Variational Autoencoders

Elia Bronzo, Shoichi Yip

05/03/2024

Learning probabilistic models

Let's start by fixing some notation:

- $\mathbf{x}^{(i)} \in \mathcal{D}$ that we assume to be N i.i.d. from $p^*(\mathbf{x}) \quad i = 1, ..., N$
- $p^*(\mathbf{x})$ is the *True* generative process we want to learn

Let's start considering a **fully observed model**

$$p_\theta(\mathbf{x}) \approx p^*(\mathbf{x})$$

Maximum Likelihood (ML)

$$\begin{aligned}\theta_{ML} &= \arg\max_\theta p_\theta(\mathcal{D}) \\ &= \arg\max_\theta \sum_{i=1}^{N} p_\theta(\mathbf{x}^{(i)})\end{aligned}$$

Maximum a posteriori (MAP)

$$\begin{aligned}\theta_{MAP} &= \arg\max_\theta p(\theta|\mathcal{D}) \\ &= \arg\max_\theta \frac{p_\theta(\mathcal{D})p(\theta)}{p(\mathcal{D})}\end{aligned}$$

Why not using DNNs?

$$p_\theta \leftarrow \text{Deep Neural Networks (DNN)}$$

- high representation capacity
- easily trained with SGD

SGD:

$$\frac{1}{N_\mathcal{D}} \nabla_\theta \log p_\theta(\mathcal{D}) \underbrace{\approx}_{\text{RHS unbiased estimator of LHS}} \underbrace{\frac{1}{N_\mathcal{M}} \nabla_\theta \log p_\theta(\mathcal{M})}_{\text{Gradient over minibatch}}$$

Latent variables models

$p_\theta(\mathbf{x}, \mathbf{z})$ where $\mathbf{z}$ latent variables

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int \underbrace{p_\theta(\mathbf{z})}_{\textbf{Prior}} p_\theta(\mathbf{x}|\mathbf{z}) d\mathbf{z}$$ **Marginal likelihood**

If $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ is a DNN $\rightarrow$ **Deep Latent Variable model (DLVM)**

Why?

- high representation capacity: $p_\theta(\boldsymbol{x}) \rightarrow$ (almost) arbitrary dependencies even with simple (ex. Gaussian) priors
- easily trained with SGD

VAE is used to learn this kind of models

Learning latent variable models

Learning a latent variable model means optimize the **marginal log-likelihood**:

$$l(\theta) = \sum_{i=1}^{N} \log p_\theta(\boldsymbol{x}^{(i)}) = \sum_{i=1}^{N} \int p_\theta(\mathbf{z}) p_\theta(\mathbf{x}^{(i)}|\mathbf{z}) \mathrm{d}\mathbf{z}$$

The *classical* way of optimizing in this problems is **Expectation-Maximization**

We will first focus on how to optimize $l(\theta)$ in case of a dataset composed by only 1 point ($\boldsymbol{x}^{(1)}$) (N=1)

Evidence Lower Bound (ELBO)

$$\begin{aligned}\log p_\theta(\boldsymbol{x}^{(1)}) &= \log \int \mathrm{d}\boldsymbol{z}\, p_\theta(\mathbf{x}^{(1)}, \mathbf{z}) \\ &= \log \underbrace{\int \mathrm{d}\boldsymbol{z}\, q(\boldsymbol{z})}_{\mathbb{E}_q} \frac{p_\theta(\mathbf{x}^{(1)}, \mathbf{z})}{q(\boldsymbol{z})} \\ &\underbrace{\geq}_{\text{Jensen's}} \underbrace{\int \mathrm{d}\boldsymbol{z}\, q(\boldsymbol{z}) \log \frac{p_\theta(\mathbf{x}^{(1)}, \mathbf{z})}{q(\boldsymbol{z})}}_{\textbf{ELBO}}\end{aligned}$$

This is true **for any** $q(z)$

Jensen's inequality f convex:

