

Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen^{1,2}

Department of Physics and Center for Computing in Science Education,
University of Oslo, Norway¹

Department of Physics and Astronomy and Facility for Rare Isotope Beams,
Michigan State University, East Lansing, Michigan, USA²

April 16, 2024

© 1999-2024, Morten Hjorth-Jensen. Released under CC Attribution-NonCommercial 4.0 license

Plans for the week April 15-19, 2024

Deep generative models

1. Finalizing discussion of Boltzmann machines, implementations using TensorFlow and Pytorch
2. Discussion of other energy-based models and Langevin sampling
3. Variational Autoencoders (VAE), mathematics and code examples
4. Video of lecture
5. Whiteboard notes

Readings

1. Reading recommendation: Goodfellow et al, for VAEs see sections 20.10-20.11
2. 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
3. See Foster, chapter 7 on energy-based models at https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition/tree/main/notebooks/07_ebm/01_ebm

Reminder from last week and layout of lecture this week

1. We will present first a reminder from last week, see for example the jupyter-notebook at <https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week12/ipynb/week12.ipynb>
2. We will then discuss codes as well as other energy-based models and Langevin sampling instead of Gibbs or Metropolis sampling.
3. Thereafter we start our discussions of Variational autoencoders and Generalized adversarial networks

Code for RBMs using PyTorch for a binary-binary RBM

```
import numpy as np
import torch
import torch.utils.data
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torchvision import datasets, transforms
from torchvision.utils import make_grid , save_image
import matplotlib.pyplot as plt
```

```
batch_size = 64
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data',
        train=True,
        download = True,
        transform = transforms.Compose(
            [transforms.ToTensor()]
        ),
        batch_size=batch_size
    )
```

```
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data',
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor()]
        )
```

RBM using TensorFlow and Keras

1. 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

Codes for Energy-based models

See discussions in Foster, chapter 7 on energy-based models at
https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition/tree/main/notebooks/07_ebm/01_ebm

That notebook is based on a recent article by Du and Mordatch,
Implicit generation and modeling with energy-based models,
see <https://arxiv.org/pdf/1903.08689.pdf>.

Langevin sampling

Also called Stochastic gradient Langevin dynamics (SGLD), is sampling technique composed of characteristics from Stochastic gradient descent (SGD) and Langevin dynamics, a mathematical extension of the Langevin equation. The SGLD is an iterative optimization algorithm which uses minibatching to create a stochastic gradient estimator, as used in SGD to optimize a differentiable objective function.

Unlike traditional SGD, SGLD can be used for Bayesian learning as a sampling method. SGLD may be viewed as Langevin dynamics applied to posterior distributions, but the key difference is that the likelihood gradient terms are minibatched, like in SGD. SGLD, like Langevin dynamics, produces samples from a posterior distribution of parameters based on available data.

More on the SGLD

The SGLD uses the probability $p(\theta)$ (note that we limit ourselves to just a variable θ) and updates the **log** of this probability by initializing it through some random prior distribution, normally just a uniform distribution which takes values between $\theta \in [-1, 1]$,
The update is given by

$$\theta_{i+1} = \theta_i + \eta \nabla_{\theta} \log p(\theta_i) + \sqrt{\eta} w_i,$$

where $w_i \sim N(0, 1)$ are normally distributed with mean zero and variance one and $i = 0, 1, \dots, k$, with k the final number of iterations. The parameter η is the learning rate. The term $\sqrt{\eta} w_i$ introduces **noise** in the equation.

Code example of Langevin Sampling

In our calculations the gradient is calculated using the model we have for the probability distribution. For an energy-based model this gives us a derivative which involves the so-called positive and negative phases discussed last week.

Read more about Langevin sampling at for example

<https://www.lyndonduong.com/sgmcmc/>. This site contains a nice example of a PyTorch code which implements Langevin sampling.

Theory of Variational Autoencoders

Let us remind ourself about what an autoencoder is, see the jupyter-notebook at <https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week10/ipynb/week10.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 \mathbf{x} via a function $f(\mathbf{x}, \mathbf{W})$ (this is the encoder part) to a **so-called code part** (or intermediate part) with the result \mathbf{h}

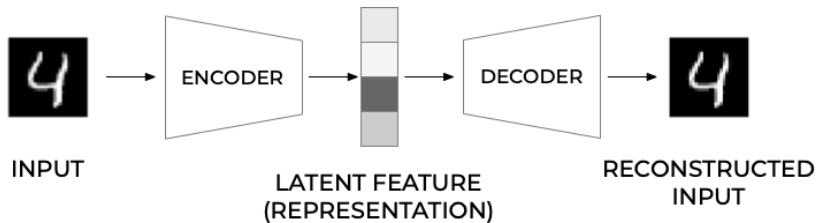
$$\mathbf{h} = f(\mathbf{x}, \mathbf{W}),$$

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

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

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 \mathbf{h} and parameters Θ as

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

for continuous variables \mathbf{h} and

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

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

Using the conditional probability

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

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

which allows us to make the dependence of \mathbf{x} on \mathbf{h} explicit by using the law of total probability. The intuition behind this approach for finding the marginal probability for \mathbf{x} is to optimize the above equations with respect to the parameters Θ . 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

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

with mean value given by the function $f(\mathbf{h}; \Theta)$ and a diagonal covariance matrix multiplied by a parameter σ^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 $p(\mathbf{x}; \Theta)$ by making $f(\mathbf{h}; \Theta)$ approach \mathbf{x} for some \mathbf{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 Θ .

