CSCI 1051 Homework 3

January 27, 2023

Submission Instructions

Please upload your solutions by **5pm Friday January 27, 2023.** Remember you have 24 hours no-questions-asked *combined* lateness across all assignments.

- You are encouraged to discuss ideas and work with your classmates. However, you **must acknowledge** your collaborators at the top of each solution on which you collaborated with others and you **must write** your solutions independently.
- Your solutions to theory questions must be typeset in LaTeX or markdown. I strongly recommend uploading the source LaTeX (found here) to Overleaf for editing.
- Your solutions to coding questions must be written in a Jupyter notebook. I strongly suggest working with colab as we do in the demos.
- You should submit your solutions as a **single PDF** via the assignment on Canvas.

Problem 1 (from January 23)

We calculated the policy gradient for the general reinforcement learning problem in class. For this problem, your task is to calculate the policy gradient in the special case where the trajectory is a single action. That is, $\tau = (a_i)$ where $i \in \{1, 2, \ldots, k\} = [k]$ and k is the number of actions. Further, assume that there is only one state and the reward for action a_i is always R_i.

Define the policy π as $\pi(a) = \text{softmax}(\theta_i)$ for $i \in [k]$ where $\theta_i \in \mathbb{R}$ is a scalar parameter encoding the value of action a_i. Suppose our learning rate is η. First, show that if action a_i is sampled, then the change in the parameters in REINFORCE is given by

$$\theta_i \leftarrow \theta_i + \eta R_i(1 - \pi(a_i)).$$

Hint: The answer uses the chain rule and log rules.

Second, explain the dynamics of the above change in weights. It may help to think about when the update is large.

Solution to Problem 1

The policy gradient for the general reinforcement learning in class was given by the equation:

$$\theta \leftarrow \theta + \eta R_(\tau) \frac{\delta}{\delta \theta} log \pi(\tau)$$

Starting with the given equation, we can plug in the assumptions that $\tau = (a_i)$ and that the reward for action a_i is always R_i. Also, since $\pi(a) = \text{softmax}(\theta_i)$, we get:

$$\theta_i \leftarrow \theta_i + \eta R(\tau) \frac{\delta}{\delta \theta_i} \log(softmax(\theta_i))$$

Simplifying the derivative part of the update, we get the following:

$$\frac{\delta}{\delta \theta_i} log(softmax(\theta_i)) = \frac{1}{softmax(\theta_i)} * \frac{\delta}{\delta \theta_i} \frac{e^{\theta_i}}{\sum_{j=1}^{k} e^{\theta_j}}$$

Rewriting the derivative of softmax(θ_i):

$$\frac{\delta}{\delta \theta_i} softmax(\theta_i) = \frac{\delta}{\delta \theta_i} e^{\theta_i} (\sum_{j=1}^{k} e^{\theta_j})^{-1}$$

Now, we can apply the power rule and the partial derivative chain rule. Also, when computing $\frac{\delta}{\delta \theta_i}$ of the sum, the result is e^{θ_i} because θ_i is a term included within the sum.

$$\frac{\delta}{\delta \theta_i} softmax(\theta_i) = e^{\theta_i} (\sum_{j=1}^{k} e^{\theta_j})^{-1} + (\frac{-e^{\theta_i}}{(\sum_{j=1}^{k} e^{\theta_j})^2}) * e^{\theta_i}$$

We can now plug this back into the equation earlier and simplify:

$$\frac{\delta}{\delta \theta_i} log(softmax(\theta_i)) = \frac{1}{softmax(\theta_i)} * (softmax(\theta_i) - \frac{e^{\theta_i 2}}{(\sum_{j=1}^{k} e^{\theta_j})^2})$$

Distributing and canceling out terms, we get:

$$1 - softmax(\theta_i)$$

Therefore, when action a_i is sampled, then the change in the parameters in REINFORCE is given by:

$$\theta_i \leftarrow \theta_i + \eta R_i (1 - \pi(a_i)).$$

Now, going into the dynamics of the above change in weights, the update would be positive when the reward is positive and negative when the reward is negative. When the chosen action has a lower probability, the update is larger, and vice versa. Therefore, when the update is large, the algorithm is making a large adjustment to the parameters to change the policy, meaning that it can more accurately predict the rewards based on different actions.