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$$

- tight bound $\iff$ f strictly convex
- $\log(x)$ concave

More on the ELBO

$$\begin{aligned}
\log p_\theta(\mathbf{x}^{(1)}) &= \mathbb{E}_q\left[\log p_\theta(\mathbf{x}^{(1)})\right] \\
&= \mathbb{E}_q\left[\log\left[\frac{p_\theta(\mathbf{x}^{(1)},\mathbf{z})}{p_\theta(\mathbf{z}|\mathbf{x}^{(1)})}\right]\right] \\
&= \mathbb{E}_q\left[\log\left[\frac{p_\theta(\mathbf{x}^{(1)},\mathbf{z})}{q(\boldsymbol{z})}\frac{q(\boldsymbol{z})}{p_\theta(\mathbf{z}|\mathbf{x}^{(1)})}\right]\right] \\
&= \mathbb{E}_q\left[\log\left[\frac{p_\theta(\mathbf{x}^{(1)},\mathbf{z})}{q(\boldsymbol{z})}\right]\right] + \mathbb{E}_q\left[\log\left[\frac{q(\mathbf{z})}{p_\theta(\mathbf{z}|\mathbf{x}^{(1)})}\right]\right] \\
&= \text{ELBO}(\boldsymbol{x}^{(1)}; q, \theta) + \mathcal{D}_{\text{KL}}\left[q(\mathbf{z}) \,\Big|\Big|\, p_\theta(\mathbf{z}|\mathbf{x})\right]
\end{aligned}$$

When does the ELBO is **tight**? (equality)

$$\text{ELBO}(\boldsymbol{x}^{(1)}; q, \theta) = \log p_\theta(\mathbf{x}^{(1)}) \iff q(\mathbf{z}) = p_\theta(\mathbf{z}|\mathbf{x}^{(1)})$$

Expectation-Maximization Algorithm

- **Expectation** step: at fixed $\theta^{(t)}$, $q(\mathbf{z}) \leftarrow p_{\theta^{(t)}}(\mathbf{z}|\mathbf{x}^{(1)})$
- **Maximization** step: at fixed q, $\theta^{(t+1)} \leftarrow \arg\max_\theta \mathrm{ELBO}(\boldsymbol{x}^{(1)}; q, \theta)$

What about a dataset with more than 1 point? $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(N)}\}$

$$l(\theta) = \sum_{i=1}^{N} \log p_\theta(\boldsymbol{x}^{(i)}) \geq \sum_{i=1}^{N} \mathrm{ELBO}(\boldsymbol{x}^{(i)}; q_i, \theta)$$

Repeat until convergence {

(E-step) **for each i**:

$$q_i(\boldsymbol{z}) \leftarrow p_{\theta^{(t)}}(\mathbf{z}|\mathbf{x}^{(i)})$$

(M-step)

$$\theta^{(t+1)} \leftarrow \arg\max_\theta \textstyle\sum_{i=1}^{N} \mathrm{ELBO}(\boldsymbol{x}^{(i)}; q, \theta) \text{ }\}$$

DLVM, intractabilities and problems with EM

$$l(\theta) = \sum_{i=1}^{N} \log p_\theta(\boldsymbol{x}^{(i)}) = \sum_{i=1}^{N} \underbrace{\int p_\theta(\mathbf{z}) p_\theta(\mathbf{x}^{(i)}|\mathbf{z}) d\mathbf{z}}_{\text{intractable for a DNN}}$$

Intractable means that we don't have neither an *analytical solution* nor an *efficient estimator* of the integral in case of a DNN

Intractable $p_\theta(\boldsymbol{x}) \iff$ Intractable **posterior** $p_\theta(\boldsymbol{z}|\boldsymbol{x}) = \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{p_\theta(\boldsymbol{x})} = \frac{p_\theta(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{p_\theta(\boldsymbol{x})}$

E-step: evaluation of an intractable posterior with MCMC **for each data-point** $\rightarrow$ computationally expensive

Solutions? $\rightarrow$ **VAE**

Variational Auto-Encoders (VAE)

Model for the posterior $q_\phi(\mathbf{z}|\mathbf{x}) \rightarrow$ another DNN

Latent representations

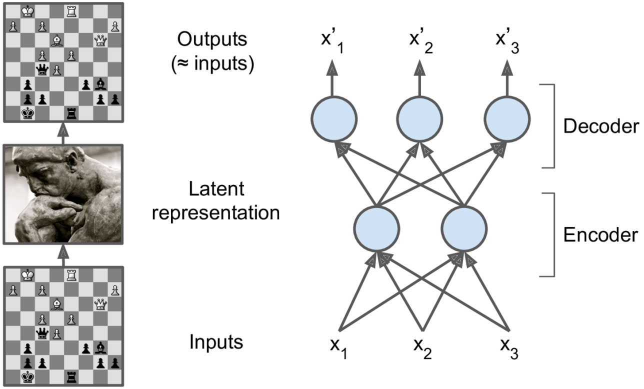

Figure: Chess memory experiment and structure of an autoencoder (from HOML).

