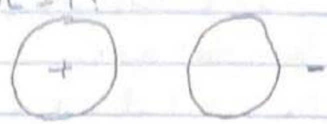
Ethan Wong
305314001
2/17/22

CS M146 Problem Set 3

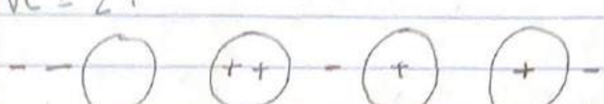
1. a) The following diagrams represent different VCs:

$VC \geq 1$?



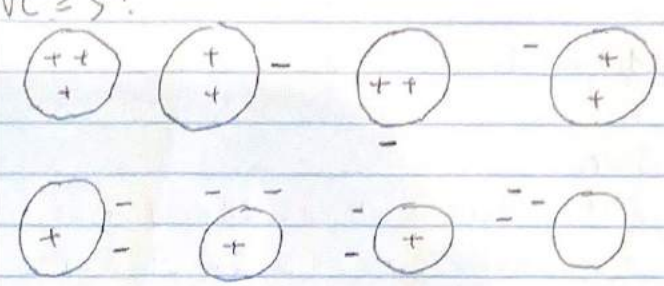
Yes - any single point can be shattered.

$VC \geq 2$?



Yes - there exist 2 points that can be shattered.

$VC \geq 3$?



Yes - there exist 3 points that can be shattered.

$VC \geq 4$

There are several examples where a circle can not shatter 4 points.

(A) Imagine 3 points form a convex hull: if the points that form the hull are + and one inner point is −, then the circle will contain the − as well.

(B) Imagine the points form a convex hull: the further points are + and the closer points are −, then similarly as (A) the circle will contain at least one −.

(C) Imagine 3 points are on the same line; the inner points are − and the outer / edge points are +, then a circle would not be able to shatter these points

Thus, we see that 4 points cannot be shattered and $VC(H_c) = 3$.

1. b) i) $H_1 \subseteq H_2 \quad VC(H_1) \leq VC(H_2)$?

Suppose that $VC(H_1) = x$ since there is a $h \in H_1$ that can shatter a subset with size x. We also know that $h \in H_2$ since $H_1 \subseteq H_2$. Thus, H_2 can shatter the subset of size x as well. This means that $VC(H_2)$ is greater than or equal to x:

$$\underbrace{VC(H_2) \geq x \qquad x = VC(H_1)}$$

$$VC(H_2) \geq VC(H_1)$$

The statement is true. If $H_1 \subseteq H_2$, then we can say that $VC(H_1) \leq VC(H_2)$

b) ii) $H_1 = H_2 \cup H_3 \quad VC(H_1) \leq VC(H_2) + VC(H_3)$

Say $H_3 = \{h_2\}$; $h_2(x) = 1$

Say $H_2 = \{h_1\}$; $h_1(x) = 0$

H_2 and H_3 will predict all data points as all positive or all negative. This means that $VC(H_2) = 0$ and $VC(H_3) = 0$; they can not shatter any point since a single point will either be 0 or 1. However, $H_1 = H_2 \cup H_3$ so $H_1 = \{0, 1\}$. The union of H_2 and H_3 can actualy classify a point, contradicting what we said earlier. This means that:

$VC(H_1) = 1 \quad VC(H_2) = 0 \quad VC(H_3) = 0$

$1 \nleq 0 + 0$

The statement is false. $VC(H_1) \nleq VC(H_2) + VC(H_3)$

2. a) $K(x,z)$ is a Kernel.

We can show this by creating two feature vectors: $\phi(x)$ and $\phi(z)$ such that $K(x,z) = \phi(x) \cdot \phi(z)$. For any set of documents we can establish a dictionary D that is a finite set of the words in the document set.

With this dictionary, the feature mapping $\phi(x)$ could be represented as so:

w_i is the i^{th} word in dictionary D
if (w_i is in document x):
  $\phi(x)_i = 1$, where $\phi(x)_i$ is the i^{th} element of $\phi(x)$
else:
  $\phi(x)_i = 0$

Alternatively:

$$\phi(x)_i = \begin{cases} 1 & \text{if } w_i \text{ is in document } x \\ 0 & \text{else} \end{cases}$$

Knowing all this, the intersection of the sets of the words in the two documents could be given by $\phi(x) \cdot \phi(z)$. This would effectively give us the kernel.

