CM146, Winter 2021

Problem Set 3: Deep Learning, Learning Theory, Kernels

Due Feb 26, 2021

Name: Joseph Picchi

UID: 605-124-511

Problem 1: VC-Dimension

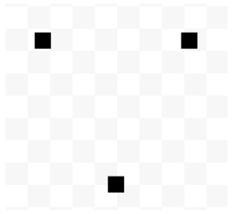
(a) Problem 1a

Solution:

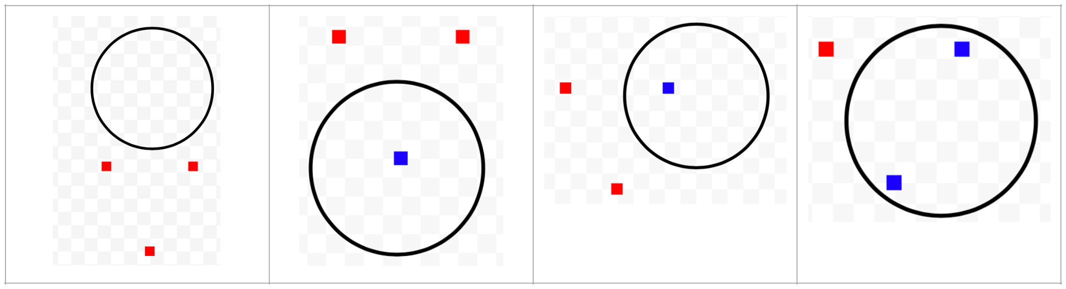
$VC(H_c) = 3$, as justified below.

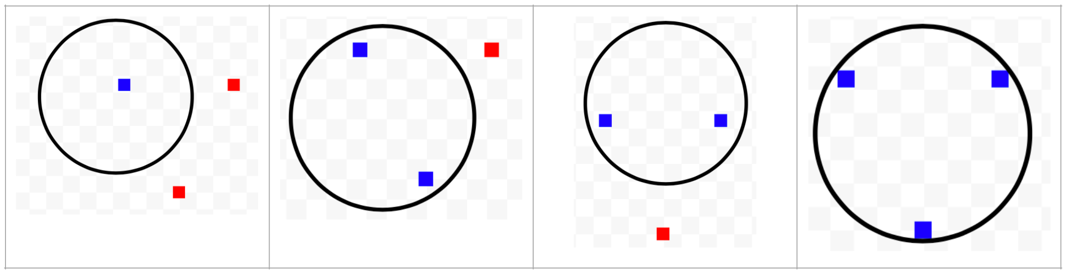
Proof:

1. Consider the following set of 3 points, arranged as a triangle:

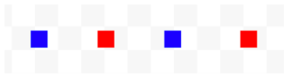


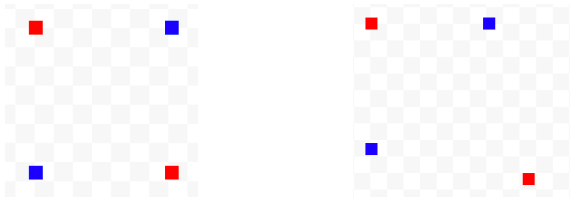
2. Each point can have one of two possible labels, so there are $2^3 = 8$ possible labelings for the set of 3 points.
3. We enumerate over all possible labelings from (2), where blue indicates positive labels and red indicates negative labels. In all cases, a circle classifier correctly classifies the points:

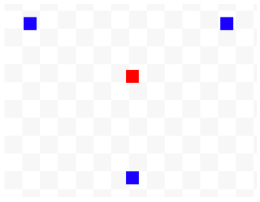




4. By (3), $VC(H_c) \geq 3$ because there exists a subset of X of size three that can be shattered by H_c
5. We enumerate over all geometric configurations of 4 points and show that no configuration can be shattered by H_c:
    1. If the 4 points are collinear
        1. label them as + - + -

           
        2. No circle can correctly classify this
    2. Make a quadrilateral convex hull
        1. Label the two points on the long diagonal as positive and the two on the short diagonal as negative (pick the long diagonal at random between the two diagonals if both diagonals are equivalent length)
            1. Examples:

               
        2. No circle can correctly classify this
    3. Make a triangular convex hull with one point inside
        1. Label the center point as - and all other points as +

           
        2. No circle can correctly classify this
6. By (5), no configuration of 4 points can be shattered.
7. By (5), $VC(H_c) < 4$
8. $\therefore$ By (3) and (7), $VC(H_c) = 3$

(b) Problem 1b

Part (i)

Solution:

The given statement is TRUE.

Proof:

1. CLAIM: $VC(H_1) \leq VC(H_2)$ for all $H_1 \subseteq H_2$
2. By contradiction, assume that there exists some H_1, H_2 such that $H_1 \subseteq H_2$ and $VC(H_1) > VC(H_2)$
3. By (2), there exists a subset S of $\left(VC(H_2) + 1\right)$ points such that H_1 can shatter S but H_2 cannot shatter S.
4. By definition, $H_1 \subseteq H_2$
5. By (4), all hypothesis in H_1 also appear in H_2.
6. By (5), any subset S' that can be shattered by H_1 can also be shattered by H_2 because H_2 contains all hypothesis from H_1 that were used to shatter S'.
7. By (3), H_1 can shatter S.
8. By (7) and (6), H_2 can shatter S
9. (3) and (8) contradict.
10. $\therefore$ By contradiction, the claim is proven.