Undercompleteness

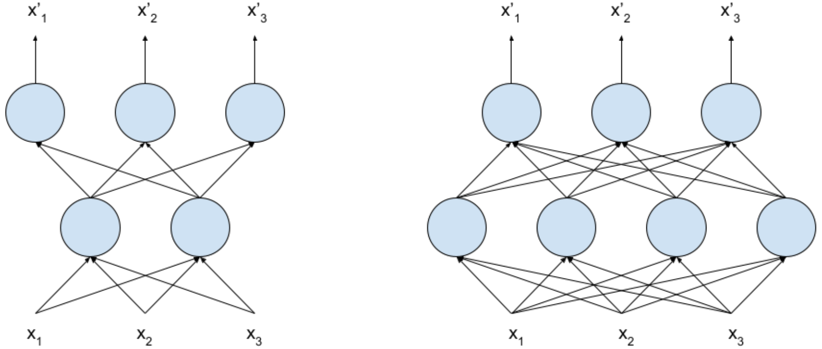

Figure: The typical structure of an autoencoder and of an MLP.

What is the autoencoder learning?

The autoencoder learns the identity. We expect the output to resemble to the input.

But it is characterised by the information bottleneck that yields undercompleteness.

Depending on the characteristics of the bottleneck, we get different autoencoders.

Stacked autoencoder

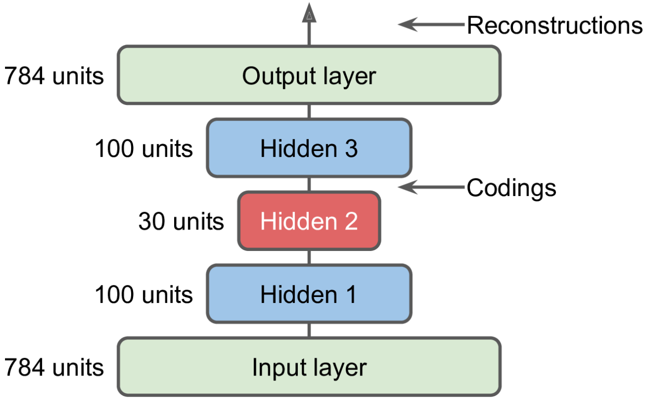

Figure: Structure of a stacked autoencoder (from HOML).

Reconstruction for the stacked autoencoder

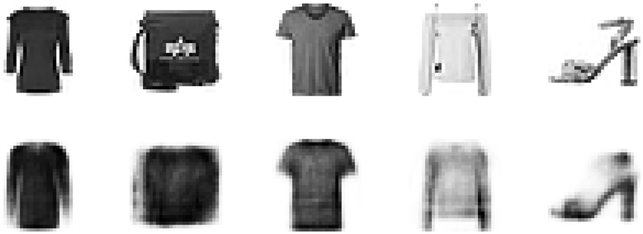

Figure: Reconstruction of images on a stacked autoencoder (from HOML).

Denoising autoencoder

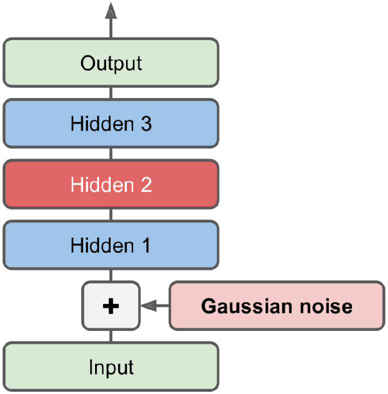

Figure: Structure of a denoising autoencoder (from HOML).

Reconstruction of noiseless images

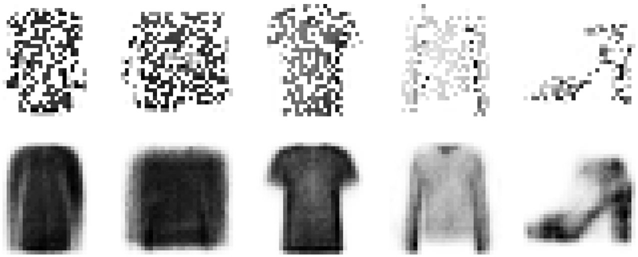

Figure: Reconstruction of noiseless images on a denoising autoencoder (from HOML).

