Deep Learning Homework_3

Jinrui Zhang

May 5, 2024

Problem 1

a.

Given

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

And from the policy gradient, we know:

$$\theta_{i+1} = \theta_i + \eta_i R_i \frac{\partial}{\partial \theta} log\pi(a_i)$$

So,

$$\Delta\theta = \eta R_i \frac{\partial}{\partial \theta} log\pi(a_i)$$
$$\Delta\theta = \eta R_i \frac{\partial}{\partial \theta} log(\frac{e^{\theta_i}}{\sum_{j=1}^{k} e^{\theta_j}})$$
$$\Delta\theta = \eta R_i \frac{\partial}{\partial \theta} (\theta_i - log \sum_{j=1}^{k} e^{\theta_j})$$
$$\Delta\theta = \eta R_i (1 - \frac{e^{\theta_i}}{\sum_{j=1}^{k} e^{\theta_j}})$$
$$\Delta\theta = \eta R_i (1 - \pi(a_i))$$

b.

If constant step sizes are used, then the update rule will lead to unstable training because a policy gradient with $softmax$ can take exponential time to converge.

When the value of $\pi(a_i)$ is small, we are far from optimal point and thus $1 - \pi(a_i)$ is large and we end up taking larger steps and when $\pi(a_i)$ is large we are close to optimal point thereby making $1 - \pi(a_i)$ is small thus ensuring that we take smaller steps. Or we could apply entropy regularization when using $softmax$

Problem 2

a) Determine the saddle point of this function. A saddle point is a point (x, y) for which f attains a local minimum along one direction and a local maximum in an orthogonal direction.

The function is $f(x, y) = 4x^2 - 4y^2$, and the gradient is followed:

$$\frac{\partial f(x,y)}{\partial x} = 8x \qquad \frac{\partial f(x,y)}{\partial y} = -8y$$

$$\frac{\partial^2 f(x,y)}{\partial x^2} = 8 \qquad \frac{\partial^2 f(x,y)}{\partial y^2} = -8$$

$$\frac{\partial^2 f(x,y)}{\partial x \partial y} = 0$$

If we let the first derivation, $f_x(x, y)$ and $f_y(x, y)$ equal to 0, we have $(0, 0)$ as a critical point.

$$\frac{\partial^2 f(x,y)}{\partial x^2} \cdot \frac{\partial^2 f(x,y)}{\partial y^2} - (\frac{\partial^2 f(x,y)}{\partial x \partial y})^2 = -64 < 0$$

Therefore, $(0, 0)$ is the saddle point.

b) Write down the gradient descent/ascent equations for solving this problem starting at some arbitrary initialization (x0, y0).

The gradient in x and y direction of point (x_i, y_i) assumes as $f_x(x_i, y_i)$, $f_y(x_i, y_i)$. Then in the first iteration, x_1 and y_1 will be :

$$x_1 = x_0 - \eta f_x(x_0, y_0)$$
$$y_1 = y_0 + \eta f_y(x_0, y_0)$$

where η is the learning rate.

For the following iteration, the expression would be:

$$x_{i+1} = x_i - \eta f_x(x_i, y_i)$$
$$y_{i+1} = y_i + \eta f_y(x_i, y_i)$$

c) Determine the range of allowable step sizes to ensure that gradient descent/ascent converges.

From problem **(a)**, we know the saddle point is $(0, 0)$. So, as long as $|x_{i+1}| < |x_i|$, $|y_{i+1}| < |y_i|$, we could say gradient de- scent/ascent converges.

$$\begin{aligned} |x_{i+1}| &< |x_i| \\ x_{i+1}^2 &< x_i^2 \\ (x_i - \eta f_x(x, y))^2 &< x_i^2 \\ (x_i - 8\eta x_i)^2 &< x_i^2 \\ x_i^2 - 16\eta x_i^2 + 64\eta^2 x_i^2 &< x_i^2 \\ 64\eta^2 &< 16\eta \\ \eta &< \frac{1}{4} \end{aligned}$$

Similarly, for y,

$$\begin{aligned} |y_{i+1}| &< |y_i| \\ (y_i + \eta f_y(x_i, y_i))^2 &< y_i^2 \\ y_i^2 - 16\eta y_i^2 + 64\eta^2 y_i^2 &< y_i^2 \\ \eta &< \frac{1}{4} \end{aligned}$$

Since η inherently larger than zero, therefore, the range of step size $\eta \in (0, \frac{1}{4})$.

d) What if you just did regular gradient descent over both variables instead? Comment on the dynamics of the updates and whether there are special cases where one might converge to the saddle point anyway.

