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 16, 2024

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)
4. Generative Adversarial Networks (GANs)

Readings

1. Reading recommendation: Goodfellow et al chapter 20.10-20-14
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

Reminder from last week and layout of lecture this week

1. We will present first a short 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

```
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

Energy-based models and Langevin sampling

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.

Theory of Variational Autoencoders

Notes to be added

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)
```

Generative Adversarial Networks

Generative Adversarial Networks are a type of unsupervised machine learning algorithm proposed by Goodfellow et. al 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

$$-v(\theta^{(g)}, \theta^{(d)}) \tag{1}$$

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

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

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,
```