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]

May 15

Plans for the week of May 12-16, 2025

Generative models

1. Summary from last week about diffusion models, mainly discussion of codes (see jupyter-notebook from the week of May 5-9)
2. Generative Adversarial Networks, see `https://lilianweng.github.io/posts/2017-08-20-gan/` for nice overview
3. A summary of the course will be made available in the form of a video. The summary material can also be viewed at `https://github.com/CompPhysics/AdvancedMachineLearning/tree/main/doc/pub/week17` in various formats (pdf, jupyter-notebook or html formats).

Readings

1. Reading recommendation: Goodfellow et al, for GANs see sections 20.10-20.11
2. For codes and background, see Raschka et al, Machine Learning with PyTorch and Scikit-Learn, chapter 17, see `https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch17` for codes
3. Babcock and Bali, Generative AI with Python and TensorFlow2, chapter 6 and codes at `https://github.com/raghavbali/generative_ai_with_tensorflow/blob/master/Chapter_6/conditional_gan.ipynb`

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.
In its simplest version, the two networks could be two standard neural networks with a given number of hidden of hidden layers and parameters to train. The generator we have trained can then be used to produce new images.

Labeling the networks

For a GAN we have:

1. a discriminator D estimates the probability of a given sample coming from the real dataset. It attempts at discriminating the trained data by the generator and is optimized to tell the fake samples from the real ones (our data set). We say a discriminator tries to distinguish between real data and those generated by the abovementioned generator.
2. a generator G outputs synthetic samples given a noise variable input z (z brings in potential output diversity). It is trained to capture the real data distribution in order to generate samples that can be as real as possible, or in other words, can trick the discriminator to offer a high probability.

At the end of the training, the generator can be used to generate for example new images. In this sense we have trained a model which can produce new samples. We say that we have implicitely defined a probability.

Which data?

GANs are generally a form of unsupervised machine learning, although they also incorporate aspects of supervised learning. Internally the discriminator sets up a supervised learning problem. Its goal is to learn to distinguish between the two classes of generated data and original data. The generator then considers this classification problem and tries to find adversarial examples, that is samples which will be misclassified by the discriminator.

Semi-supervised learning

One can also design GAN architectures which work in a semi-supervised learning setting. A semi-supervised learning environment includes both labeled and unlabeled data. See https://proceedings.neurips.cc/paper_files/paper/2016/file/8a3363abe792db2d8761d6403605aeb7-Paper.pdf for a further discussion.

Thus, GANs can be used both on labeled and on unlabeled data and are used in three most commonly used contexts, that is

1. with labeled data (supervised training)
2. with unlabeled data (unsupervised learning)
3. a with a mix labed and unlabeled data

Improving functionalities

These two models compete against each other during the training process: the generator G is trying hard to trick the discriminator, while the critic model D is trying hard not to be cheated. This interesting zero-sum game between two models motivates both to improve their functionalities.

More on Generative Adversarial Networks

Generative adversarial networks have shown great results in many generative tasks to replicate the real-world rich content such as images, human language, and music. It is inspired by game theory: two models, a generator and a discriminator, are competing with each other while making each other stronger at the same time. However, it is rather challenging to train a GANs model, training instability or failure to converge.

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

Setup of the GAN

We define a probability $p_{\boldsymbol{h}}$ which is used by the generator. Usually it is given by a uniform distribution over the input $\boldsymbol{h}$. Thereafter we define the distribution of the generator which we want to train, p_g This is the generator's distribution over the data $\boldsymbol{x}$. Finally, we have the distribution p_r over the real sample $\boldsymbol{x}$

Generative Adversarial Networks

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.

Improving functionalities

These two models compete against each other during the training process: the generator G is trying hard to trick the discriminator, while the critic model D is trying hard not to be cheated. This interesting zero-sum game between two models motivates both to improve their functionalities.

Setup of the GAN

We define a probability $p_{\boldsymbol{h}}$ which is used by the generator. Usually it is given by a uniform distribution over the input input $\boldsymbol{h}$. Thereafter we define the distribution of the generator which we want to train, p_g This is the generator's distribution over the data $\boldsymbol{x}$. Finally, we have the distribution p_r over the real sample $\boldsymbol{x}$

Optimization part

On one hand, we want to make sure the discriminator D's decisions over real data are accurate by maximizing $\mathbb{E}_{\boldsymbol{x} \sim p_r(\boldsymbol{x})}[\log D(\boldsymbol{x})]$.
Meanwhile, given a fake sample $G(\boldsymbol{h}), \boldsymbol{h} \sim p_{\boldsymbol{h}}(\boldsymbol{h})$, the discriminator is expected to output a probability, $D(G(\boldsymbol{h}))$, close to zero by maximizing $\mathbb{E}_{\boldsymbol{h} \sim p_{\boldsymbol{h}}(\boldsymbol{h})}[\log(1 - D(G(\boldsymbol{h})))]$.
On the other hand, the generator is trained to increase the chances of D producing a high probability for a fake example, thus to minimize $\mathbb{E}_{\boldsymbol{h} \sim p_{\boldsymbol{h}}(\boldsymbol{h})}[\log(1 - D(G(\boldsymbol{h})))]$.