Pros and Cons

What autoencoders do:

- learn latent representation
- semisupervised learning
- pretraining using greedy layerwise approach
- denoising images

Pros and Cons

What autoencoders do not do:

- have a natural interpolation scheme
- have generative capacity

Variational autoencoder

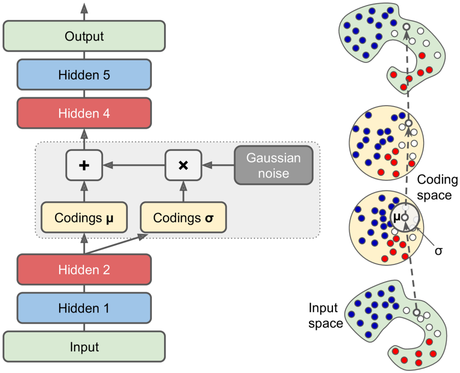

Figure: Structure of a variational autoencoder (from HOML).

The decoder

We have said that our model is generative.

The generative role of the variational autoencoder is played by the decoder.The role of the decoder is to generate new samples that belong to the input space $\mathcal{X}$ starting from a given vector $\boldsymbol{z}$ that belongs to the space of latent variables.

The role of the decoder network is to learn the nonlinear mapping $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ that associates a value of $\boldsymbol{x}$ given a point in the latent space $\boldsymbol{z}$.[1]

[1]This is justified by the **universal approximation theory** that states that deep ANNs are able to approximate any function.

The decoder

We can make the assumption that the $p(\boldsymbol{x}|\boldsymbol{z})$ is a multivariate Gaussian distribution, for example in the case of MNIST datasets.

$$p_\theta(\boldsymbol{x}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{x}|\mu_\theta(\boldsymbol{z}), \sigma_\theta^2(\boldsymbol{z}))$$

Inputs in that case are, in fact, vectors of real valued variables (i.e. pixel brightness).

Back to the probabilistic picture

Remember that we are trying to maximise the **likelihood** $p(x_i)$ for each i-th data point.

Given a new probability distribution $q_i(\boldsymbol{z})$ we have

$$\begin{aligned} p(x_i) &= \int p(x_i|\boldsymbol{z})p(\boldsymbol{z})\,\mathrm{d}\boldsymbol{z} \\ &= \int p(x_i|\boldsymbol{z})p(\boldsymbol{z})\frac{q_i(\boldsymbol{z})}{q_i(\boldsymbol{z})}\,\mathrm{d}\boldsymbol{z} \\ &= \left\langle \frac{p(x_i|\boldsymbol{z})p(\boldsymbol{z})}{q_i(\boldsymbol{z})} \right\rangle_{z\sim q_i(\boldsymbol{z})} \end{aligned}$$

Back to the probabilistic picture

Per Jensen, we have that the **log likelihood** has the following bound

$$\begin{aligned}\log(p(x_i)) &\geq \left\langle \log \frac{p(x_i|\boldsymbol{z})p(\boldsymbol{z})}{q_i(\boldsymbol{z})} \right\rangle_{z\sim q_i(z)} \\ &\geq \langle \log p(x_i|\boldsymbol{z})(p(\boldsymbol{z}))\rangle_{z\sim q_i(z)} - H[q_i(z)]\end{aligned}$$

The RHS is what we have previously called ELBO (Evidence Lower Bound). We choose to maximise the ELBO instead of the log likelihood. But we are maximising against a lot of parameters: we have to optimise against θ and against each i-th $q_i(z)$!

Can we reduce the number of parameters to optimise against?

Introducing: the encoder network

What if we add another network, but this time to perform the **inference**? This means that we choose the $q_i(z)$ to be

$$q_i(z) = q_\phi(\boldsymbol{z}|\boldsymbol{x})$$

where we call $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ the **encoder** and ϕ is the set of parameters of the variational inference.

Deep and amortised

Let us recap. In one go we

1. chose to perform the **deep inference**, which implies that we will train a neural network to **predict** the variational parameters ϕ instead of getting them by directly optimising the ELBO.
2. reduced the number of parameters that we have to optimise against since we do not have anymore N approximate posteriors $q_i(z)$'s, but one parameter set ϕ shared by all the data points. Since the task of inference is amortised across the entire dataset, this technique is called **amortised variational inference**.

