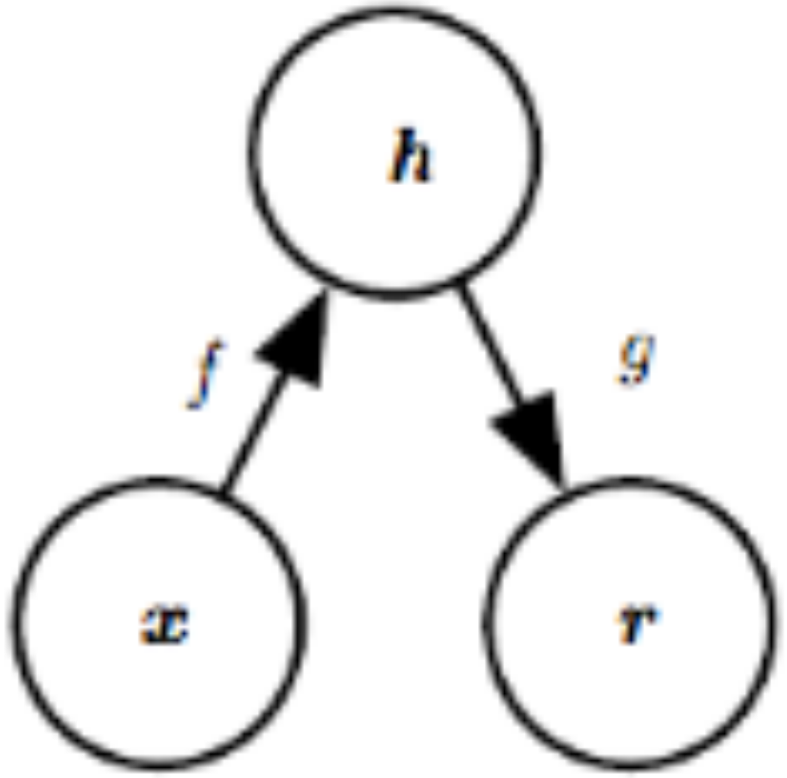
```
with neural_network:
    inference = pm.ADVI()
    approx = pm.fit(n=30000, method=inference)
```

28 seconds.

- we have not talked about learning θ, the parametrization of $p_\theta(x, z)$.
- also, what if there are data-point specific parameters in out model. In other words we have a ϕ_i, where i indexes the data-points
- an example of this is topic modeling or generative image modeling
- we can do this by fitting a ϕ_i as a regression model on the x_i. And we can make this model non-linear by considering an ANN!

