# Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1]

Department of Physics and Center for Computing in Science Education,
University of Oslo, Norway[1]

April 10, 2025

# Plans for the week April 7-11, 2025

## Generative methods, energy models and Boltzmann machines

1. Summary of discussions on Restricted Boltzmann machines, reminder from last week
2. Introduction to Variational Autoencoders (VAEs)
3. Video of lecture
4. Whiteboard notes

# Reading recommendations

1. Boltzmann machines: Goodfellow et al chapters 18.1-18.2, 20.1-20-7; To create Boltzmann machine using Keras, see Babcock and Bali chapter 4, see https://github.com/PacktPublishing/Hands-On-Generative-AI-with-Python-and-TensorFlow-2/blob/master/Chapter_4/models/rbm.py
2. More on Boltzmann machines: see also Foster, chapter 7 on energy-based models at https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition/tree/main/notebooks/07_ebm/01_ebm
3. VAEs: Goodfellow et al, for VAEs see sections 20.10-20.11

# Essential elements of generative models

The aim of generative methods is to train a probability distribution $p$. The methods we will focus on are:

1. Energy based models, with the family of Boltzmann distributions as a typical example
2. Variational autoencoders, based on our discussions on autoencoders
3. Generative adversarial networks (GANs) and
4. Diffusion models

# Energy models, reminders from last two weeks

During the last two weeks we defined a domain $\boldsymbol{X}$ of stochastic variables $\boldsymbol{X} = \{x_0, x_1, \ldots, x_{n-1}\}$ with a pertinent probability distribution

$$p(\boldsymbol{X}) = \prod_{x_i \in \boldsymbol{X}} p(x_i),$$

where we have assumed that the random varaibles $x_i$ are all independent and identically distributed (iid).
We will now assume that we can defined this function in terms of optimization parameters $\boldsymbol{\Theta}$, which could be the biases and weights of deep network, and a set of hidden variables we also assume to be random variables which also are iid. The domain of these variables is $\boldsymbol{H} = \{h_0, h_1, \ldots, h_{m-1}\}$.

# Probability model

We define a probability

$$p(x_i, h_j; \boldsymbol{\Theta}) = \frac{f(x_i, h_j; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

where $f(x_i, h_j; \boldsymbol{\Theta})$ is a function which we assume is larger or equal than zero and obeys all properties required for a probability distribution and $Z(\boldsymbol{\Theta})$ is a normalization constant. Inspired by statistical mechanics, we call it often for the partition function. It is defined as (assuming that we have discrete probability distributions)

$$Z(\boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j; \boldsymbol{\Theta}).$$

# Marginal and conditional probabilities

We can in turn define the marginal probabilities

$$p(x_i; \boldsymbol{\Theta}) = \frac{\sum_{h_j \in \boldsymbol{H}} f(x_i, h_j; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

and

$$p(h_i; \boldsymbol{\Theta}) = \frac{\sum_{x_i \in \boldsymbol{X}} f(x_i, h_j; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})}.$$

# Change of notation

**Note the change to a vector notation**. A variable like $\boldsymbol{x}$ represents now a specific **configuration**. We can generate an infinity of such configurations. The final partition function is then the sum over all such possible configurations, that is

$$Z(\boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j; \boldsymbol{\Theta}),$$

changes to

$$Z(\boldsymbol{\Theta}) = \sum_{\boldsymbol{x}} \sum_{\boldsymbol{h}} f(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta}).$$

If we have a binary set of variable $x_i$ and $h_j$ and $M$ values of $x_i$ and $N$ values of $h_j$ we have in total $2^M$ and $2^N$ possible $\boldsymbol{x}$ and $\boldsymbol{h}$ configurations, respectively.
We see that even for the modest binary case, we can easily approach a number of configuration which is not possible to deal with.

# Optimization problem

At the end, we are not interested in the probabilities of the hidden variables. The probability we thus want to optimize is

$$p(\boldsymbol{X};\boldsymbol{\Theta}) = \prod_{x_i \in \boldsymbol{X}} p(x_i;\boldsymbol{\Theta}) = \prod_{x_i \in \boldsymbol{X}} \left( \frac{\sum_{h_j \in \boldsymbol{H}} f(x_i, h_j;\boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})} \right),$$

which we rewrite as

$$p(\boldsymbol{X};\boldsymbol{\Theta}) = \frac{1}{Z(\boldsymbol{\Theta})} \prod_{x_i \in \boldsymbol{X}} \left( \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j;\boldsymbol{\Theta}) \right).$$