2. b) $$\left(1+\left(\frac{x}{\|x\|}\right)\cdot\left(\frac{z}{\|z\|}\right)\right)^3$$

This kernel could be constructed by following these steps:

① Scale

$$K_1(x,z) = \frac{1}{\|x\|}\frac{1}{\|z\|}K(x,z) = \left(\frac{x}{\|x\|}\cdot\frac{z}{\|z\|}\right)$$

② Sum

$$K_2(x,z) = 1 + K_1(x,z) = 1+\left(\frac{x}{\|x\|}\cdot\frac{z}{\|z\|}\right)$$

③ Product

$$K_3(x,z) = K_2(x,z)\cdot K_2(x,z) = \left[1+\left(\frac{x}{\|x\|}\cdot\frac{z}{\|z\|}\right)\right]^2$$

$$K_4(x,z) = K_3(x,z)\cdot K_2(x,z) = \left[1+\left(\frac{x}{\|x\|}\cdot\frac{z}{\|z\|}\right)\right]^3$$

By following these steps, we see that the given expression is a kernel as we can use a series of scalings, sums, and products to achieve it.

2. c) $K_\beta(x,z) = (1 + \beta x \cdot z)^3$

$K_\beta(x,z) = (1 + \beta(x_1 z_1 + x_2 z_2))^3$

Expand:

$K_\beta(x,z) = 1 + 3\beta(x_1 z_1 + x_2 z_2) + 3\beta^2(x_1^2 z_1^2 + 2 x_1 z_1 x_2 z_2 + x_2^2 z_2^2)$
$\quad + \beta^3(x_1^3 z_1^3 + 3x_1^2 z_1^2 x_2 z_2 + 3x_1 z_1 x_2^2 z_2^2 + x_2^3 z_2^3)$

$\phi_\beta(x) = (1, \sqrt{3\beta}\, x_1, \sqrt{3\beta}\, x_2, \sqrt{3}\beta x_1^2, \sqrt{6}\beta x_1 x_2, \sqrt{3}\beta x_2^2, \sqrt{\beta^3}\, x_1^3, \sqrt{3\beta^3}\, x_1^2 x_2,$
$\quad \sqrt{3\beta^3}\, x_1 x_2^2, \sqrt{\beta^3}\, x_2^3)^T$

$\phi(x) = (1, \sqrt{3}x, \sqrt{3}x_2, \sqrt{3}x_1^2, \sqrt{6}x_1 x_2, \sqrt{3}x_2^2, \sqrt{3}x_1^2 x_2, \sqrt{3}x_1 x_2^2, x_1^3, x_2^3)^T$

The parameter β serves as a regularizer term because it makes the higher order terms more costly to use than the lower order terms. This is because we have $\beta^{n/2}$ (n is degree), and this becomes larger as n decreases. This shows that the kernel is biased towards lower-degree polynomials. We observe that as β approaches $+\infty$, only the higher-degree terms and constants are left as they outweight the lower-degree terms. As β approaches 0, we instead have a linear separator.

Similarities between $K(x,z)$ and $K_\beta(x,z)$

- Both have offset terms of 1
- Both are polynomial kernels with degree 3 (exponent = 3)

Differences between $K(x,z)$ and $K_\beta(x,z)$

- In $K_\beta(x,z)$, the cubic terms get scaled by $\beta^{3/2}$
- In $K_\beta(x,z)$, the linear terms get scaled by $\sqrt{\beta}$
- In $K_\beta(x,z)$, the quadratic terms get scaled by β

3. a) $x = (a, e)^T \quad y = -1 \quad$ Constraint: $y_n \theta^T x_n = 1$

This SVM has a single negative data point. This would orient θ so that it points in the opposite direction of that data point. This is done in order to minimize the objective function while satisfying the constraint mentioned above.

As a result, the corresponding θ is given by:

$$\boxed{\theta^* = \frac{-x}{\|x\|^2}}$$

b) $x_1 = (1, 1)^T \quad x_2 = (1, 0)^T \quad y_1 = 1 \quad y_2 = -1$

This SVM will use both x_1 and x_2 as support vectors.

- $y_1 \theta^T x_1 = 1$ — $(1)\, \theta^T \binom{1}{1} = 1$
- $y_2 \theta^T x_2 = 1$ — $(-1)\, \theta^T \binom{1}{0} = 1$