If I did regular gradient descent over both variables, there would be only one special case in which the function would converge. If and only if the initialization point of y is 0, the function could converge by gradient descent. Assume the initialization pint is $(x_0, 0)$, where x_0 is a random number under the domain of definition.

$$f_x(x_0, 0) = 8x_0 \qquad f_y(x_t, 0) = 0$$

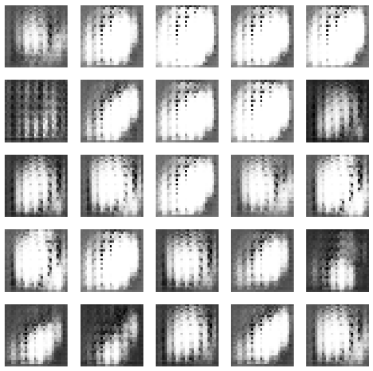
In this case, updates would be:

$$x_t = x_{t-1} - \eta f_x(x_{t-1}, 0) \qquad y_t = y_0 = 0$$

Therefore, only if the y axis value of the initial point is zero could the function converge to the saddle point.

Problem 3

The code for problem 3 is followed on the next page. The image generated after 1, 10, 30, and 50 epochs is shown below:



(a) Epoch =1

(b) Epoch =10

(c) Epoch =30

(d) Epoch =50

q3-google

April 28, 2024

0.1 Probelm 3

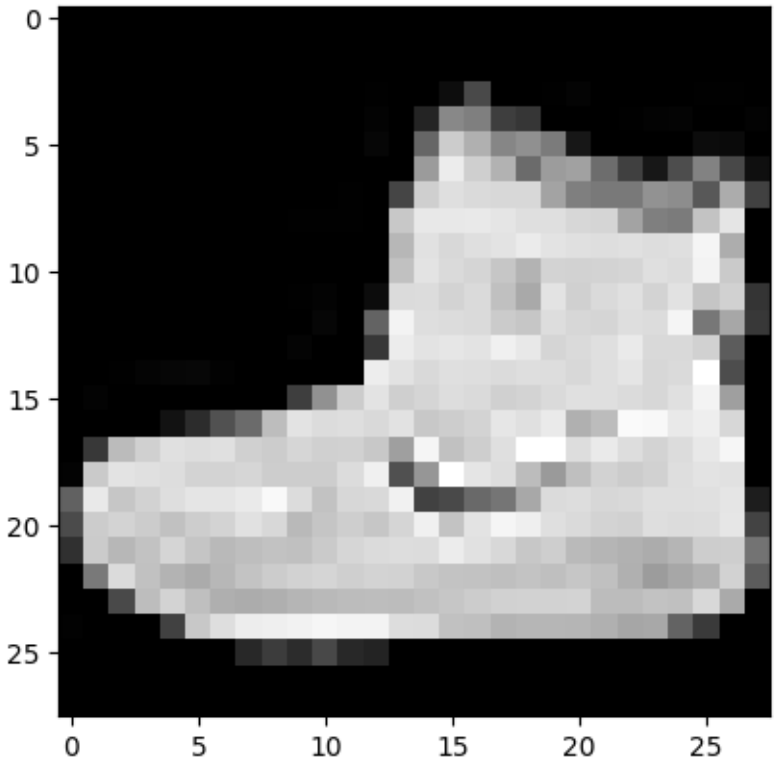
0.1.1 3a. Use the FashionMNIST training dataset (which we used in previous assignments) to train the DCGAN. Images are grayscale and size 28 × 28.

[1]:
```
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
import random
```

[3]:
```
## download the data
train_data = torchvision.datasets.FashionMNIST('./FashionMNIST/
 ↪',train=True,download=True,transform=torchvision.transforms.ToTensor())
test_data = torchvision.datasets.FashionMNIST('./FashionMNIST/
 ↪',train=False,download=True,transform=torchvision.transforms.ToTensor())

image, label = train_data[0]
print(image.size())

plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