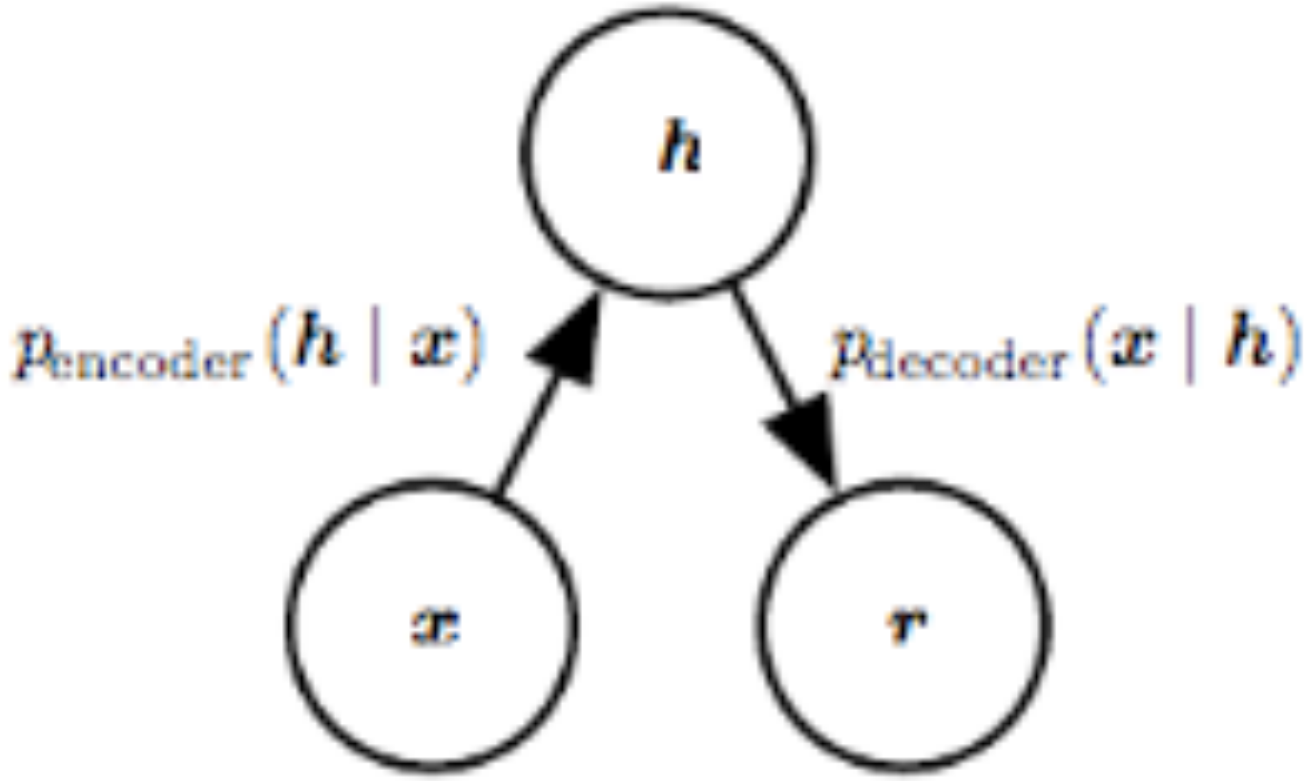
Variational Autoencoders

Autoencoders: basic idea



- **h** is the representation. An *undercomplete* autoencoder makes **h** of smaller dimension than x
- f is the encoder and g the decoder
- simplest idea: minimize $L(x, g(f(x)))$

- can think of an autoencoder as a way of approximately training a generative model.
- the features of the autoencoder describe the latent variables that explain the input
- can go deep!
- generalize to a stochastic autoencoder. The standard autoencoder then is a specific hidden state h or z


$\boldsymbol{h}$
$p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$
$p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$
$\boldsymbol{x}$
$\boldsymbol{r}$

Variational Autoencoder

- just as in ADVI, we want to learn an approximate "encoding posterior" $p(z|x)$
- note that we have now again gone back to thinking of z as a (possibly) deep latent variable, or "representation".

We know how to do this:

ELBO maximization

Basic Setup in VAE

$KL + ELBO = log(p(x))$: ELBO bounds log(evidence)

$$ELBO(q) = E_q[log\,\frac{p(z,x)}{q(z)}] = E_q[log\,\frac{p(x|z)p(z)}{q(z)}] = E_q[log\,p(x|z)] + E_q[log\,\frac{p(z)}{q(z)}]$$

$$\implies ELBO(q) = E_{q(z)}[(log(p(x|z))] - KL(q(z)||p(z))$$

(likelihood-prior balance)

From `edwardlib`: $p(\mathbf{x} \mid \mathbf{z})$

describes how any data $\mathbf{x}$ depend on the latent variables $\mathbf{z}$.