Use a system of equations:

$$\begin{cases} \theta_1 + \theta_2 = 1 \\ -\theta_1 + 0 = 1 \end{cases}$$

$$\theta_1 = -1, \quad \theta_2 = 2$$

$$\text{margin}(\theta^*) = \frac{1}{\|\theta^*\|} = \frac{1}{\sqrt{1+4}} = \frac{1}{\sqrt{5}}$$

$$\boxed{\text{Thus: } \theta^* = [-1, 2]^T \quad \gamma = \frac{1}{\sqrt{5}}}$$

c) $x_1 = (1, 1)^T \quad x_2 = (1, 0)^T \quad y_1 = 1 \quad y_2 = -1$

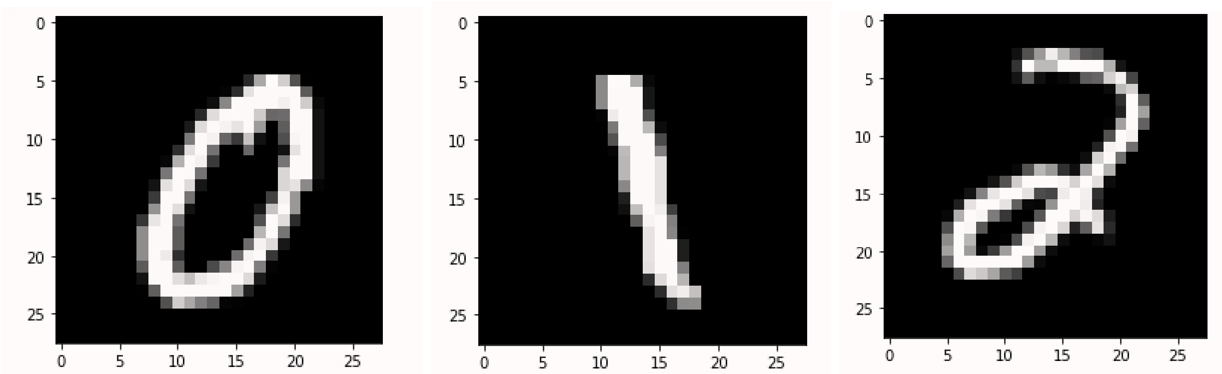
- If we were to plot x_1 and x_2, we'd see that the line $x = \frac{1}{2}$ would maximize the margin between the points.
- Normal vectors to the line
  - $\hookrightarrow \theta_1^* = 0 \quad \theta_2^* > 0$
- $y_1[\theta^T x_1 + b] = 1$
  - $(1)[(0 \ \theta_2)\binom{1}{1} + b] = 1$
  - $\theta_2 + b = 1$
- $y_2[\theta^T x_2 + b] = 1$
  - $(-1)[(0 \ \theta_2)\binom{1}{0} + b] = 1$
  - $b = -1$

$$\underbrace{\theta_2 + b = 1 \qquad b = -1}_{\theta_2 = 2}$$

$$\text{margin}(\theta^*) = \frac{1}{\|\theta^*\|} = \frac{1}{\sqrt{0+4}} = \frac{1}{2}$$

Thus: $\boxed{\theta^* = [0, 2]^T \quad b^* = -1 \quad \gamma = \frac{1}{2}}$

4A)



```python
### ========== TODO : START ========== ###
### part a: print out three training images with different labels
setX0 = False
setX1 = False
setX2 = False
while True:
    randRow = np.random.randint(y_train.shape[0])
    if ((y_train[randRow] == 2) and (not setX2)):
      setX2 = True
      X2 = X_train[randRow]
    if ((y_train[randRow] == 1) and (not setX1)):
      setX1 = True
      X1 = X_train[randRow]
    if ((y_train[randRow] == 0) and (not setX0)):
      setX0 = True
      X0 = X_train[randRow]
    if (setX2 and setX1 and setX0):
      break

plot_img(X0)
plot_img(X1)
plot_img(X2)
### ========== TODO : END ========== ###
```

4B)

"You do not have to submit anything for this part."

4C)

```
### ========== TODO : START ========== ###
### part c: prepare dataloaders for training, validation, and testing
###         we expect to get a batch of pairs (x_n, y_n) from the dataloader
train_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(X_train, y_train), batch_size = 10)
valid_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(X_valid, y_valid), batch_size = 10)
test_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(X_test, y_test), batch_size = 10)
### ========== TODO : END ========== ###
```

4D)

```
class OneLayerNetwork(torch.nn.Module):
    def __init__(self):
        super(OneLayerNetwork, self).__init__()

        ### ========== TODO : START ========== ###
        ### part d: implement OneLayerNetwork with torch.nn.Linear
        self.WOne = torch.nn.Linear(784, 3)
        ### ========== TODO : END ========== ###

    def forward(self, x):
        # x.shape = (n_batch, n_features)

        ### ========== TODO : START ========== ###
        ### part d: implement the foward function
        outputs = self.WOne(x)
        ### ========== TODO : END ========== ###
        return outputs
```

4E)

```
### ========== TODO : START ========== ###
### part e: prepare OneLayerNetwork, criterion, and optimizer
model_one = OneLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_one.parameters(), lr = 0.0005)
### ========== TODO : END ========== ###
```