Minimax game

When combining both aspects together, D and G are playing a **minimax game** in which we should optimize the following loss function:

$$\begin{aligned}\min_G \max_D L(D,G) &= \mathbb{E}_{\boldsymbol{x}\sim p_r(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{h}\sim p_{\boldsymbol{h}}(\boldsymbol{h})}[\log(1 - D(G(\boldsymbol{h})))] \\ &= \mathbb{E}_{\boldsymbol{x}\sim p_r(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g(\boldsymbol{x})}[\log(1 - D(\boldsymbol{x})]\end{aligned}$$

where $\mathbb{E}_{\boldsymbol{x}\sim p_r(\boldsymbol{x})}[\log D(\boldsymbol{x})]$ has no impact on G during gradient descent updates.

Optimal value for D

Now we have a well-defined loss function. Let's first examine what is the best value for D.

$$L(G, D) = \int_{\boldsymbol{x}} \Big(p_r(\boldsymbol{x}) \log(D(\boldsymbol{x})) + p_g(\boldsymbol{x}) \log(1 - D(\boldsymbol{x})) \Big) dx$$

Best value of D

Since we are interested in what is the best value of $D(\boldsymbol{x})$ to maximize $L(G, D)$, let us label

$$\tilde{\boldsymbol{x}} = D(\boldsymbol{x}), A = p_r(\boldsymbol{x}), B = p_g(\boldsymbol{x})$$

Ignore integral

And then what is inside the integral (we can safely ignore the integral because $\boldsymbol{x}$ is sampled over all the possible values) is:

$$\begin{aligned} f(\tilde{\boldsymbol{x}}) &= A \log \tilde{\boldsymbol{x}} + B \log (1 - \tilde{\boldsymbol{x}}) \\ \frac{df(\tilde{\boldsymbol{x}})}{d\tilde{\boldsymbol{x}}} &= A \frac{1}{\tilde{\boldsymbol{x}}} - B \frac{1}{1 - \tilde{\boldsymbol{x}}} \\ &= \frac{A - (A + B)\tilde{\boldsymbol{x}}}{\tilde{\boldsymbol{x}}(1 - \tilde{\boldsymbol{x}})}. \end{aligned}$$

Best values

Thus, if we set $\frac{df(\tilde{\boldsymbol{x}})}{d\tilde{\boldsymbol{x}}} = 0$, we get the best value of the discriminator: $D^*(\boldsymbol{x}) = \tilde{\boldsymbol{x}}^* = \frac{A}{A+B} = \frac{p_r(\boldsymbol{x})}{p_r(\boldsymbol{x})+p_g(\boldsymbol{x})} \in [0,1]$. Once the generator is trained to its optimal, p_g gets very close to p_r. When $p_g = p_r$, $D^*(\boldsymbol{x})$ becomes $1/2$. We will observe this when running the code below here.
When both G and D are at their optimal values, we have $p_g = p_r$ and $D^*(\boldsymbol{x}) = 1/2$ and the loss function becomes:

$$\begin{aligned} L(G, D^*) &= \int_{\boldsymbol{x}} \Big(p_r(\boldsymbol{x}) \log(D^*(\boldsymbol{x})) + p_g(\boldsymbol{x}) \log(1 - D^*(\boldsymbol{x})) \Big) d\boldsymbol{x} \\ &= \log \frac{1}{2} \int_{\boldsymbol{h}} p_r(\boldsymbol{x}) d\boldsymbol{x} + \log \frac{1}{2} \int_{\boldsymbol{x}} p_g(\boldsymbol{x}) d\boldsymbol{x} \\ &= -2 \log 2 \end{aligned}$$

What does the Loss Function Represent?