Problem 2 (from January 24)

Suppose we are building a deep Q-learning neural network to play your favorite game. We are trying to decide what rewards to assign for different actions. There are two options:

1. We receive reward r_t^a for taking action a_t in state s_t.
2. We receive reward $r_t^b = r_t^a + \delta$ for taking action a_t in state s_t. Here, δ is a constant offset.

Is there a difference in the the Q learning algorithm (described by the pseudo code in class) if we use option b instead of option a? The answer is yes! Your task is to show why.

Recall

$$G_t = \sum_{i=0}^{\infty} r_{t+i}\gamma^i$$

and

$$Q(s,a) = \mathbb{E}[G_t | s_t = s, a_t = a].$$

You should:

- Write out G_t^a and G_t^b (these are the gains when we use the rewards r_t^a and reward r_t^b, respectively).
- Then write out Q_t^b in terms of Q_t^a (these are the Q values when we use r_t^a and reward r_t^b, respectively). You should conclude that $Q^b(s,a) = Q^a(s,a) + C$ for some constant C. Your job is to find this constant!
- Now see if there is a difference in the Q learning algorithm if we use Q_t^b instead of Q_t^a. Referring to the pseudo code for the Q learning algorithm from class, notice that we use the Q function twice (once in step 1 and once in step 3). You should check if there is a difference in both cases.

Solution to Problem 2

The two different expressions for the gain G_t are the following:

$$G_t^a = \sum_{i=0}^{\infty} \gamma^i r_{t+i}^a \tag{1}$$

$$G_t^b = \sum_{i=0}^{\infty} \gamma^i (r_{t+i}^b + \delta) \tag{2}$$

The difference in the two G_t expressions is the δ term, which is the value difference for the two action options.

Now, rewriting Q_t^b in terms of Q_t^a:

$$Q^b(s,a) = \mathbb{E}[G_t^b | s_t = s, a_t = a]$$

$$Q^b(s,a) = \mathbb{E}[\sum_{i=0}^{\infty} \gamma^i (r_{t+i} + \delta) | s_t = s, a_t = a]$$

Pulling δ out of the equation because it is a constant, we get:

$$Q^b(s,a) = \delta \sum_{i=0}^{\infty} \gamma^i + \mathbb{E}[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s, a_t = a]$$

Now, using G_t^a as defined above, we get:

$$Q^b(s,a) = \delta \sum_{i=0}^{\infty} \gamma^i + Q^a(s,a)$$

Since $\delta \sum_{i=0}^{\infty} \gamma^i$ is a geometric sum, we can now simplify to the following:

$$Q^b(s,a) = \frac{\delta}{1-\gamma} + Q^a(s,a)$$

Therefore, when determining is there is a difference in the Q-learning algorithm if we use Q_t^b instead of Q_t^a, we have to see which two places the Q function was used. We used Q when choosing an action and when updating $Q(s,a)$. In the first scenario, there would be no difference because the maximum value action would be chosen regardless. In the second scenario, there would be a difference because the Q_t^b values have a larger update.

Problem 3 (from January 25)

In this problem, your task is to modify the demo so that a) we're generating images of FashionMNIST (instead of MNIST) and b) we're using a GAN (instead of a Conditional GAN).

Comment on the difference in quality of the fake images from your FashionMNIST GAN and the MNIST Conditional GAN we wrote in class.

Solution to Problem 3

The quality of the fake images produced from the MNIST Conditional GAN in class was worse than the quality of the fake images produced from the FashionMNIST GAN.