4F)

```
### ========== TODO : START ========== ###
### part f: implement the training process
y = model.forward(batch_X)
optimizer.zero_grad()
L = criterion(y, batch_y)
L.backward()
optimizer.step()
### ========== TODO : END ========== ###
```

```
Start training OneLayerNetwork...
| epoch  1 | train loss 1.075398 | train acc 0.453333 | valid loss 1.084938 | valid acc 0.453333 |
| epoch  2 | train loss 1.021364 | train acc 0.566667 | valid loss 1.031102 | valid acc 0.553333 |
| epoch  3 | train loss 0.972648 | train acc 0.630000 | valid loss 0.982742 | valid acc 0.593333 |
| epoch  4 | train loss 0.928398 | train acc 0.710000 | valid loss 0.938953 | valid acc 0.640000 |
| epoch  5 | train loss 0.887963 | train acc 0.783333 | valid loss 0.899045 | valid acc 0.700000 |
| epoch  6 | train loss 0.850839 | train acc 0.826667 | valid loss 0.862485 | valid acc 0.753333 |
| epoch  7 | train loss 0.816627 | train acc 0.850000 | valid loss 0.828852 | valid acc 0.793333 |
| epoch  8 | train loss 0.785000 | train acc 0.886667 | valid loss 0.797807 | valid acc 0.846667 |
| epoch  9 | train loss 0.755688 | train acc 0.900000 | valid loss 0.769067 | valid acc 0.866667 |
| epoch 10 | train loss 0.728461 | train acc 0.903333 | valid loss 0.742397 | valid acc 0.873333 |
| epoch 11 | train loss 0.703122 | train acc 0.913333 | valid loss 0.717596 | valid acc 0.880000 |
| epoch 12 | train loss 0.679499 | train acc 0.920000 | valid loss 0.694488 | valid acc 0.886667 |
| epoch 13 | train loss 0.657439 | train acc 0.933333 | valid loss 0.672921 | valid acc 0.886667 |
| epoch 14 | train loss 0.636807 | train acc 0.943333 | valid loss 0.652760 | valid acc 0.886667 |
| epoch 15 | train loss 0.617482 | train acc 0.943333 | valid loss 0.633883 | valid acc 0.886667 |
| epoch 16 | train loss 0.599356 | train acc 0.943333 | valid loss 0.616184 | valid acc 0.886667 |
| epoch 17 | train loss 0.582330 | train acc 0.943333 | valid loss 0.599565 | valid acc 0.893333 |
| epoch 18 | train loss 0.566316 | train acc 0.943333 | valid loss 0.583938 | valid acc 0.900000 |
| epoch 19 | train loss 0.551234 | train acc 0.943333 | valid loss 0.569225 | valid acc 0.906667 |
| epoch 20 | train loss 0.537010 | train acc 0.943333 | valid loss 0.555355 | valid acc 0.906667 |
| epoch 21 | train loss 0.523580 | train acc 0.943333 | valid loss 0.542262 | valid acc 0.906667 |
| epoch 22 | train loss 0.510882 | train acc 0.943333 | valid loss 0.529888 | valid acc 0.906667 |
| epoch 23 | train loss 0.498862 | train acc 0.950000 | valid loss 0.518179 | valid acc 0.906667 |
| epoch 24 | train loss 0.487470 | train acc 0.950000 | valid loss 0.507086 | valid acc 0.906667 |
| epoch 25 | train loss 0.476660 | train acc 0.950000 | valid loss 0.496564 | valid acc 0.906667 |
| epoch 26 | train loss 0.466391 | train acc 0.953333 | valid loss 0.486573 | valid acc 0.926667 |
| epoch 27 | train loss 0.456625 | train acc 0.953333 | valid loss 0.477076 | valid acc 0.926667 |
| epoch 28 | train loss 0.447328 | train acc 0.953333 | valid loss 0.468038 | valid acc 0.926667 |
| epoch 29 | train loss 0.438467 | train acc 0.956667 | valid loss 0.459429 | valid acc 0.933333 |
| epoch 30 | train loss 0.430013 | train acc 0.956667 | valid loss 0.451220 | valid acc 0.940000 |
Done!
```

4G)