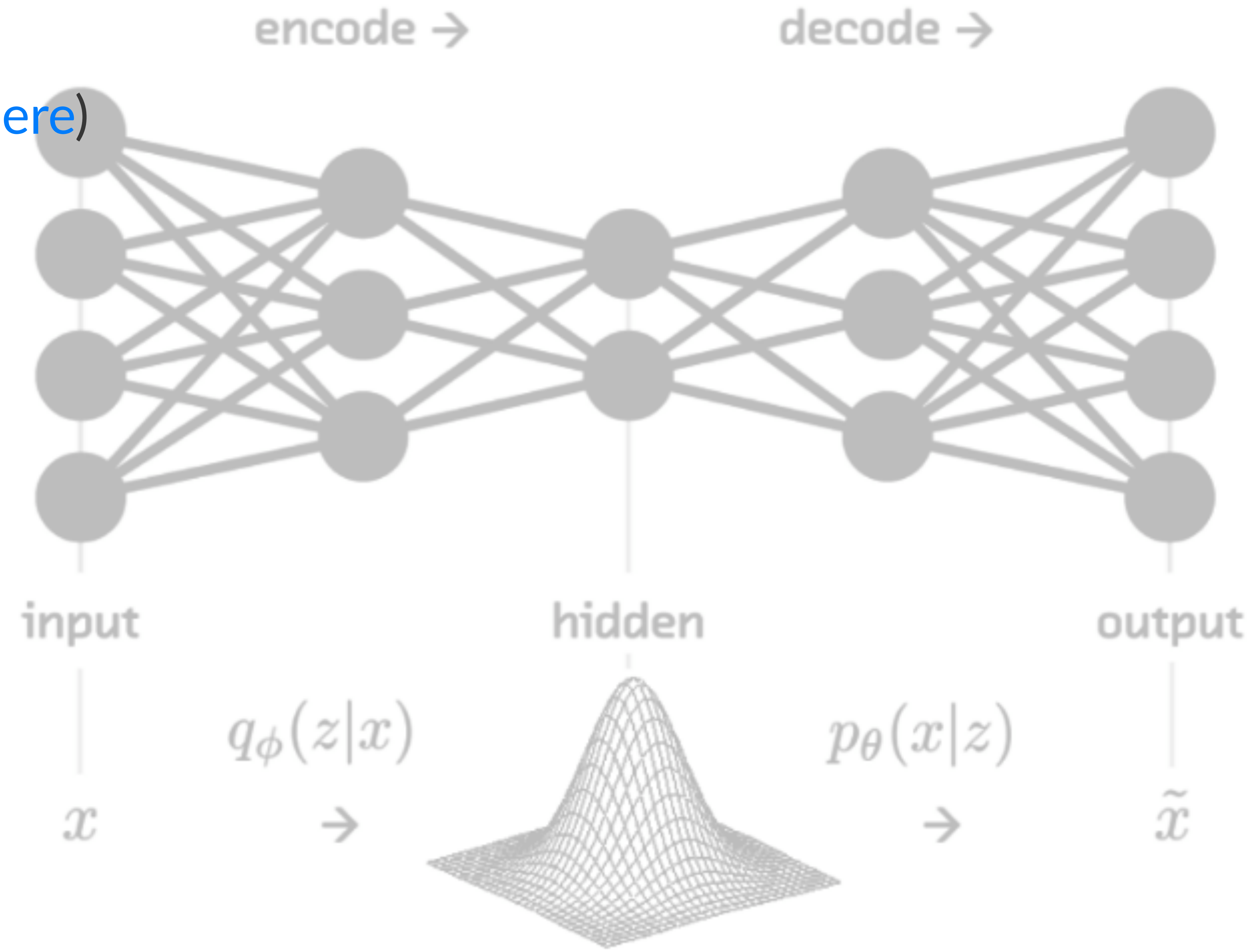
- **The likelihood posits a data generating process**, where the data $\mathbf{x}$ are assumed drawn from the likelihood conditioned on a particular hidden pattern described by $\mathbf{z}$.
- The *prior* $p(\mathbf{z})$ is a probability distribution that describes the latent variables present in the data. **The prior posits a generating process of the hidden structure**.

The Game

$$ELBO(q) = E_{q(z|x)}[(log(p(x|z))] - KL(q(z|x)||p(z))$$

- get z samples coming for fixed x, $q(z|x)$- to be close to some prior, $p(z)$, typically chosen as an isotropic gaussian...the regularization term
- first term is called "reconstruction loss", or "capacity of model to generate something like the data".