# Further simplifications

We simplify further by rewriting it as

$$p(\boldsymbol{X};\boldsymbol{\Theta}) = \frac{1}{Z(\boldsymbol{\Theta})} \prod_{x_i \in \boldsymbol{X}} f(x_i;\boldsymbol{\Theta}),$$

where we used $p(x_i;\boldsymbol{\Theta}) = \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j;\boldsymbol{\Theta})$. The optimization problem is then

$$\arg \max_{\boldsymbol{\Theta} \in \mathbb{R}^p} p(\boldsymbol{X};\boldsymbol{\Theta}).$$

# Optimizing the logarithm instead

Computing the derivatives with respect to the parameters $\boldsymbol{\Theta}$ is easier (and equivalent) with taking the logarithm of the probability. We will thus optimize

$$\arg \max_{\boldsymbol{\Theta} \in \mathbb{R}^p} \log p(\boldsymbol{X}; \boldsymbol{\Theta}),$$

which leads to

$$\nabla_{\boldsymbol{\Theta}} \log p(\boldsymbol{X}; \boldsymbol{\Theta}) = 0.$$

# Expression for the gradients

This leads to the following equation

$$\nabla_{\boldsymbol{\Theta}} \log p(\boldsymbol{X}; \boldsymbol{\Theta}) = \nabla_{\boldsymbol{\Theta}} \left( \sum_{x_i \in \boldsymbol{X}} \log f(x_i; \boldsymbol{\Theta}) \right) - \nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = 0.$$

The first term is called the positive phase and we assume that we have a model for the function $f$ from which we can sample values. Below we will develop an explicit model for this. The second term is called the negative phase and is the one which leads to more difficulties.

# The derivative of the partition function

The partition function, defined above as

$$Z(\boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j; \boldsymbol{\Theta}),$$

is in general the most problematic term. In principle both $x$ and $h$ can span large degrees of freedom, if not even infinitely many ones, and computing the partition function itself is often not desirable or even feasible. The above derivative of the partition function can however be written in terms of an expectation value which is in turn evaluated using Monte Carlo sampling and the theory of Markov chains, popularly shortened to MCMC (or just $\mathrm{MC}^2$).

# Explicit expression for the derivative

We can rewrite

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \frac{\nabla_{\boldsymbol{\Theta}} Z(\boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

which reads in more detail

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \frac{\nabla_{\boldsymbol{\Theta}} \sum_{x_i \in \boldsymbol{X}} f(x_i; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})}.$$

We can rewrite the function $f$ (we have assumed that is larger or equal than zero) as $f = \exp \log f$. We can then reqrite the last equation as

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \frac{\sum_{x_i \in \boldsymbol{X}} \nabla_{\boldsymbol{\Theta}} \exp \log f(x_i; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})}.$$

# Final expression

Taking the derivative gives us

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \frac{\sum_{x_i \in \boldsymbol{X}} f(x_i; \boldsymbol{\Theta}) \nabla_{\boldsymbol{\Theta}} \log f(x_i; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

which is the expectation value of $\log f$

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} p(x_i; \boldsymbol{\Theta}) \nabla_{\boldsymbol{\Theta}} \log f(x_i; \boldsymbol{\Theta}),$$

that is

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \mathbb{E}(\log f(x_i; \boldsymbol{\Theta})).$$

This quantity is evaluated using Monte Carlo sampling, with Gibbs sampling as the standard sampling rule. Before we discuss the explicit algorithms, we need to remind ourselves about Markov chains and sampling rules like the Metropolis-Hastings algorithm and Gibbs sampling.

# Positive and negative phases

As discussed earlier, the data-dependent term in the gradient is known as the positive phase of the gradient, while the model-dependent term is known as the negative phase of the gradient. The aim of the training is to lower the energy of configurations that are near observed data points (increasing their probability), and raising the energy of configurations that are far from observed data points (decreasing their probability).

# Theory of Variational Autoencoders

Let us remind ourself about what an autoencoder is, see the jupyter-notebooks at https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week8/ipynb/week8.ipynb and https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week9/ipynb/week9.ipynb.