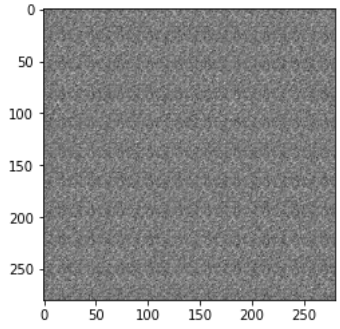
```
import torch
from torchvision import datasets
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
import numpy as np
import os
from matplotlib.pyplot import imsave
import matplotlib.pyplot as plt
import matplotlib.image as mpimg


# A transform to convert the images to tensor and normalize their RGB values
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=[0.5], std=[0.5])]
)

data = datasets.FashionMNIST(root='../data/', train=True, transform=transform, download=True)

batch_size = 64
data_loader = DataLoader(dataset=data, batch_size=batch_size, shuffle=True, drop_last=True)


def get_sample_image(G, DEVICE, n_noise=100):
    img = np.zeros([280, 280])
    for j in range(10):
        z = torch.randn(10, n_noise).to(DEVICE)
        y_hat = G(z).view(10, 28, 28)
        result = y_hat.cpu().data.numpy()
        img[j*28:(j+1)*28] = np.concatenate([x for x in result], axis=-1)
    return img


class Generator(nn.Module):
    def __init__(self, input_size=100,image_size=28*28):
        super(Generator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 128), # auxillary dimension for label
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, image_size),
            nn.Tanh()
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)
        #v = torch.cat((x, 1)) # v: [input, label] concatenated vector
        y_ = self.network(x)
        y_ = y_.view(x.size(0), 1, 28, 28)
        return y_
```

```
class Discriminator(nn.Module):
    def __init__(self, input_size=28*28, num_output=1):
        super(Discriminator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, num_output),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)
        # v = torch.cat((x), 1) # v: [input, label] concatenated vector
        y_ = self.network(x)
        return y_


MODEL_NAME = 'GAN'
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

D = Discriminator().to(DEVICE)
G = Generator().to(DEVICE)

max_epoch = 10
step = 0
n_noise = 100 # size of noise vector

criterion = nn.BCELoss()
D_opt = torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
G_opt = torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))

# We will denote real images as 1s and fake images as 0s
# This is why we needed to drop the last batch of the data loader
D_labels = torch.ones([batch_size, 1]).to(DEVICE) # Discriminator label: real
D_fakes = torch.zeros([batch_size, 1]).to(DEVICE) # Discriminator Label: fake


# a directory to save the generated images
if not os.path.exists('samples'):
    os.makedirs('samples')

for epoch in range(max_epoch):
    for idx, (images, _) in enumerate(data_loader):
        # Training Discriminator
        x = images.to(DEVICE)
        x_outputs = D(x) # input includes labels
        D_x_loss = criterion(x_outputs, D_labels) # Discriminator loss for real images

        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z)) # input to both generator and discriminator includes labels
        D_z_loss = criterion(z_outputs, D_fakes) # Discriminator loss for fake images
        D_loss = D_x_loss + D_z_loss # Total Discriminator loss

        D.zero_grad()
        D_loss.backward()
        D_opt.step()

        # Training Generator
        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z))
        G_loss = -1 * criterion(z_outputs, D_fakes) # Generator loss is negative disciminator loss

        G.zero_grad()
        G_loss.backward()
        G_opt.step()

        if step % 500 == 0:
            print('Epoch: {}/{}, Step: {}, D Loss: {}, G Loss: {}'.format(epoch, max_epoch, step, D_loss.item(), G_loss.item()))

        if step % 1000 == 0:
            G.eval()
            img = get_sample_image(G, DEVICE, n_noise)
            imsave('samples/{}_step{}.jpg'.format(MODEL_NAME, str(step).zfill(3)), img, cmap='gray')
            G.train()
        step += 1
```