The Kullback-Leibler divergence

We are going to compute the Kullback-Leibler divergence for a particular case, in which the real and approximate posteriors have take the multinomial Gaussian form.

$$\mathcal{D}_{\mathsf{KL}}\left[q_\phi(\boldsymbol{z}|\boldsymbol{x}) \,\Big|\Big|\, p_\theta(\boldsymbol{z})\right] = \int \mathrm{d}z\, q_\phi(\boldsymbol{z})[\log p_\theta(\boldsymbol{z}) - \log q_\phi(\boldsymbol{z})] \tag{1}$$

This assumption can be made, for example, in the case of the MNIST dataset, since pixels are labeled with values that range from 0 to 255 depending on the intensity of the pixel. If pixels were binary variables, a Bernoulli distribution assumption would have been more suited.

First term of D_{KL}

$$\int q_\phi(\boldsymbol{z}) \log p(\boldsymbol{z}) \, \mathrm{d}\boldsymbol{z} =$$

$$= \int \mathrm{d}\boldsymbol{z} \, \frac{1}{\sqrt{\det(\Sigma)}\sqrt{2\pi}^L} \prod_{l=1}^{L} \exp\left\{ -\frac{(z_l - \mu_l)^2}{2\sigma_l^2} \right\} \log \frac{1}{\sqrt{2\pi}^L} e^{-\frac{|\boldsymbol{z}|^2}{2}}$$

First term of D_{KL}

$$\int q_\phi(\boldsymbol{z}) \log p(\boldsymbol{z}) \, \mathrm{d}\boldsymbol{z} =$$

$$= \int \mathrm{d}\boldsymbol{z} \, \frac{1}{\sqrt{\det(\Sigma)}\sqrt{2\pi}^L} \prod_{l=1}^{L} \exp\left\{ -\frac{(z_l - \mu_l)^2}{2\sigma_l^2} \right\} \left[-\frac{L}{2} \log 2\pi - \frac{|\boldsymbol{z}|^2}{2} \right]$$

$$= -\frac{L}{2} \log 2\pi - \frac{1}{2} \sum_{l=1}^{L} (\sigma_l^2 + \mu_l^2)$$

Second term of D_{KL}

$$
\begin{aligned}
&\int q_\phi(\boldsymbol{z}) \log q_\phi(\boldsymbol{z}) \, \mathrm{d}\boldsymbol{z} \\
&= \int \mathrm{d}\boldsymbol{z} \, \frac{1}{\sqrt{\det(\Sigma)}\sqrt{2\pi}^L} \prod_{l=1}^{L} \exp\left\{-\frac{(z_l - \mu_l)^2}{2\sigma_l^2}\right\} \\
&\qquad \log \frac{1}{\sqrt{\det(\Sigma)}\sqrt{2\pi}^L} \exp\left\{-\frac{(z_l - \mu_l)^2}{2\sigma_l^2}\right\}
\end{aligned}
$$

Second term of D_{KL}

$$
\begin{aligned}
&\int q_\phi(\boldsymbol{z}) \log q_\phi(\boldsymbol{z}) \,\mathrm{d}\boldsymbol{z} \\
&= \int \mathrm{d}\boldsymbol{z}\, \frac{1}{\sqrt{\det(\Sigma)}\sqrt{2\pi}^L} \prod_{l=1}^{L} \exp\left\{-\frac{(z_l-\mu_l)^2}{2\sigma_l^2}\right\} \\
&\qquad \left[-\frac{L}{2}\log 2\pi - \log \sigma_l^2 - \frac{(z_l-\mu_l)^2}{2\sigma_l^2}\right] \\
&= -\frac{L}{2}\log 2\pi - \frac{1}{2}\sum_{l=1}^{L}(1+\log \sigma_l^2)
\end{aligned}
$$

Final expression for the Kullback-Leibler divergence

KL divergence for the Gaussian case

The KL divergence in the Gaussian case is given by

$$\mathcal{D}_{\mathrm{KL}}\left[q_\phi(\boldsymbol{z}|\boldsymbol{x}) \,\middle\|\, p_\theta(\boldsymbol{z})\right] = \frac{1}{2}\sum_{l=1}^{L}\left[1 + \log\sigma_l^2 - \sigma_l^2 - \mu_l^2\right] \tag{2}$$

We have hence obtained the tightness term of our loss function given by the ELBO.