The JS divergence between p_r and p_g can be computed as:

$$\begin{aligned}
D_{JS}(p_r \| p_g) =& \frac{1}{2} D_{KL}(p_r || \frac{p_r + p_g}{2}) + \frac{1}{2} D_{KL}(p_g || \frac{p_r + p_g}{2}) \\
=& \frac{1}{2}\left(\log 2 + \int_x p_r(\boldsymbol{x}) \log \frac{p_r(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x} \right) + \\
& \frac{1}{2}\left(\log 2 + \int_x p_g(\boldsymbol{x}) \log \frac{p_g(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x} \right) \\
=& \frac{1}{2}\left(\log 4 + L(G, D^*) \right)
\end{aligned}$$

What does the loss function quantify?

We have

$$L(G, D^*) = 2D_{JS}(p_r \| p_g) - 2\log 2.$$

Essentially the loss function of a GAN quantifies the similarity between the generative data distribution p_g and the real sample distribution p_r by JS divergence when the discriminator is optimal. The best G^* that replicates the real data distribution leads to the minimum $L(G^*, D^*) = -2\log 2$ which is aligned with the equations above.

Problems with GANs

Although GANs have achieved great success in the generation of realistic images, the training is not easy; The process is known to be slow and unstable.

Hard to reach equilibrium.

Two models are trained simultaneously to an equilibrium to a two-player non-cooperative game. However, each model updates its cost independently with no respect to another player in the game. Updating the gradient of both models concurrently cannot guarantee a convergence.

Vanishing Gradient

When the discriminator is perfect, we are guaranteed with $D(\boldsymbol{x}) = 1, \forall \boldsymbol{x} \in p_r$ and $D(\boldsymbol{x}) = 0, \forall \boldsymbol{x} \in p_g$.
Then, the loss function L falls to zero and we end up with no gradient to update the loss during learning iterations. One can encouter situations where the discriminator gets better and the gradient vanishes fast.
As a result, training GANs may face the following problems

1. If the discriminator behaves badly, the generator does not have accurate feedback and the loss function cannot represent the real data
2. If the discriminator does a great job, the gradient of the loss function drops down to close to zero and the learning can become slow

Improved GANs

One of the solutions to improved GANs training, is the introduction of what is called the Wasserstein diatance, which is a way to compute the difference/distance between two probability distributions. For those interested in reading more, we recommend for example chapter 17 of Rashcka's et al textbook, Machine Learning with PyTorch and Scikit-Learn, chapter 17, see `https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch17`

For a definition of the Wasserstein distance, see for example `https://arxiv.org/pdf/2103.01678`

Writing Our First Generative Adversarial Network

This part is best seen using the jupyter-notebook. We follow here closely the code developed by Raschka et al from chapter 17 of their textbook, see `https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch17` for codes.

Code elements

```python
import torch
print(torch.__version__)
print("GPU Available:", torch.cuda.is_available())

if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = "cpu"

import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

Setting up the GAN

```python
## define a function for the generator:
def make_generator_network(
        input_size=20,
        num_hidden_layers=1,
        num_hidden_units=100,
        num_output_units=784):
    model = nn.Sequential()
    for i in range(num_hidden_layers):
        model.add_module(f'fc_g{i}',
                         nn.Linear(input_size,
                                   num_hidden_units))
        model.add_module(f'relu_g{i}',
                         nn.LeakyReLU())
        input_size = num_hidden_units
    model.add_module(f'fc_g{num_hidden_layers}',
                     nn.Linear(input_size, num_output_units))
    model.add_module('tanh_g', nn.Tanh())
    return model

## define a function for the discriminator:
def make_discriminator_network(
        input_size,
        num_hidden_layers=1,
        num_hidden_units=100,
        num_output_units=1):
    model = nn.Sequential()
    for i in range(num_hidden_layers):
        model.add_module(f'fc_d{i}',
```

Printing the model

```
image_size = (28, 28)
z_size = 20

gen_hidden_layers = 1
gen_hidden_size = 100
disc_hidden_layers = 1
disc_hidden_size = 100

torch.manual_seed(1)

gen_model = make_generator_network(
    input_size=z_size,
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size))

print(gen_model)

disc_model = make_discriminator_network(
    input_size=np.prod(image_size),
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size)

print(disc_model)
```

Defining the training set

```python
import torchvision
from torchvision import transforms


image_path = './'
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5), std=(0.5)),
])
mnist_dataset = torchvision.datasets.MNIST(root=image_path,
                                           train=True,
                                           transform=transform,
                                           download=True)

example, label = next(iter(mnist_dataset))
print(f'Min: {example.min()} Max: {example.max()}')
print(example.shape)
```

Defining the training set, part 2