# The Autoencoder again

Autoencoders are neural networks where the outputs are its own inputs. They are split into an **encoder part** which maps the input $\boldsymbol{x}$ via a function $f(\boldsymbol{x}, \boldsymbol{W})$ (this is the encoder part) to a **so-called code part** (or intermediate part) with the result $\boldsymbol{h}$

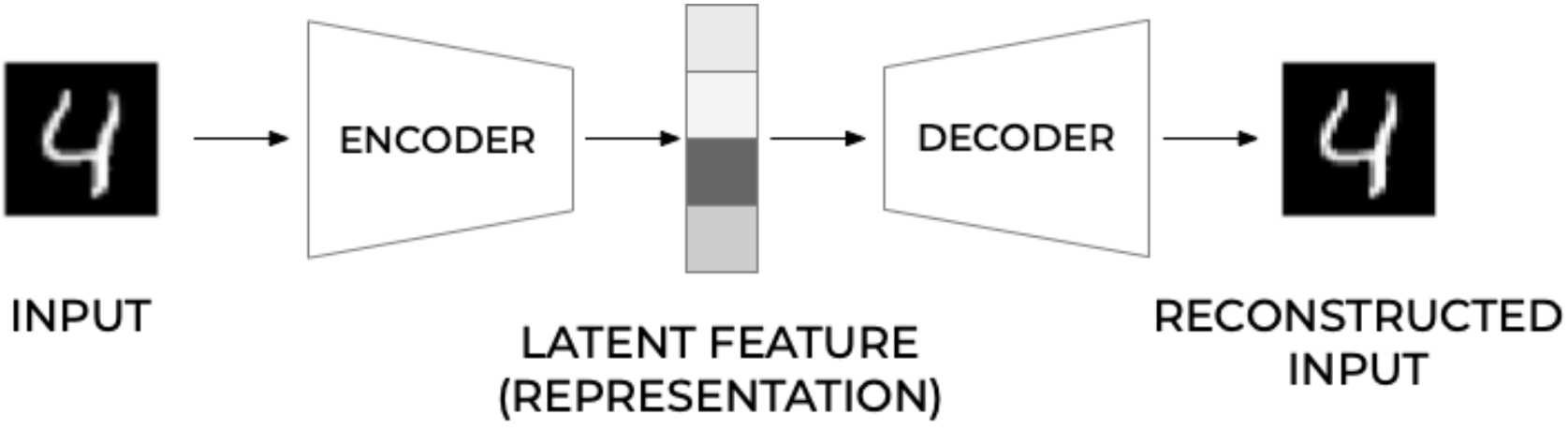
$$\boldsymbol{h} = f(\boldsymbol{x}, \boldsymbol{W})),$$

where $\boldsymbol{W}$ are the weights to be determined. The **decoder** parts maps, via its own parameters (weights given by the matrix $\boldsymbol{V}$ and its own biases) to the final ouput

$$\tilde{\boldsymbol{x}} = g(\boldsymbol{h}, \boldsymbol{V})).$$

The goal is to minimize the construction error, often done by optimizing the means squared error.

# Schematic image of an Autoencoder



# Mathematics of Variational Autoencoders

We have defined earlier a probability (marginal) distribution with hidden variables $\boldsymbol{h}$ and parameters $\boldsymbol{\Theta}$ as

$$p(\boldsymbol{x};\boldsymbol{\Theta}) = \int d\boldsymbol{h} p(\boldsymbol{x},\boldsymbol{h};\boldsymbol{\Theta}),$$

for continuous variables $\boldsymbol{h}$ and

$$p(\boldsymbol{x};\boldsymbol{\Theta}) = \sum_{\boldsymbol{h}} p(\boldsymbol{x},\boldsymbol{h};\boldsymbol{\Theta}),$$

for discrete stochastic events $\boldsymbol{h}$. The variables $\boldsymbol{h}$ are normally called the **latent variables** in the theory of autoencoders. We will also call then for that here.

# Using the conditional probability

Using the the definition of the conditional probabilities $p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta})$, $p(\boldsymbol{h}|\boldsymbol{x};\boldsymbol{\Theta})$ and and the prior $p(\boldsymbol{h})$, we can rewrite the above equation as

$$p(\boldsymbol{x};\boldsymbol{\Theta}) = \sum_{\boldsymbol{h}} p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta})p(\boldsymbol{h}),$$

which allows us to make the dependence of $\boldsymbol{x}$ on $\boldsymbol{h}$ explicit by using the law of total probability. The intuition behind this approach for finding the marginal probability for $\boldsymbol{x}$ is to optimize the above equations with respect to the parameters $\boldsymbol{\Theta}$. This is done normally by maximizing the probability, the so-called maximum-likelihood approach discussed earlier.