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 $p(\mathbf{x})$ without performing Markov Chain Monte Carlo.

Training VAEs

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

Kullback-Leibler relative entropy (notation to be updated)

When the goal of the training is to approximate a probability distribution, as it is in generative modeling, another relevant measure is the **Kullback-Leibler divergence**, also known as the relative entropy or Shannon entropy. It is a non-symmetric measure of the dissimilarity between two probability density functions p and q . If p is the unknown probability which we approximate with q , we can measure the difference by

$$\text{KL}(p||q) = \int_{-\infty}^{\infty} p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x}.$$

Kullback-Leibler divergence and RBMs

Thus, the Kullback-Leibler divergence between the distribution of the training data $f(\mathbf{x})$ and the model marginal distribution $p(\mathbf{x}; \Theta)$ from an RBM is

$$\begin{aligned}\text{KL}(f(\mathbf{x})||p(\mathbf{x}|\Theta)) &= \int_{-\infty}^{\infty} f(\mathbf{x}) \log \frac{f(\mathbf{x})}{p(\mathbf{x}; \Theta)} d\mathbf{x} \\ &= \int_{-\infty}^{\infty} f(\mathbf{x}) \log f(\mathbf{x}) d\mathbf{x} - \int_{-\infty}^{\infty} f(\mathbf{x}) \log p(\mathbf{x}; \Theta) d\mathbf{x} \\ &= \langle \log f(\mathbf{x}) \rangle_{f(\mathbf{x})} - \langle \log p(\mathbf{x}; \Theta) \rangle_{f(\mathbf{x})} \\ &= \langle \log f(\mathbf{x}) \rangle_{data} + \langle E(\mathbf{x}) \rangle_{data} + \log Z.\end{aligned}$$

Maximizing log-likelihood

The first term is constant with respect to Θ since $f(\mathbf{x})$ is independent of Θ . Thus the Kullback-Leibler divergence is minimal when the second term is minimal. The second term is the log-likelihood cost function, hence minimizing the Kullback-Leibler divergence is equivalent to maximizing the log-likelihood.

Back to VAEs

We want to train the marginal probability with some latent variables \mathbf{h}

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

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

Using the KL divergence

In practice, for most \mathbf{h} , $p(\mathbf{x}|\mathbf{h}; \Theta)$ will be nearly zero, and hence contribute almost nothing to our estimate of $p(\mathbf{x})$.

The key idea behind the variational autoencoder is to attempt to sample values of \mathbf{h} that are likely to have produced \mathbf{x} , and compute $p(\mathbf{x})$ just from those.

This means that we need a new function $Q(\mathbf{h}|\mathbf{x})$ which can take a value of \mathbf{x} and give us a distribution over \mathbf{h} values that are likely to produce \mathbf{x} . Hopefully the space of \mathbf{h} values that are likely under Q will be much smaller than the space of all \mathbf{h} 's that are likely under the prior $p(\mathbf{h})$. This lets us, for example, compute $E_{\mathbf{h} \sim Q} p(\mathbf{x}|\mathbf{h})$ relatively easily. Note that we drop Θ from here and for notational simplicity.

Kullback-Leibler again

However, if \mathbf{h} is sampled from an arbitrary distribution with PDF $Q(\mathbf{h})$, which is not $\mathcal{N}(0, I)$, then how does that help us optimize $p(\mathbf{x})$?

The first thing we need to do is relate $E_{\mathbf{h} \sim Q} P(\mathbf{x}|\mathbf{h})$ and $p(\mathbf{x})$. We will see where Q comes from later.

The relationship between $E_{\mathbf{h} \sim Q} p(\mathbf{x}|\mathbf{h})$ and $p(\mathbf{x})$ is one of the cornerstones of variational Bayesian methods. We begin with the definition of Kullback-Leibler divergence (KL divergence or \mathcal{D}) between $p(\mathbf{h}|\mathbf{x})$ and $Q(\mathbf{h})$, for some arbitrary Q (which may or may not depend on \mathbf{x}):

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

And applying Bayes rule

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

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

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

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

Rearranging

Using Bayes rule we obtain

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

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

$$\begin{aligned} \log P(y_i | x_i) - E_{\mathbf{h} \sim Q} [\log Q(\mathbf{h}) - \log p(\mathbf{h} | x_i, y_i)] = \\ E_{\mathbf{h} \sim Q} [\log p(y_i | \mathbf{h}, x_i) + \log p(\mathbf{h} | x_i) - \log Q(\mathbf{h})] \end{aligned}$$

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

Inferring the probability

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

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

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

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 $\log p(\mathbf{x})$ 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 $Q(\mathbf{h}|\mathbf{x})$ will take. The usual choice is to say that

$Q(\mathbf{h}|\mathbf{x}) = \mathcal{N}(\mathbf{h}|\mu(\mathbf{x}; \vartheta), \Sigma(\mathbf{x}; \vartheta))$, where μ and Σ are arbitrary deterministic functions with parameters ϑ that can be learned from data (we will omit ϑ in later equations). In practice, μ and Σ are again implemented via neural networks, and Σ is constrained to be a diagonal matrix.

More on the SGD

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

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

where k is the dimensionality of the distribution.

Simplification

In our case, this simplifies to:

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

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:

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

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

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

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 MNIST

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)
```