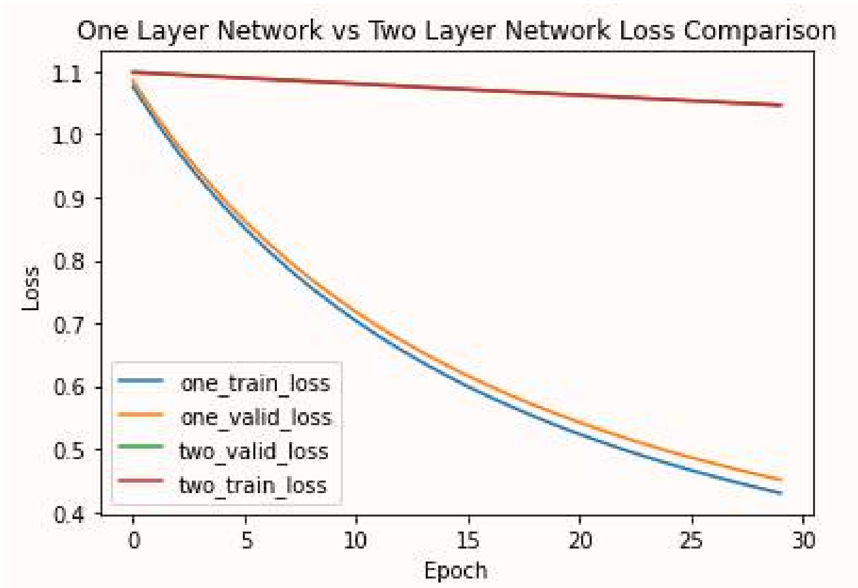
```python
class TwoLayerNetwork(torch.nn.Module):
    def __init__(self):
        super(TwoLayerNetwork, self).__init__()
        ### ========== TODO : START ========== ###
        ### part g: implement TwoLayerNetwork with torch.nn.Linear
        self.one = torch.nn.Linear(784, 400)
        self.two = torch.nn.Linear(400, 3)
        ### ========== TODO : END ========== ###

    def forward(self, x):
        # x.shape = (n_batch, n_features)

        ### ========== TODO : START ========== ###
        ### part g: implement the foward function
        LayerOne = self.one(x)
        s = torch.nn.Sigmoid()

        LayerOne = s(LayerOne)
        outputs = self.two(LayerOne)
        ### ========== TODO : END ========== ###
        return outputs
```

4H)

```
### ========== TODO : START ========== ###
### part h: prepare TwoLayerNetwork, criterion, and optimizer
model_two = TwoLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_two.parameters(), lr = 0.0005)
### ========== TODO : END ========== ###
```

```
Start training TwoLayerNetwork...
| epoch  1 | train loss 1.098020 | train acc 0.240000 | valid loss 1.098498 | valid acc 0.253333 |
| epoch  2 | train loss 1.096157 | train acc 0.283333 | valid loss 1.096622 | valid acc 0.340000 |
| epoch  3 | train loss 1.094329 | train acc 0.386667 | valid loss 1.094783 | valid acc 0.380000 |
| epoch  4 | train loss 1.092512 | train acc 0.433333 | valid loss 1.092956 | valid acc 0.400000 |
| epoch  5 | train loss 1.090700 | train acc 0.470000 | valid loss 1.091135 | valid acc 0.413333 |
| epoch  6 | train loss 1.088891 | train acc 0.486667 | valid loss 1.089318 | valid acc 0.420000 |
| epoch  7 | train loss 1.087085 | train acc 0.496667 | valid loss 1.087503 | valid acc 0.453333 |
| epoch  8 | train loss 1.085281 | train acc 0.526667 | valid loss 1.085691 | valid acc 0.466667 |
| epoch  9 | train loss 1.083480 | train acc 0.533333 | valid loss 1.083882 | valid acc 0.486667 |
| epoch 10 | train loss 1.081682 | train acc 0.550000 | valid loss 1.082076 | valid acc 0.506667 |
| epoch 11 | train loss 1.079886 | train acc 0.560000 | valid loss 1.080273 | valid acc 0.540000 |
| epoch 12 | train loss 1.078093 | train acc 0.573333 | valid loss 1.078472 | valid acc 0.553333 |
| epoch 13 | train loss 1.076302 | train acc 0.593333 | valid loss 1.076674 | valid acc 0.566667 |
| epoch 14 | train loss 1.074514 | train acc 0.633333 | valid loss 1.074878 | valid acc 0.626667 |
| epoch 15 | train loss 1.072727 | train acc 0.683333 | valid loss 1.073084 | valid acc 0.660000 |
| epoch 16 | train loss 1.070942 | train acc 0.750000 | valid loss 1.071292 | valid acc 0.693333 |
| epoch 17 | train loss 1.069159 | train acc 0.776667 | valid loss 1.069502 | valid acc 0.746667 |
| epoch 18 | train loss 1.067377 | train acc 0.806667 | valid loss 1.067713 | valid acc 0.773333 |
| epoch 19 | train loss 1.065597 | train acc 0.820000 | valid loss 1.065926 | valid acc 0.800000 |
| epoch 20 | train loss 1.063817 | train acc 0.826667 | valid loss 1.064139 | valid acc 0.820000 |
| epoch 21 | train loss 1.062038 | train acc 0.843333 | valid loss 1.062354 | valid acc 0.833333 |
| epoch 22 | train loss 1.060260 | train acc 0.860000 | valid loss 1.060569 | valid acc 0.840000 |
| epoch 23 | train loss 1.058483 | train acc 0.870000 | valid loss 1.058785 | valid acc 0.853333 |
| epoch 24 | train loss 1.056706 | train acc 0.876667 | valid loss 1.057001 | valid acc 0.860000 |
| epoch 25 | train loss 1.054928 | train acc 0.883333 | valid loss 1.055217 | valid acc 0.880000 |
| epoch 26 | train loss 1.053151 | train acc 0.886667 | valid loss 1.053433 | valid acc 0.886667 |
| epoch 27 | train loss 1.051374 | train acc 0.890000 | valid loss 1.051650 | valid acc 0.893333 |
| epoch 28 | train loss 1.049596 | train acc 0.893333 | valid loss 1.049865 | valid acc 0.900000 |
| epoch 29 | train loss 1.047818 | train acc 0.893333 | valid loss 1.048081 | valid acc 0.900000 |
| epoch 30 | train loss 1.046038 | train acc 0.896667 | valid loss 1.046295 | valid acc 0.893333 |
Done!
```