# VAEs versus autoencoders

This trained probability is assumed to be able to produce similar samples as the input. In VAEs it is then common to compare via for example the mean-squared error or the cross-entropy the predicted values with the input values. Compared with autoencoders, we are now producing a probability instead of a functions which mimicks the input.
In VAEs, the choice of this output distribution is often Gaussian, meaning that the conditional probability is

$$p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta}) = N(\boldsymbol{x}|f(\boldsymbol{h};\boldsymbol{\Theta}), \sigma^2 \times \boldsymbol{I}),$$

with mean value given by the function $f(\boldsymbol{h};\boldsymbol{\Theta})$ and a diagonal covariance matrix multiplied by a parameter $\sigma^2$ which is treated as a hyperparameter.

# Gradient descent

By having a Gaussian distribution, we can use gradient descent (or any other optimization technique) to increase $p(\boldsymbol{x}; \boldsymbol{\Theta})$ by making $f(\boldsymbol{h}; \boldsymbol{\Theta})$ approach $\boldsymbol{x}$ for some $\boldsymbol{h}$, gradually making the training data more likely under the generative model. The important property is simply that the marginal probability can be computed, and it is continuous in $\boldsymbol{\Theta}$.

# Are VAEs just modified autoencoders?

The mathematical basis of VAEs actually has relatively little to do with classical autoencoders, for example the sparse autoencoders or denoising autoencoders discussed earlier.
VAEs approximately maximize the probability equation discussed above. They are called autoencoders only because the final training objective that derives from this setup does have an encoder and a decoder, and resembles a traditional autoencoder. Unlike sparse autoencoders, there are generally no tuning parameters analogous to the sparsity penalties. And unlike sparse and denoising autoencoders, we can sample directly from $p(\boldsymbol{x})$ without performing Markov Chain Monte Carlo.

# Training VAEs

To solve the integral or sum for $p(\boldsymbol{x})$, there are two problems that VAEs must deal with: how to define the latent variables $\boldsymbol{h}$, that is decide what information they represent, and how to deal with the integral over $\boldsymbol{h}$. VAEs give a definite answer to both.

# Motivation from Kingma and Welling, An Introduction to Variational Autoencoders,

https://arxiv.org/abs/1906.02691

> *There are many reasons why generative modeling is attractive. First, we can express physical laws and constraints into the generative process while details that we don't know or care about, i.e. nuisance variables, are treated as noise. The resulting models are usually highly intuitive and interpretable and by testing them against observations we can confirm or reject our theories about how the world works. Another reason for trying to understand the generative process of data is that it naturally expresses causal relations of the world. Causal relations have the great advantage that they generalize much better to new situations than mere correlations. For instance, once we understand the generative process of an earthquake, we can use that knowledge both in California and in Chile.*

# Mathematics of VAEs

We want to train the marginal probability with some latent varrables $\boldsymbol{h}$

$$p(\boldsymbol{x}; \boldsymbol{\Theta}) = \int d\boldsymbol{h} p(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta}),$$

for the continuous version (see previous slides for the discrete variant).

# Using the KL divergence

In practice, for most $\boldsymbol{h}$, $p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta})$ will be nearly zero, and hence contributes almost nothing to our estimate of $p(\boldsymbol{x})$.
The key idea behind the variational autoencoder is to attempt to sample values of $\boldsymbol{h}$ that are likely to have produced $\boldsymbol{x}$, and compute $p(\boldsymbol{x})$ just from those.
This means that we need a new function $Q(\boldsymbol{h}|\boldsymbol{x})$ which can take a value of $\boldsymbol{x}$ and give us a distribution over $\boldsymbol{h}$ values that are likely to produce $\boldsymbol{x}$. Hopefully the space of $\boldsymbol{h}$ values that are likely under $Q$ will be much smaller than the space of all $\boldsymbol{h}$'s that are likely under the prior $p(\boldsymbol{h})$. This lets us, for example, compute $E_{\boldsymbol{h}\sim Q}p(\boldsymbol{x}|\boldsymbol{h})$ relatively easily. Note that we drop $\boldsymbol{\Theta}$ from here and for notational simplicity.