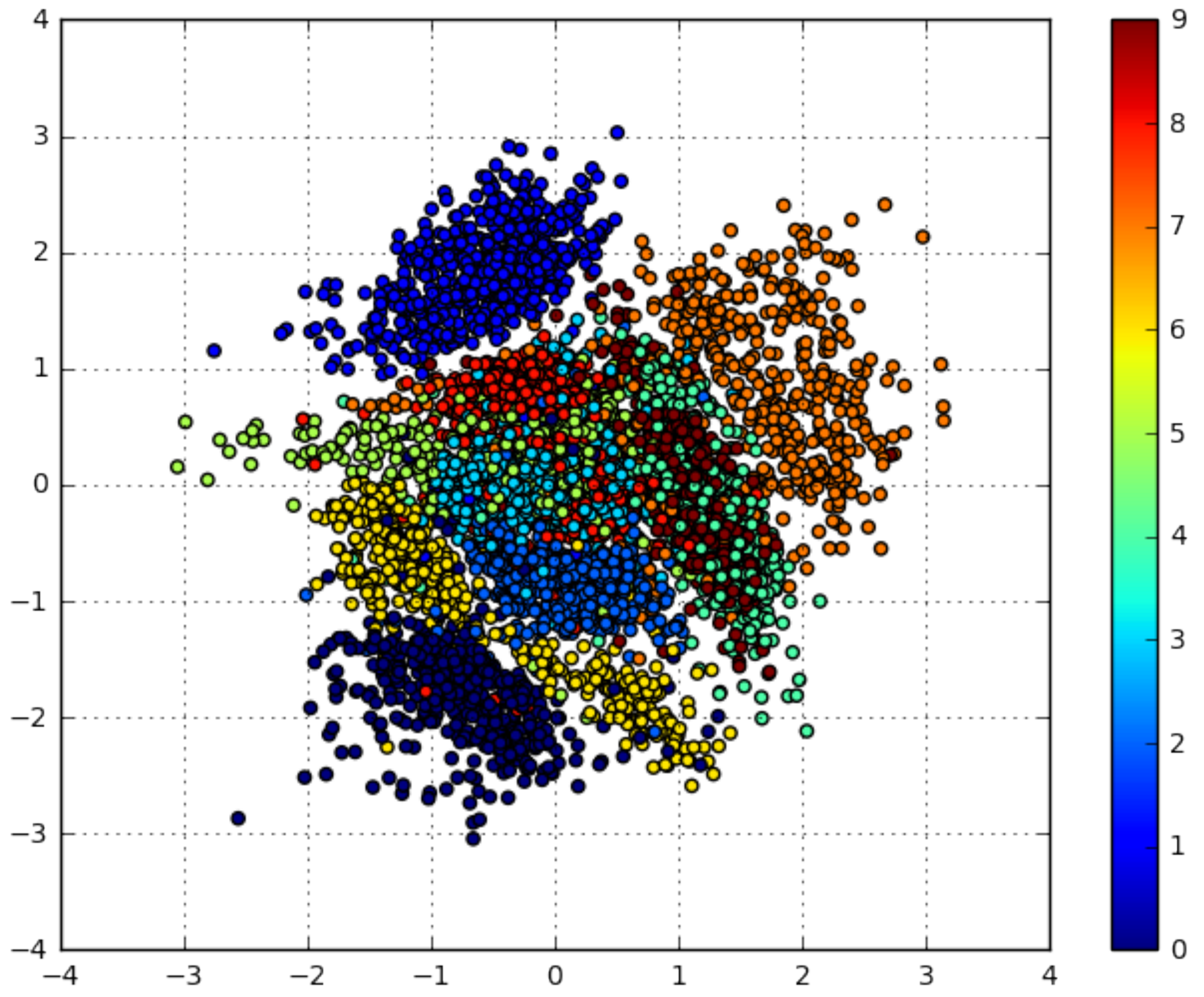
(from here)