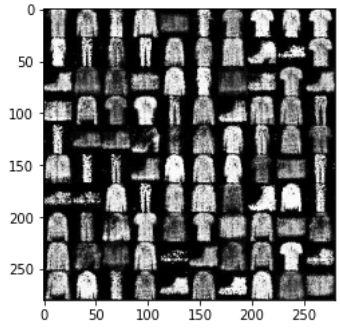
```
Epoch: 0/10, Step: 0, D Loss: 1.4120373725891113, G Loss: -0.7062667608261108
Epoch: 0/10, Step: 500, D Loss: 1.1913490295410156, G Loss: -0.5061103701591492
Epoch: 1/10, Step: 1000, D Loss: 1.1233105659484863, G Loss: -0.593504786491394
Epoch: 1/10, Step: 1500, D Loss: 1.2383441925048828, G Loss: -0.45383721590042114
Epoch: 2/10, Step: 2000, D Loss: 1.2721788883209229, G Loss: -0.4557284116744995
Epoch: 2/10, Step: 2500, D Loss: 1.3356597423553467, G Loss: -0.5755120515823364
Epoch: 3/10, Step: 3000, D Loss: 1.331081748008728, G Loss: -0.5234083533287048
Epoch: 3/10, Step: 3500, D Loss: 1.38785719871521, G Loss: -0.627362847328186
Epoch: 4/10, Step: 4000, D Loss: 1.3459150791168213, G Loss: -0.5192853808403015
Epoch: 4/10, Step: 4500, D Loss: 1.3248074054718018, G Loss: -0.5603094696998596
Epoch: 5/10, Step: 5000, D Loss: 1.3582115173339844, G Loss: -0.6155641078948975
Epoch: 5/10, Step: 5500, D Loss: 1.2880699634552002, G Loss: -0.5496246814727783
Epoch: 6/10, Step: 6000, D Loss: 1.3480061292648315, G Loss: -0.6984512805938721
Epoch: 6/10, Step: 6500, D Loss: 1.2495630979537964, G Loss: -0.5555623173713684
Epoch: 7/10, Step: 7000, D Loss: 1.299135446548462, G Loss: -0.5941945314407349
Epoch: 8/10, Step: 7500, D Loss: 1.3808751106262207, G Loss: -0.5168596506118774
Epoch: 8/10, Step: 8000, D Loss: 1.3835018873214722, G Loss: -0.6137067675590515
Epoch: 9/10, Step: 8500, D Loss: 1.370305061340332, G Loss: -0.6274106502532959
Epoch: 9/10, Step: 9000, D Loss: 1.32705557346344, G Loss: -0.6069644689559937
```

Now let's plot these images. At first, the generator just produces noise (as we expect).

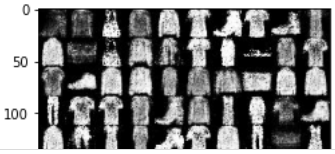
```
img = mpimg.imread('samples/GAN_step000.jpg')
imgplot = plt.imshow(img)
plt.show()
```




```
img = mpimg.imread('samples/GAN_step5000.jpg')
imgplot = plt.imshow(img)
plt.show()
```




```
img = mpimg.imread('samples/GAN_step9000.jpg')
imgplot = plt.imshow(img)
plt.show()
```


0
50
100

Problem 4 (from January 26)

In class, we considered a *symmetric* matrix $\bar{\mathbf{A}}$. We proved that we can write it as

$$\sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top$$

where $0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n \leq 1$ are the eigenvalues and $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ are the corresponding eigenvectors. Remember eigenvectors are *orthonormal*: $\mathbf{v}_i^\top \mathbf{v}_j = 1$ if $i = j$ and 0 otherwise.

In the process of analyzing the Frobenius norm of $\bar{\mathbf{A}}$, we wanted to show that the eigenvalues of $\bar{\mathbf{A}}^\top \bar{\mathbf{A}}$ are $\lambda_1^2 \leq \lambda_2^2 \leq \ldots \lambda_n^2$. Your homework is to show this is true. **Hint**: Use our formulation of $\bar{\mathbf{A}}$ in terms of its eigenvalues and eigenvectors to compute $\bar{\mathbf{A}}^\top \bar{\mathbf{A}}$.

Solution to Problem 4

In this proof, we want to show that $\bar{\mathbf{A}}^\top \bar{\mathbf{A}} = \sum_{i=1}^{n} \mathbf{v}_i \mathbf{v}_i^\top \lambda_i^2$

To start, since we know that $\bar{\mathbf{A}}$ is $\sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top$,

$$\bar{\mathbf{A}}^\top \bar{\mathbf{A}} = (\sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top)(\sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top)$$

We can now pull out the sums:

$$\bar{\mathbf{A}}^\top \bar{\mathbf{A}} = \sum_{i=1}^{n} \sum_{j=1}^{n} (v_i v_i^\top \lambda_i)(v_j v_j^\top \lambda_j)$$

Every term is 0 except when $i = j$, therefore when $i = j$, we get the following:

$$\bar{\mathbf{A}}^\top \bar{\mathbf{A}} = \sum_{i=1}^{n} v_i v_i^\top v_i v_i^\top \lambda_i \lambda_i$$

In this equation, $v_i^\top v_i$ are the only two vectors that vector multiplication can be computed with as an 1xn vector is being multiplied by an nx1 vector resulting in a 1x1. The two vectors are orthonormal, so they cancel out to 1 as described in class. Thus we get the following:

$$\bar{\mathbf{A}}^\top \bar{\mathbf{A}} = v_i v_i^\top \lambda_i^2$$