# Kullback-Leibler again

However, if $\boldsymbol{h}$ is sampled from an arbitrary distribution with PDF $Q(\boldsymbol{h})$, which is not $\mathcal{N}(0, I)$, then how does that help us optimize $p(\boldsymbol{x})$?
The first thing we need to do is relate $E_{\boldsymbol{h}\sim Q} P(\boldsymbol{x}|\boldsymbol{h})$ and $p(\boldsymbol{x})$. We will see where $Q$ comes from later.
The relationship between $E_{\boldsymbol{h}\sim Q} p(\boldsymbol{x}|\boldsymbol{h})$ and $p(\boldsymbol{x})$ is one of the cornerstones of variational Bayesian methods. We begin with the definition of Kullback-Leibler divergence (KL divergence or $\mathcal{D}$) between $p(\boldsymbol{h}|\boldsymbol{x})$ and $Q(\boldsymbol{h})$, for some arbitrary $Q$ (which may or may not depend on $\boldsymbol{x}$):

$$\mathcal{D}\left[Q(\boldsymbol{h}) \| p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{h}|\boldsymbol{x})\right].$$

# And applying Bayes rule

We can get both $p(\boldsymbol{x})$ and $p(\boldsymbol{x}|\boldsymbol{h})$ into this equation by applying Bayes rule to $p(\boldsymbol{h}|\boldsymbol{x})$

$$\mathcal{D}\left[Q(\boldsymbol{h})\|p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{x}|\boldsymbol{h}) - \log p(\boldsymbol{h})\right] + \log p(\boldsymbol{x}).$$

Here, $\log p(\boldsymbol{x})$ comes out of the expectation because it does not depend on $\boldsymbol{h}$. Negating both sides, rearranging, and contracting part of $E_{\boldsymbol{h}\sim Q}$ into a KL-divergence terms yields:

$$\log p(\boldsymbol{x}) - \mathcal{D}\left[Q(\boldsymbol{h})\|p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{x}|\boldsymbol{h})\right] - \mathcal{D}\left[Q(\boldsymbol{h})\|P(\boldsymbol{h})\right].$$

# Rearranging

Using Bayes rule we obtain

$$E_{\boldsymbol{h}\sim Q}\left[\log p(y_i|\boldsymbol{h}, x_i)\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{h}|y_i, x_i) - \log p(\boldsymbol{h}|x_i) + \log p(y_i|x_i)\right]$$

Rearranging the terms and subtracting $E_{\boldsymbol{h}\sim Q} \log Q(\boldsymbol{h})$ from both sides gives

$$\log P(y_i|x_i) - E_{\boldsymbol{h}\sim Q}\left[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{h}|x_i, y_i)\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(y_i|\boldsymbol{h}, x_i) + \log p(\boldsymbol{h}|x_i) - \log Q(\boldsymbol{h})\right]$$

Note that $\boldsymbol{x}$ is fixed, and $Q$ can be *any* distribution, not just a distribution which does a good job mapping $\boldsymbol{x}$ to the $\boldsymbol{h}$'s that can produce $X$.

# Inferring the probability

Since we are interested in inferring $p(\boldsymbol{x})$, it makes sense to construct a $Q$ which *does* depend on $\boldsymbol{x}$, and in particular, one which makes $\mathcal{D}\left[Q(\boldsymbol{h})\|p(\boldsymbol{h}|\boldsymbol{x})\right]$ small

$$\log p(\boldsymbol{x}) - \mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x})\|p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{x}|\boldsymbol{h})\right] - \mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x})\|p(\boldsymbol{h})\right].$$

Hence, during training, it makes sense to choose a $Q$ which will make $E_{\boldsymbol{h}\sim Q}[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{h}|x_i, y_i)]$ (a $\mathcal{D}$-divergence) small, such that the right hand side is a close approximation to $\log p(y_i|y_i)$.

# Central equation of VAEs

This equation serves as the core of the variational autoencoder, and it is worth spending some time thinking about what it means.

1. The left hand side has the quantity we want to maximize, namely $\log p(\boldsymbol{x})$ plus an error term.
2. The right hand side is something we can optimize via stochastic gradient descent given the right choice of $Q$.

# Setting up SGD

So how can we perform stochastic gradient descent?
First we need to be a bit more specific about the form that $Q(\boldsymbol{h}|\boldsymbol{x})$ will take. The usual choice is to say that
$Q(\boldsymbol{h}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{h}|\mu(\boldsymbol{x};\vartheta), \Sigma(;\vartheta))$, where $\mu$ and $\Sigma$ are arbitrary deterministic functions with parameters $\vartheta$ that can be learned from data (we will omit $\vartheta$ in later equations). In practice, $\mu$ and $\Sigma$ are again implemented via neural networks, and $\Sigma$ is constrained to be a diagonal matrix.