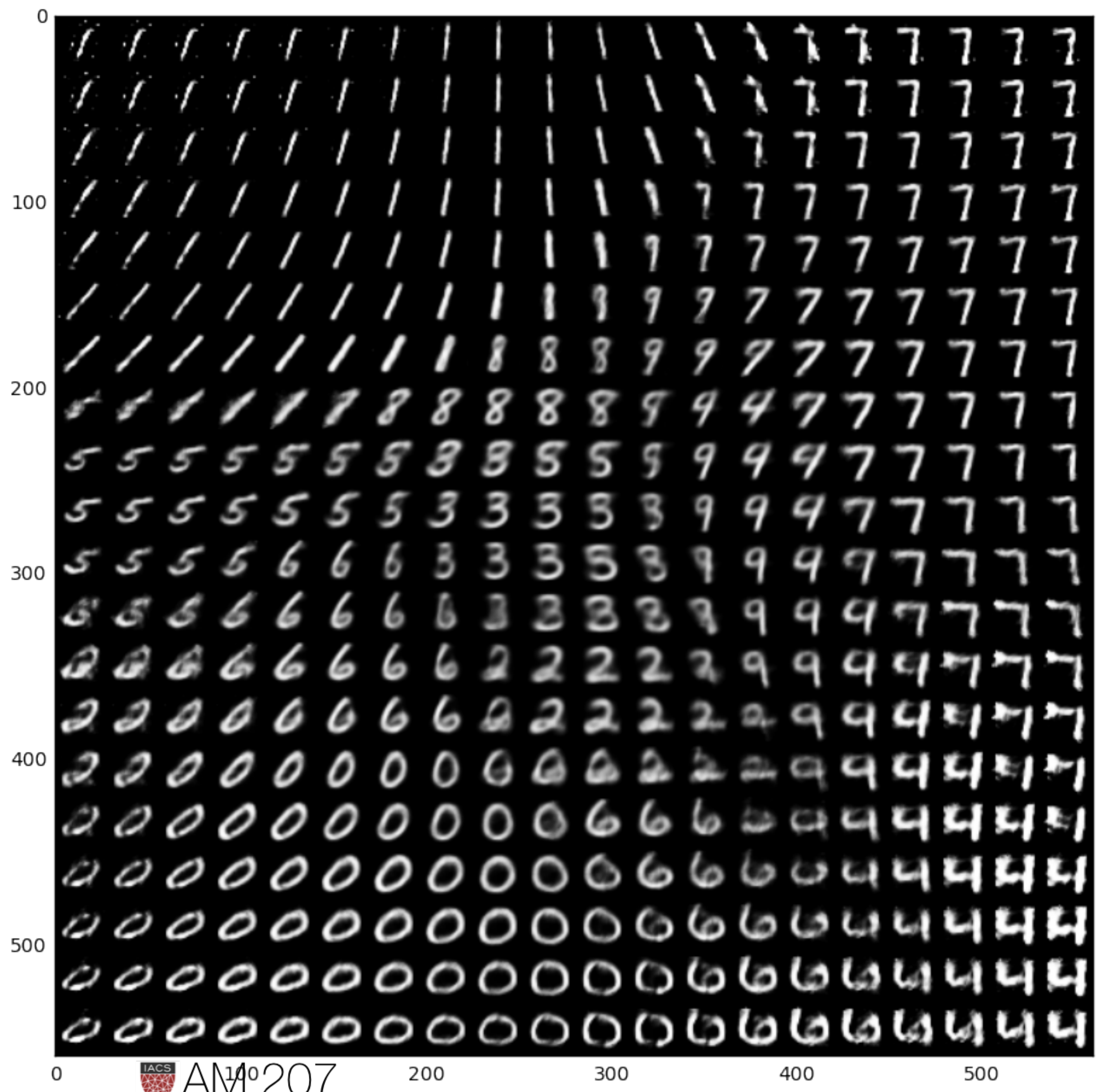

encode →
decode →
input
hidden
output
$q_\phi(z|x)$
$p_\theta(x|z)$
x
→
→
$\tilde{x}$

VAE steps for MNIST

- details in original paper and notebook
- linear encoder for both μ and $log(\sigma^2)$
- then transformation to $N(0, 1)$ to be able to take gradient inside expectation as in ADVI
- then decode using a loss: binary cross-entropy $p(x|z)$ (for images) minus KL

```python
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def encode(self, x):
        h1 = self.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparameterize(self, mu, logvar):
        if self.training:
            std = logvar.mul(0.5).exp_()
            eps = Variable(std.data.new(std.size()).normal_())
            return eps.mul(std).add_(mu)
        else:
            return mu

    def decode(self, z):
        h3 = self.relu(self.fc3(z))
        return self.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar
```