4l)

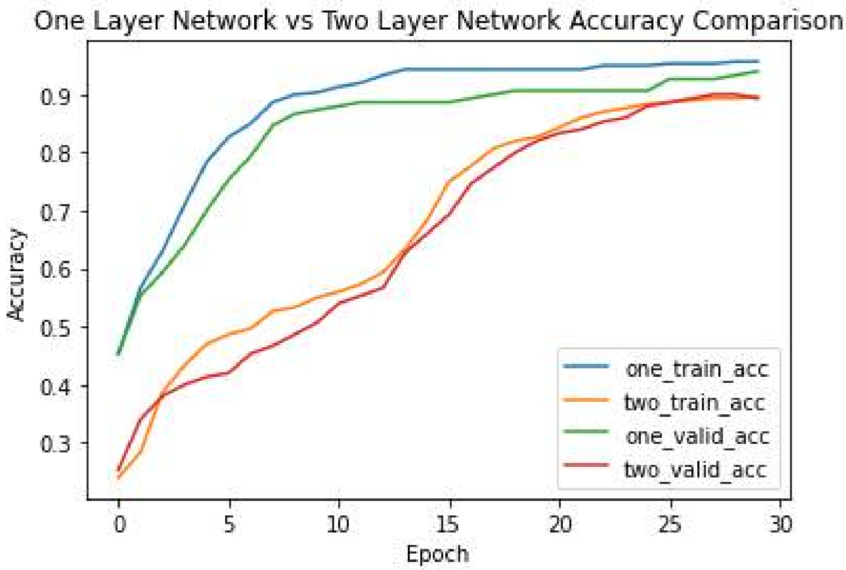


The one layer network and the two layer network behave differently as epoch increases. The two layer network experiences a linear decrease as epoch increases. Even after 30 Epochs, the loss is still relatively high as it never dropped below a loss of 1.0. The training loss and valid loss were nearly identical across all epochs, which is why it is difficult to identify the individual lines in the plot above.

For the one layer network, the decrease in error follows a more exponential decrease as the epoch increases. The loss achieved by the one layer network was mch lower than the two layer network, with a final loss of about 0.45. The training loss was slightly lower than the valid loss across all epochs.

```
### ========== TODO : START ========== ###
### part i: generate a plot to comare one_train_loss, one_valid_loss, two_train_loss, two_valid_loss
plt.figure()
plt.plot(one_train_loss, label = 'one_train_loss')
plt.plot(one_valid_loss, label = 'one_valid_loss')

plt.plot(two_valid_loss, label = 'two_valid_loss')
plt.plot(two_train_loss, label = 'two_train_loss')

plt.title('One Layer Network vs Two Layer Network Loss Comparison')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
### ========== TODO : END ========== ###
```

4J)



The one layer network and the two layer network behave differently as epoch increases. The two layer network generates curves that look reminiscent of a combination of two logarithmic curves. This is most likely because there are 2 layers in the model, hence the 2 log curves. The training accuracy is slightly higher than the valid accuracy for the majority of the algorithm, but by the final epoch they are almost identical values. By the final epoch, the accuracy is about 0.9.