# More on the SGD

The name variational "autoencoder" comes from the fact that $\mu$ and $\Sigma$ are "encoding" $\boldsymbol{x}$ into the latent space $\boldsymbol{h}$. The advantages of this choice are computational, as they make it clear how to compute the right hand side. The last term—$\mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x})\|p(\boldsymbol{h})\right]$—is now a KL-divergence between two multivariate Gaussian distributions, which can be computed in closed form as:

$$\mathcal{D}[\mathcal{N}(\mu_0, \Sigma_0)\|\mathcal{N}(\mu_1, \Sigma_1)] =$$
$$\frac{1}{2}\left(\mathrm{tr}\left(\Sigma_1^{-1}\Sigma_0\right) + (\mu_1 - \mu_0)^\top \Sigma_1^{-1}(\mu_1 - \mu_0) - k + \log\left(\frac{\det \Sigma_1}{\det \Sigma_0}\right.\right.$$

where $k$ is the dimensionality of the distribution.

# Simplification

In our case, this simplifies to:

$$\mathcal{D}[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)] =$$
$$\frac{1}{2}\left(\mathrm{tr}\left(\Sigma(X)\right) + \left(\mu(X)\right)^{\top}\left(\mu(X)\right) - k - \log\det\left(\Sigma(X)\right)\right).$$

# Terms to compute

The first term on the right hand side is a bit more tricky. We could use sampling to estimate $E_{z\sim Q}[\log P(X|z)]$, but getting a good estimate would require passing many samples of $z$ through $f$, which would be expensive. Hence, as is standard in stochastic gradient descent, we take one sample of $z$ and treat $\log P(X|z)$ for that $z$ as an approximation of $E_{z\sim Q}[\log P(X|z)]$. After all, we are already doing stochastic gradient descent over different values of $X$ sampled from a dataset $D$. The full equation we want to optimize is:

$$E_{X\sim D}[\log P(X) - \mathcal{D}[Q(z|X)\|P(z|X)]] = E_{X\sim D}[E_{z\sim Q}[\log P(X|z)] - \mathcal{D}[Q(z|X)\|P(z)]].$$

# Computing the gradients

If we take the gradient of this equation, the gradient symbol can be moved into the expectations. Therefore, we can sample a single value of $X$ and a single value of $z$ from the distribution $Q(z|X)$, and compute the gradient of:

$$\log P(X|z) - \mathcal{D}\left[Q(z|X)\|P(z)\right]. \tag{1}$$

We can then average the gradient of this function over arbitrarily many samples of $X$ and $z$, and the result converges to the gradient. There is, however, a significant problem $E_{z\sim Q}\left[\log P(X|z)\right]$ depends not just on the parameters of $P$, but also on the parameters of $Q$. In order to make VAEs work, it is essential to drive $Q$ to produce codes for $X$ that $P$ can reliably decode.

$$E_{X\sim D}\left[E_{\epsilon\sim\mathcal{N}(0,I)}[\log P(X|z=\mu(X)+\Sigma^{1/2}(X)*\epsilon)] - \mathcal{D}\left[Q(z|X)\|P(z)\right]\right]$$

# Code examples using Keras

Code taken from

https://keras.io/examples/generative/vae/

```
"""
Title: Variational AutoEncoder
Author: [fchollet](https://twitter.com/fchollet)
Date created: 2020/05/03
Last modified: 2023/11/22
Description: Convolutional Variational AutoEncoder (VAE) trained on MN
Accelerator: GPU
"""

"""
## Setup
"""

import os

os.environ["KERAS_BACKEND"] = "tensorflow"

import numpy as np
import tensorflow as tf
import keras
from keras import layers

"""
## Create a sampling layer
"""
```

# Code in PyTorch for VAEs

```
import torch
from torch.autograd import Variable
import numpy as np
import torch.nn.functional as F
import torchvision
from torchvision import transforms
import torch.optim as optim
from torch import nn
import matplotlib.pyplot as plt
from torch import distributions

class Encoder(torch.nn.Module):
    def __init__(self, D_in, H, latent_size):
        super(Encoder, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, H)
        self.enc_mu = torch.nn.Linear(H, latent_size)
        self.enc_log_sigma = torch.nn.Linear(H, latent_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        mu = self.enc_mu(x)
        log_sigma = self.enc_log_sigma(x)
        sigma = torch.exp(log_sigma)
        return torch.distributions.Normal(loc=mu, scale=sigma)
```