```
torch.Size([1, 28, 28])
```

[3]:
```
<matplotlib.image.AxesImage at 0x7e3de60bf970>
```



0.2 3b.

Use the following discriminator architecture (kernel size = 5 × 5 with stride = 2 in both directions):

- 2D convolutions(1×28×28→64×14×14→128×7×7)
- each convolutional layer is equipped with a Leaky ReLU with slope 0.3, followed by Dropout with parameter 0.3.
- a dense layer that takes the flattened output of the last convolution and maps it to a scalar.

[4]:
```
import torch
import torch.nn as nn

class Discriminator(nn.Module):
    def __init__(self, input_channels=1):
        super(Discriminator, self).__init__()

        self.conv1 = nn.Conv2d(input_channels, 64, kernel_size=5, stride=2,␣
 ↪padding=2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=5, stride=2, padding=2)

        self.leaky_relu = nn.LeakyReLU(0.3)
        self.dropout = nn.Dropout(0.3)
```

```
        self.dense = nn.Linear(128*7*7, 1)  # 128 channels, 7x7 feature map

    def forward(self, x):
        x = self.conv1(x)
        x = self.leaky_relu(x)
        x = self.dropout(x)

        x = self.conv2(x)
        x = self.leaky_relu(x)
        x = self.dropout(x)

        x = torch.flatten(x, 1)

        x = self.dense(x)

        return x
```

0.3 2.c

Use the following generator architecture (which is essentially the reverse of a standard discriminative architecture). You can use the same kernel size. Construct:

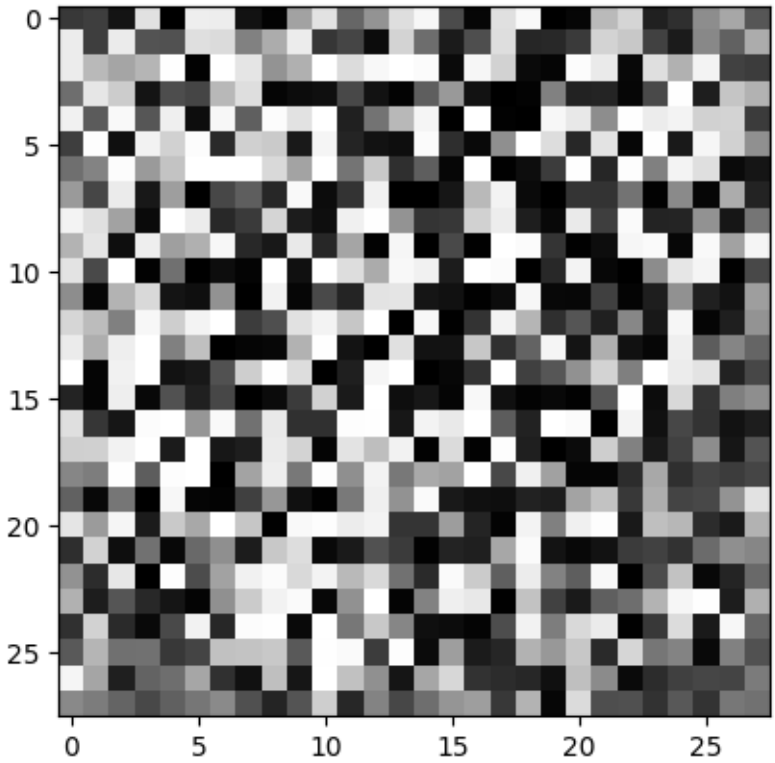
- a dense layer that takes a unit Gaussian noise vector of length 100 and maps it to a vector of size $(7 \times 7 \times 256)$. No bias terms.
- several transpose 2D convolutions $(256 \times 7 \times 7 \rightarrow 128 \times 7 \times 7 \rightarrow 64 \times 14 \times 14 \rightarrow 1 \times 28 \times 28)$. No bias terms.
- each convolutional layer (except the last one) is equipped with Batch Normalization (batch norm), followed by Leaky ReLU with slope 0.3. The last (output) layer is equipped with tanh activation (no batch norm).

[5]:
```
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, input_dim=100, output_channels=1):
        super(Generator, self).__init__()

        self.dense = nn.Linear(input_dim, 7*7*256, bias=False)

        self.conv1 = nn.ConvTranspose2d(256, 128, kernel_size=5, stride=1,␣
 ↪padding=2, bias=False)
        self.conv2 = nn.ConvTranspose2d(128, 64, kernel_size=5, stride=2,␣
 ↪padding=2, output_padding=1, bias=False)
        self.conv3 = nn.ConvTranspose2d(64, output_channels, kernel_size=5,␣
 ↪stride=2, padding=2, output_padding=1, bias=False)

        self.batch_norm1 = nn.BatchNorm2d(128)
```

```
        self.batch_norm2 = nn.BatchNorm2d(64)

        self.leaky_relu = nn.LeakyReLU(0.3)