The one layer network starts with a much higher accuracy of about 0.45 where the two layer network started much lower. The accuracy for the one layer network is also much higher, as it reaches nearly 100% accuracy about halfway through the algorithm and remains there until the end. Throughout the algorithm the training accuracy is slightly higher than the valid accuracy.

```
### ========== TODO : START ========== ###
### part j: generate a plot to comare one_train_acc, one_valid_acc, two_train_acc, two_valid_acc
plt.figure()
plt.plot(one_train_acc, label = 'one_train_acc')
plt.plot(two_train_acc, label = 'two_train_acc')

plt.plot(one_valid_acc, label = 'one_valid_acc')
plt.plot(two_valid_acc, label = 'two_valid_acc')

plt.title('One Layer Network vs Two Layer Network Accuracy Comparison')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
### ========== TODO : END ========== ##
```

4K)

```
The One Layer Network Test Accuracy is 96.00 %
The Two Layer Network Test Accuracy is 90.00 %
```

The test accuracy of both the one layer network and the two layer network is seen above. The accuracy of both the networks can be improved by changing the learning rate. I will change the learning rate to 0.05 instead of 0.0005, which was used above. After changing the learning rate the improved results are seen as follows:

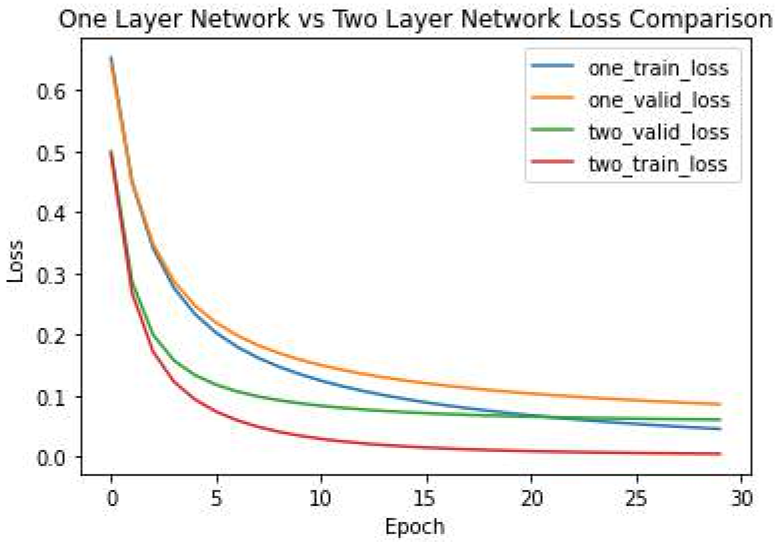
```
The One Layer Network Test Accuracy is 97.33 %
The Two Layer Network Test Accuracy is 97.33 %
```

```
### ========== TODO : START ========== ###
### part k: calculate the test accuracy
testOne = 100 * float(evaluate_acc(model_one, test_loader))
testTwo = 100 * float(evaluate_acc(model_two, test_loader))

print('The One Layer Network Test Accuracy is %2.2f '% (testOne) + '%')
print('The Two Layer Network Test Accuracy is %2.2f '% (testTwo) + '%')
### ========== TODO : END ========== ###
```

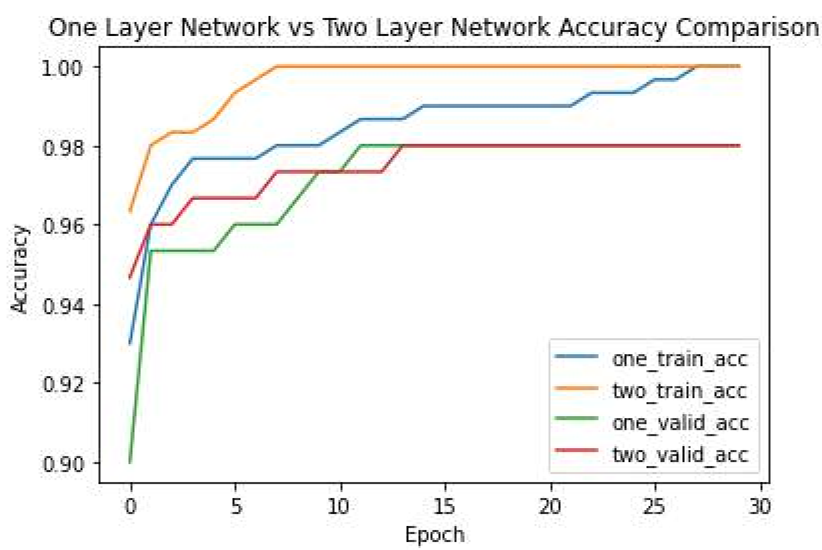
I changed the learning rate to 0.05 in this part of the code, where it was originally 0.0005.

```
### ========== TODO : START ========== ###
### part e: prepare OneLayerNetwork, criterion, and optimizer
model_one = OneLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_one.parameters(), lr = 0.05)
### ========== TODO : END ========== ###

print("Start training OneLayerNetwork...")
results_one = train(model_one, criterion, optimizer, train_loader, valid_loader)
print("Done!")

### ========== TODO : START ========== ###
### part h: prepare TwoLayerNetwork, criterion, and optimizer
model_two = TwoLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_two.parameters(), lr = 0.05)
### ========== TODO : END ========== ###
```

