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, Norway[1]

Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA[2]

April 23, 2024

Plans for the week April 22-26, 2024

Deep generative models

1. Discussion of project 2
2. Variational Autoencoders (VAE), Mathematics and codes, continuation from last week
3. Generative Adversarial Networks (GANs)
4. Reading recommendation:
   - 4.1 Goodfellow et al chapter 20.10-20-14
   - 4.2 Calvin Luo `https://calvinyluo.com/2022/08/26/diffusion-tutorial.html`
   - 4.3 An Introduction to Variational Autoencoders, by Kingma and Welling, see `https://arxiv.org/abs/1906.02691`
5. Video of lecture
6. Whiteboard notes

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 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" $\boldsymbol{x}$ 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)
```

What is a GAN?

A GAN is a deep neural network which consists of two networks, a so-called generator network and a discriminating network, or just discriminator. Through several iterations of generation and discrimination, the idea is that these networks will train each other, while also trying to outsmart each other.

What is a generator network?

A generator network is often a deep network which uses existing data to generate new data (from for example simulations of physical systems, imagesm video, audio and more) from randomly generated inputs, the so-called latent space. Training the network allows us to generate say new data, images etc. As an example a generator network could for example be a Boltzmann machine as discussed earlier. This machine is trained to produce for example a quantum mechanical probability distribution.
It can a simple neural network with an input layer and an output layer and a given number of hidden layers.

And what is a discriminator network?

A discriminator tries to distinguish between real data and those generated by the abovementioned generator.

Appplications of GANs

There are exteremely many applications of GANs

1. Image generation
2. Text-to-image analysis
3. Face-aging
4. Image-to-image translation
5. Video synthesis
6. High-resolution image generation
7. Completing missing parts of images and much more

Generative Adversarial Networks

Generative Adversarial Networks are a type of unsupervised machine learning algorithm proposed by Goodfellow et. al, see https://arxiv.org/pdf/1406.2661.pdf in 2014 (Read the paper first it's only 6 pages). The simplest formulation of the model is based on a game theoretic approach, *zero sum game*, where we pit two neural networks against one another. We define two rival networks, one generator g, and one discriminator d. The generator directly produces samples

$$x = g(z; \theta^{(g)}).$$

Discriminator

The discriminator attempts to distinguish between samples drawn from the training data and samples drawn from the generator. In other words, it tries to tell the difference between the fake data produced by g and the actual data samples we want to do prediction on. The discriminator outputs a probability value given by

$$d(x; \theta^{(d)}).$$

indicating the probability that x is a real training example rather than a fake sample the generator has generated.

Zero-sum game

The simplest way to formulate the learning process in a generative adversarial network is a zero-sum game, in which a function

$$v(\theta^{(g)}, \theta^{(d)}),$$

determines the reward for the discriminator, while the generator gets the conjugate reward

$$-v(\theta^{(g)}, \theta^{(d)})$$

Maximizing reward

During learning both of the networks maximize their own reward function, so that the generator gets better and better at tricking the discriminator, while the discriminator gets better and better at telling the difference between the fake and real data. The generator and discriminator alternate on which one trains at one time (i.e. for one epoch). In other words, we keep the generator constant and train the discriminator, then we keep the discriminator constant to train the generator and repeat. It is this back and forth dynamic which lets GANs tackle otherwise intractable generative problems. As the generator improves with training, the discriminator's performance gets worse because it cannot easily tell the difference between real and fake. If the generator ends up succeeding perfectly, the the discriminator will do no better than random guessing i.e. 50%.

Progression in training

This progression in the training poses a problem for the convergence criteria for GANs. The discriminator feedback gets less meaningful over time, if we continue training after this point then the generator is effectively training on junk data which can undo the learning up to that point. Therefore, we stop training when the discriminator starts outputting $1/2$ everywhere. At convergence we have

$$g^* = \operatorname*{argmin}_g \max_d v(\theta^{(g)}, \theta^{(d)}),$$

Deafault choice

The default choice for v is

$$v(\theta^{(g)}, \theta^{(d)}) = \mathbb{E}_{x \sim p_{\text{data}}} \log d(x) + \mathbb{E}_{x \sim p_{\text{model}}} \log(1 - d(x)).$$

Design of GANs

The main motivation for the design of GANs is that the learning process requires neither approximate inference (variational autoencoders for example) nor approximation of a partition function. In the case where

$$\max_{d} v(\theta^{(g)}, \theta^{(d)})$$

is convex in $\theta^{(g)}$ then the procedure is guaranteed to converge and is asymptotically consistent (Seth Lloyd on QuGANs). This is in general not the case and it is possible to get situations where the training process never converges because the generator and discriminator chase one another around in the parameter space indefinitely.

More references

A much deeper discussion on the currently open research problem of GAN convergence is available from https://www.deeplearningbook.org/contents/generative_models.html. To anyone interested in learning more about GANs it is a highly recommended read. Direct quote: **In this best-performing formulation, the generator aims to increase the log probability that the discriminator makes a mistake, rather than aiming to decrease the log probability that the discriminator makes the correct prediction.** Another interesting read can be found at https://arxiv.org/abs/1701.00160.

Writing Our First Generative Adversarial Network

This part is best seen using the jupyter-notebook.

Let us implement a GAN in tensorflow. We will study the performance of our GAN on the MNIST dataset. This code is based on and adapted from the Google tutorial at

https://www.tensorflow.org/tutorials/generative/dcgan

First we import our libraries

```
import os
import time
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model
```

Next we define our hyperparameters and import our data the usual way

```
BUFFER_SIZE = 60000
BATCH_SIZE = 256
EPOCHS = 30

data = tf.keras.datasets.mnist.load_data()
(train_images, train_labels), (test_images, test_labels) = data
train_images = np.reshape(train_images, (train_images.shape[0],
                                         28,
```