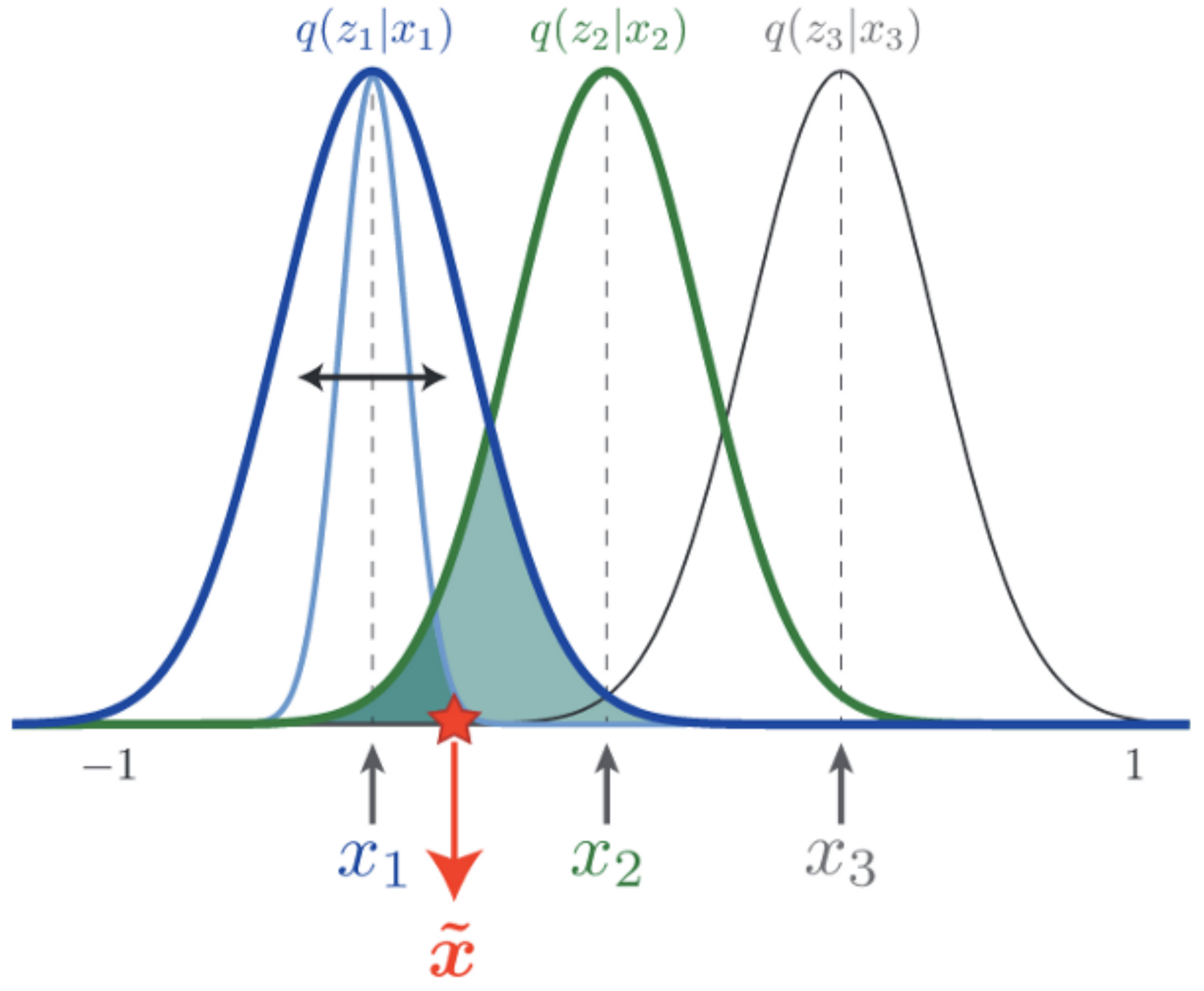


```
model = VAE()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x,
        x.view(-1, 784), size_average=False)
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return BCE + KLD

def train(epoch):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        data = Variable(data)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)
        loss = loss_function(recon_batch, data, mu, logvar)
        loss.backward()
        train_loss += loss.data[0]
        optimizer.step()
    return train_loss / len(train_loader.dataset)

def test(epoch):
    model.eval()
    test_loss = 0
    for i, (data, _) in enumerate(test_loader):
        data = Variable(data, volatile=True)
        recon_batch, mu, logvar = model(data)
        test_loss += loss_function(recon_batch, data, mu, logvar).data[0]
    test_loss /= len(test_loader.dataset)
    return test_loss
```

Images from here

Disentanglement Issues

- can be understood from a gaussian mixtures perspective
- we would prefer data locality
- thus crank up the prior (regularization) term
- this is called the βVAE



How to implement?

- possible in pytorch, also in pymc3
- see convolutional VAE for MNIST in pymc3
- notice that MNIST, which we did earlier as supervised is now being done unsupervised.

Why?

See pymc3 for e.g. for auto-encoding LDA

- variational auto-encoders algorithm which allows us to perform inference efficiently for large datasets
- use tunable and flexible encoders such as multilayer perceptrons (MLPs) as our variational distribution to approximate complex variational posterior
- then its just ADVI with mini-batch on PyMC3 or pytorch. Can use for any posterior, example LDA, or custom for MNIST