        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.dense(x)
        x = x.view(-1, 256, 7, 7)

        x = self.conv1(x)
        x = self.batch_norm1(x)
        x = self.leaky_relu(x)

        x = self.conv2(x)
        x = self.batch_norm2(x)
        x = self.leaky_relu(x)

        x = self.conv3(x)
        x = self.tanh(x)

        return x
```

[6]:
```
noise_dim = 100

noise = torch.randn(1, noise_dim)
generator = Generator(input_dim=noise_dim, output_channels=1)
generated_image = generator(noise)

random_img = generated_image.detach().numpy()
plt.imshow(random_img.squeeze(), cmap=plt.cm.gray)
```

[6]:
```
<matplotlib.image.AxesImage at 0x7e3de5ffa350>
```



0.4 3d.

Use the binary cross-entropy loss for training both the generator and the discriminator. Use the Adam optimizer with learning rate 10-4.

[7]:
```
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

device = torch.device('cuda:0' if torch.cuda.is_available() else 'mps')

torch.manual_seed(0)

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5), (0.5,0.5))
])

batch_size = 64
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True,␣
 ↪num_workers=2)
```

```
discriminator = Discriminator().to(device)
generator = Generator(input_dim=noise_dim).to(device)

# Define the loss function (binary cross-entropy)
criterion = nn.BCEWithLogitsLoss().to(device)

# Define the optimizer
lr = 10e-4

d_optimizer = optim.Adam(discriminator.parameters(), lr=lr)
g_optimizer = optim.Adam(generator.parameters(), lr=lr)
```

0.5 3e.

Train it for 50 epochs. You can use minibatch sizes of 16, 32, or 64. Training may take several minutes (or even up to an hour), so be patient! Display intermediate images generated after T = 10, T = 30, and T = 50 epochs.

[13]:
```
noise_dim = 100
num_epochs = 50

d_loss_history=[]
g_loss_history=[]

for epoch in range(num_epochs):
    for i, (real_images, _) in enumerate(train_loader):
        real_images = real_images.to(device)
        batch_size = real_images.size(0)

        # Train the discriminator
        discriminator.zero_grad()

        # Real images
        real_labels = torch.ones(batch_size, 1).to(device)
        real_output = discriminator(real_images)
        d_loss_real = criterion(real_output, real_labels)

        # Fake images
        noise = torch.randn(batch_size, noise_dim).to(device)
        fake_images = generator(noise)
        fake_labels = torch.zeros(batch_size, 1).to(device)
        fake_output = discriminator(fake_images.detach())
        d_loss_fake = criterion(fake_output, fake_labels)

        # Total discriminator loss
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
```

```
        d_optimizer.step()

        # Train the generator
        generator.zero_grad()

        noise = torch.randn(batch_size, noise_dim).to(device)
        fake_images = generator(noise).to(device)
        labels = torch.ones(batch_size, 1).to(device)  # We want the generator␣
↪to fool the discriminator
        output = discriminator(fake_images).to(device)
        g_loss = criterion(output, labels)
        g_loss.backward()
        g_optimizer.step()


    print(f"Epoch [{epoch}/{num_epochs}],"
    f"Discriminator Loss: {d_loss.item():.4f}, Generator Loss: {g_loss.item():.
↪4f}")

    g_loss_history.append(g_loss.item())
    d_loss_history.append(d_loss.item())

    if epoch % 10 == 0 or epoch == num_epochs-1 :
        plt.figure(figsize=(8, 8))
        generator.eval()
        with torch.no_grad():
            noise = torch.randn(25, noise_dim).to(device)
            generated_images = generator(noise).cpu().detach()
            for i in range(25):
                plt.subplot(5, 5, i + 1)
                plt.imshow(generated_images[i].view(28, 28), cmap='gray')
                plt.axis('off')
        plt.savefig(f'Gen_img_ep_{epoch+1}.png')
        plt.close()

print("Training finished.")

plt.figure(figsize=(8, 8))
generator.eval()
with torch.no_grad():
    noise = torch.randn(25, noise_dim).to(device)
    generated_images = generator(noise).cpu().detach()
    for i in range(25):
        plt.subplot(5, 5, i + 1)
        plt.imshow(generated_images[i].view(28, 28), cmap='gray')
        plt.axis('off')
plt.show()
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork()
was called. os.fork() is incompatible with multithreaded code, and JAX is
multithreaded, so this will likely lead to a deadlock.
  self.pid = os.fork()
```