4L)



The plot for the loss comparison looks much different using the Adam optimizer when compared to the SGD optimizer. All four of the curves follow a similar pattern of exponential decrease. The one layer networks seem to have slightly worse performance as the the two layer network nearly reaches a loss of 0 with two_train_loss when epoch is 30. The two layer network also started at a lower loss of about 0.5, whereas the one layer network started with a loss of about 0.7. Overall it seems that the two layer loss has better performance with the Adam optimizer as the loss is consistently lower than the one layer network at almost all epochs.



The plot for the accuracy also changes quite dramatically with the Adam optimizer. All four of the curves form a stair-like pattern where they increase steadily for a couple iterations before increasing again. After about 15 iterations, they all start to level off or experience smaller increases in accuracy. Once again, the two layer network seems to perform better here as two_train_acc achieves 100% accuracy around epoch 7. The one layer network eventually reaches 100% accuracy as well with one_train_acc, but it happens much later at around epoch 28. The two layer network also starts off with a higher accuracy at epoch 0; the one layer network starts off with much lower accuracy. Overall, it seems like the two layer network performs better with the Adam optimizer as the accuracy is consistently better than the one layer network.

Learning rate = 0.0005

```
The One Layer Network Test Accuracy is 97.33 %
The Two Layer Network Test Accuracy is 96.67 %
```

It seemed like the one layer network was much slower in improving its accuracy, but its test accuracy was higher than that of the two layer network. The performance using the learning rate of 0.0005 with the Adam optimizer was a couple percent better than that of the SGD optimizer.

Learning Rate = 0.05

```
The One Layer Network Test Accuracy is 96.67 %
The Two Layer Network Test Accuracy is 99.33 %
```

Changing the learning rate to 0.05 again actually did help the accuracy of the two layer network. However, it was not very helpful for the one layer network. The plots also look extremely erratic and noisy when doing this. Overall, it seems that the learning rate of 0.0005 is better for the Adam optimizer — the small increase in test accuracy for the two layer network outweighs the side effects of tweaking the learning rate.

This experiment has shown us that efficiency of a model can be heavily affected by the choice of optimizer, the number of layers used, and the hyperparameter values used.

```python
### part 1: replace the SGD optimizer with the Adam optimizer and do the experiments again
model_one = OneLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_one.parameters(), lr = 0.05)
print("Start training OneLayerNetwork using Adam...")
results_one = train(model_one, criterion, optimizer, train_loader, valid_loader)

model_two = TwoLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_two.parameters(), lr = 0.05)
print("Start training TwoLayerNetwork using Adam...")
results_two = train(model_two, criterion, optimizer, train_loader, valid_loader)

one_train_loss, one_valid_loss, one_train_acc, one_valid_acc = results_one
two_train_loss, two_valid_loss, two_train_acc, two_valid_acc = results_two

plt.figure()
plt.plot(one_train_loss, label = 'one_train_loss')
plt.plot(one_valid_loss, label = 'one_valid_loss')
plt.plot(two_valid_loss, label = 'two_valid_loss')
plt.plot(two_train_loss, label = 'two_train_loss')
plt.title('One Layer Network vs Two Layer Network Loss Comparison')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

plt.figure()
plt.plot(one_train_acc, label = 'one_train_acc')
plt.plot(two_train_acc, label = 'two_train_acc')
plt.plot(one_valid_acc, label = 'one_valid_acc')
plt.plot(two_valid_acc, label = 'two_valid_acc')
plt.title('One Layer Network vs Two Layer Network Accuracy Comparison')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

testOne = 100 * float(evaluate_acc(model_one, test_loader))
testTwo = 100 * float(evaluate_acc(model_two, test_loader))
print('The One Layer Network Test Accuracy is %2.2f '% (testOne) + '%')
print('The Two Layer Network Test Accuracy is %2.2f '% (testTwo) + '%')
### ========== TODO : END ========== ###
```