```
def create_noise(batch_size, z_size, mode_z):
    if mode_z == 'uniform':
        input_z = torch.rand(batch_size, z_size)*2 - 1
    elif mode_z == 'normal':
        input_z = torch.randn(batch_size, z_size)
    return input_z


from torch.utils.data import DataLoader


batch_size = 32
dataloader = DataLoader(mnist_dataset, batch_size, shuffle=False)
input_real, label = next(iter(dataloader))
input_real = input_real.view(batch_size, -1)

torch.manual_seed(1)
mode_z = 'uniform'  # 'uniform' vs. 'normal'
input_z = create_noise(batch_size, z_size, mode_z)

print('input-z -- shape:', input_z.shape)
print('input-real -- shape:', input_real.shape)

g_output = gen_model(input_z)
print('Output of G -- shape:', g_output.shape)

d_proba_real = disc_model(input_real)
d_proba_fake = disc_model(g_output)
```

Training the GAN

```python
loss_fn = nn.BCELoss()

## Loss for the Generator
g_labels_real = torch.ones_like(d_proba_fake)
g_loss = loss_fn(d_proba_fake, g_labels_real)
print(f'Generator Loss: {g_loss:.4f}')

## Loss for the Discriminator
d_labels_real = torch.ones_like(d_proba_real)
d_labels_fake = torch.zeros_like(d_proba_fake)

d_loss_real = loss_fn(d_proba_real, d_labels_real)
d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
print(f'Discriminator Losses: Real {d_loss_real:.4f} Fake {d_loss_fake
```

More on training

```
batch_size = 64

torch.manual_seed(1)
np.random.seed(1)

## Set up the dataset
mnist_dl = DataLoader(mnist_dataset, batch_size=batch_size,
                      shuffle=True, drop_last=True)

## Set up the models
gen_model = make_generator_network(
    input_size=z_size,
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size)).to(device)

disc_model = make_discriminator_network(
    input_size=np.prod(image_size),
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size).to(device)

## Loss function and optimizers:
loss_fn = nn.BCELoss()
g_optimizer = torch.optim.Adam(gen_model.parameters())
d_optimizer = torch.optim.Adam(disc_model.parameters())

## Train the discriminator
def d_train(x):
```

Visualizing

```
import itertools


fig = plt.figure(figsize=(16, 6))

## Plotting the losses
ax = fig.add_subplot(1, 2, 1)

plt.plot(all_g_losses, label='Generator loss')
half_d_losses = [all_d_loss/2 for all_d_loss in all_d_losses]
plt.plot(half_d_losses, label='Discriminator loss')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Loss', size=15)

## Plotting the outputs of the discriminator
ax = fig.add_subplot(1, 2, 2)
plt.plot(all_d_real, label=r'Real: $D(\mathbf{x})$')
plt.plot(all_d_fake, label=r'Fake: $D(G(\mathbf{z}))$')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Discriminator output', size=15)

#plt.savefig('figures/ch17-gan-learning-curve.pdf')
plt.show()

selected_epochs = [1, 2, 4, 10, 50, 100]
fig = plt.figure(figsize=(10, 14))
```

Calculating scores

```
import math


def distance(X, Y, sqrt):
    nX = X.size(0)
    nY = Y.size(0)
    X = X.view(nX,-1).cuda()
    X2 = (X*X).sum(1).resize_(nX,1)
    Y = Y.view(nY,-1).cuda()
    Y2 = (Y*Y).sum(1).resize_(nY,1)

    M = torch.zeros(nX, nY)
    M.copy_(X2.expand(nX,nY) + Y2.expand(nY,nX).transpose(0,1) - 2*tor

    del X, X2, Y, Y2

    if sqrt:
        M = ((M+M.abs())/2).sqrt()

    return M

def mmd(Mxx, Mxy, Myy, sigma) :
    scale = Mxx.mean()
    Mxx = torch.exp(-Mxx/(scale*2*sigma*sigma))
    Mxy = torch.exp(-Mxy/(scale*2*sigma*sigma))
    Myy = torch.exp(-Myy/(scale*2*sigma*sigma))
    a = Mxx.mean()+Myy.mean()-2*Mxy.mean()
    mmd = math.sqrt(max(a, 0))
```

More codes

1. For codes and background, see Raschka et al, Machine Learning with PyTorch and Scikit-Learn, chapter 17, see `https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch17` for codes
2. Babcock and Bali, Generative AI with Python and TensorFlow2, chapter 6 and codes at `https://github.com/raghavbali/generative_ai_with_tensorflow/blob/master/Chapter_6/conditional_gan.ipynb`
3. See also Foster's text Generative Deep Learning and chapter 4 with codes at `https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition/tree/main/notebooks/04_gan`