```
Epoch [0/50],Discriminator Loss: 0.2978, Generator Loss: 7.8964
Epoch [1/50],Discriminator Loss: 0.2418, Generator Loss: 8.0672
Epoch [2/50],Discriminator Loss: 0.2277, Generator Loss: 6.9869
Epoch [3/50],Discriminator Loss: 0.3562, Generator Loss: 6.8122
Epoch [4/50],Discriminator Loss: 0.1652, Generator Loss: 6.2451
Epoch [5/50],Discriminator Loss: 0.0717, Generator Loss: 6.2053
Epoch [6/50],Discriminator Loss: 0.2070, Generator Loss: 5.3354
Epoch [7/50],Discriminator Loss: 0.2293, Generator Loss: 4.5872
Epoch [8/50],Discriminator Loss: 0.4453, Generator Loss: 5.4327
Epoch [9/50],Discriminator Loss: 0.1583, Generator Loss: 5.6135
Epoch [10/50],Discriminator Loss: 0.2398, Generator Loss: 4.7858
Epoch [11/50],Discriminator Loss: 0.0679, Generator Loss: 7.8822
Epoch [12/50],Discriminator Loss: 0.3442, Generator Loss: 4.7166
Epoch [13/50],Discriminator Loss: 0.2779, Generator Loss: 3.8867
Epoch [14/50],Discriminator Loss: 0.0992, Generator Loss: 7.1785
Epoch [15/50],Discriminator Loss: 0.1821, Generator Loss: 7.3255
Epoch [16/50],Discriminator Loss: 0.1737, Generator Loss: 6.7124
Epoch [17/50],Discriminator Loss: 0.3358, Generator Loss: 8.7758
Epoch [18/50],Discriminator Loss: 0.1975, Generator Loss: 7.0364
Epoch [19/50],Discriminator Loss: 0.1460, Generator Loss: 7.0675
Epoch [20/50],Discriminator Loss: 0.3119, Generator Loss: 7.1276
Epoch [21/50],Discriminator Loss: 0.1490, Generator Loss: 4.8565
Epoch [22/50],Discriminator Loss: 0.0597, Generator Loss: 7.1303
Epoch [23/50],Discriminator Loss: 0.3916, Generator Loss: 6.0852
Epoch [24/50],Discriminator Loss: 0.4327, Generator Loss: 6.1055
Epoch [25/50],Discriminator Loss: 0.3093, Generator Loss: 5.8036
Epoch [26/50],Discriminator Loss: 0.2480, Generator Loss: 6.0888
Epoch [27/50],Discriminator Loss: 0.1277, Generator Loss: 5.6262
Epoch [28/50],Discriminator Loss: 0.2203, Generator Loss: 3.1419
Epoch [29/50],Discriminator Loss: 0.2904, Generator Loss: 5.2217
Epoch [30/50],Discriminator Loss: 0.2198, Generator Loss: 4.9111
Epoch [31/50],Discriminator Loss: 0.3253, Generator Loss: 5.2738
Epoch [32/50],Discriminator Loss: 0.4809, Generator Loss: 6.4696
Epoch [33/50],Discriminator Loss: 0.3488, Generator Loss: 5.7013
Epoch [34/50],Discriminator Loss: 0.2277, Generator Loss: 4.4278
Epoch [35/50],Discriminator Loss: 0.2271, Generator Loss: 4.9127
Epoch [36/50],Discriminator Loss: 0.0945, Generator Loss: 6.8787
Epoch [37/50],Discriminator Loss: 0.2340, Generator Loss: 6.5240
Epoch [38/50],Discriminator Loss: 0.5721, Generator Loss: 4.5657
Epoch [39/50],Discriminator Loss: 0.4156, Generator Loss: 2.9890
Epoch [40/50],Discriminator Loss: 0.0594, Generator Loss: 6.5327
Epoch [41/50],Discriminator Loss: 0.3381, Generator Loss: 3.6181
Epoch [42/50],Discriminator Loss: 0.2119, Generator Loss: 4.0544
```

```
Epoch [43/50],Discriminator Loss: 0.3433, Generator Loss: 3.5498
Epoch [44/50],Discriminator Loss: 0.7513, Generator Loss: 6.0383
Epoch [45/50],Discriminator Loss: 0.6723, Generator Loss: 5.2906
Epoch [46/50],Discriminator Loss: 0.2062, Generator Loss: 3.2382
Epoch [47/50],Discriminator Loss: 0.4452, Generator Loss: 3.1585
Epoch [48/50],Discriminator Loss: 1.0545, Generator Loss: 4.0914
Epoch [49/50],Discriminator Loss: 0.1397, Generator Loss: 5.4225
Training finished.
```

[17]:
```
plt.plot(d_loss_history,'-',linewidth=3,label='Generator Loss')
plt.plot(g_loss_history,'-',linewidth=3,label='Discriminator Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
```

```
plt.grid(True)
plt.legend()
```

[17]: <matplotlib.legend.Legend at 0x7e3dd02dc2b0>