The loss function

Loss function for the Gaussian case

The loss function for the Gaussian case is given by

$$\mathcal{L}_{\theta,\phi}(\boldsymbol{x}^{(i)}) = (\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}}^{(i)})^2 - \frac{1}{2}\sum_{l=1}^{L}\left[1 + \log\sigma_l^2 - \sigma_l^2 - \mu_l^2\right] \tag{3}$$

In this simplified case, hence, we can get an closed explicit form for the loss function. It can be directly implemented in the neural network.

A nondeterministic component

Rememeber the structure of the variational autoencoder.

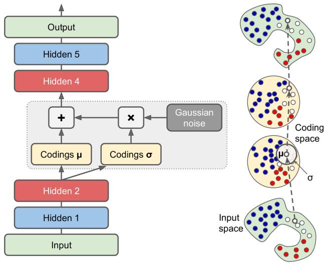

Figure: Structure of a variational autoencoder (from HOML).

The part in grey is a nondeterministic component, since it has a noise term. This makes backpropagation unfeasible! How shall we deal with it?

Framing the problem

Formally, what bothers us is that when we are backpropagating weight updates through the neural network, we want to be able to compute

$$\theta^{\text{new}} = \theta^{\text{old}} + \eta \nabla_\theta \text{ELBO}_{\theta,\phi}$$

$$\phi^{\text{new}} = \phi^{\text{old}} + \eta \nabla_\phi \text{ELBO}_{\theta,\phi}$$

where

$$\text{ELBO}_{\theta,\phi} = \langle \log p(x_i|\boldsymbol{z})(p(\boldsymbol{z}))\rangle_{z\sim q_\phi(\boldsymbol{z}|\boldsymbol{x})} - H[q_\phi(\boldsymbol{z}|\boldsymbol{x})]$$

First derivation is OK!
Second derivation is NOT OK! Expectation value and gradient do not commute in this case. What can we do?

Let it commute!

Let us make it commute! In fact, in particularly simple settings such as when the approximate posteriors $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ have Gaussian form

$$q_\phi(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{\mu}(\boldsymbol{x}), \boldsymbol{\sigma}(\boldsymbol{x}))$$

the $\boldsymbol{z} \sim q_\phi(\boldsymbol{z}|\boldsymbol{x})$ can be rewritten as

$$\boldsymbol{z} = \mu(\boldsymbol{x}) + \epsilon \cdot \sigma(\boldsymbol{x})$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$.

The reparametrization trick

Instead of sampling from the approximate variational posterior we sample ϵ from a base noise distribution and apply a deterministic transformation.

This amounts to replacing $\boldsymbol{z} \sim q_\phi(\boldsymbol{z}|\boldsymbol{x})$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$ and defining the new deterministic $\boldsymbol{z} = \boldsymbol{m}u + \boldsymbol{\epsilon}\sigma$. It is called reparametrization trick

$$\begin{aligned}\nabla_\phi \mathrm{ELBO}_{\theta,\phi} &= \nabla_\phi \left\langle \log p(\boldsymbol{x}, \boldsymbol{\mu} + \epsilon\sigma) \right\rangle_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{1})} - \nabla_\phi H[q_\phi(\boldsymbol{z}|\boldsymbol{x})] \\ &= \left\langle \nabla_\phi \log p(\boldsymbol{x}, \boldsymbol{\mu} + \epsilon\sigma) \right\rangle_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{1})} - \nabla_\phi H[q_\phi(\boldsymbol{z}|\boldsymbol{x})]\end{aligned}$$

Structure of our network

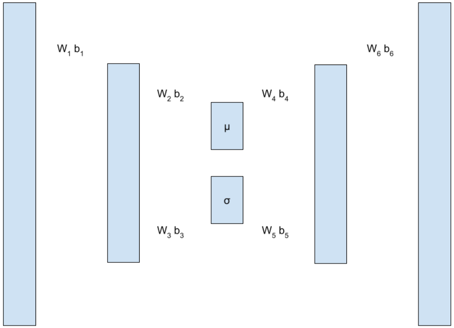

Figure: Structure of our variational autoencoder

Let's get to work

The Colab notebook is available at the following URL
`https://bit.ly/pcsml_vae`