Part (ii)

Solution:

The given statement is FALSE.

Counterexample:

1. H_2 consists of one hypothesis that always predicts positive.
2. H_3 consists of one hypothesis that always predicts negative.
3. $VC(H_2) < 1$ because H_2 can never correctly classify 1 point when that point has a negative label.
4. $VC(H_3) < 1$ because H_3 can never correctly classify 1 point when that point has a positive label.
5. $VC(H_1) \geq 1$ because we use the hypothesis from H_2 when the one point is positive, and we use the hypothesis from H_3 when the one point is negative.

6. By (3)-(5), the claim is false because…
   $VC(H_1) \geq 1 > VC(H_2) + VC(H_3)$
   $VC(H_1) > VC(H_2) + VC(H_3)$

Problem 2: Kernels

(a) Problem 2a

Solution:

YES, this function is a kernel.

Proof…

From lecture 11, a kernel function is a bivariate function that satisfies the following 2 properties:

1. $k(x,z) = k(z,x)$
2. $k(x,z) = \phi(x)^T \phi(z)$ for some function $\phi(\cdot)$

We can verify that both properties (1) and (2) are true:

1. Property 1:
    1. $k(x,z) = |x \cap z| = |z \cap x| = k(z,x)$
    2. $\therefore$ the function satisfies property 1
2. Property 2:
    1. $|x \cup z| = M$ and define $W = \{w_1, w_2, \ldots, w_i, \ldots, w_M\}$ to be an arbitrarily ordered list of all words in $x \cup z$.
    2. Define $\phi(\cdot) = \begin{bmatrix}\phi_1 & \phi_2 & \ldots & \phi_M\end{bmatrix}$ such that for each ϕ_i in $\phi(y)$,
       $$\phi_i(y) = \begin{cases} 1 & w_i \in y \\ 0 & w_i \notin y \end{cases}$$
    3. By (2), $\phi(x)^T \phi(z) = \sum_{i=1}^{M} \phi_i(x)\phi_i(z)$ where each $\phi_i(x)\phi_i(z)$ has the value…
       $$\phi_i(x)\phi_i(z) = \begin{cases} 1 & (w_i \in x) \wedge (w_i \in z) \\ 0 & else \end{cases}$$
    4. By (3), $\phi(x)^T \phi(z) = |x \cap z| = k(x,z)$
    5. By (4), $k(x,z) = \phi(x)^T \phi(z)$ for the function $\phi(\cdot)$ defined in (2)
    6. $\therefore$ the function satisfies property 2

The function satisfies the required properties of a kernel.
∴ the function is a kernel function.

(b) Problem 2b

Solution:

We first show that $k(x, z) = 1$ is a valid kernel:

From lecture 11, a kernel function is a bivariate function that satisfies the following 2 properties:

1. $k(x, z) = k(z, x)$
2. $k(x, z) = \phi(x)^T \phi(z)$ for some function $\phi(\cdot)$

We can verify that both properties (1) and (2) are true:

1. Property 1:
    1. $k(x, z) = 1 = k(z, x)$
    2. ∴ the function satisfies property 1
2. Property 2:
    1. Define $\phi(x) = [1 \quad 0]^T$
    2. $\phi(x)^T \phi(z) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1 + 0 = 1 = k(x, z)$
    3. By (2.2), $k(x, z) = \phi(x)^T \phi(z)$ for the function $\phi(\cdot)$ defined in (2.1)
    4. ∴ the function satisfies property 2

The function satisfies the required properties of a kernel.
∴ $k(x, z) = 1$ is a kernel.

We now show that the given equation is a kernel:

$$\begin{aligned}\left(1+\left(\frac{x}{\|x\|}\right)\cdot\left(\frac{z}{\|z\|}\right)\right)^3 &= \left(k_1(x,z)+\frac{1}{\|x\|}(x\cdot z)\frac{1}{\|z\|}\right)^3\\ &= \left(k_1(x,z)+\frac{1}{\|x\|}\cdot k_2(x,z)\cdot\frac{1}{\|z\|}\right)^3\\ &= \left(k_1(x,z)+k_3(x,z)\right)^3\\ &= \left(k_4(x,z)\right)^3\\ &= \left(k_4(x,z)\cdot k_4(x,z)\right)\cdot k_4(x,z)\\ &= k_5(x,z)\cdot k_4(x,z)\\ &= k_6(x,z)\\ &= \text{a valid kernel}\end{aligned}$$

Where $k_1(x,z), k_2(x,z), \ldots, k_6(x,z)$ are valid kernels because they are defined as follows:

$$\begin{aligned}k_1(x,z) &= 1\\ k_2(x,z) &= x\cdot z\\ k_3(x,z) &= f(x)\cdot k_2(x,z)\cdot f(z) \quad \text{where } f(x)=\frac{1}{\|x\|}\\ &= \frac{1}{\|x\|}\cdot k_2(x,z)\cdot\frac{1}{\|z\|}\\ k_4(x,z) &= k_1(x,z)+k_3(x,z)\\ k_5(x,z) &= k_4(x,z)\cdot k_4(x,z)\\ k_6(x,z) &= k_5(x,z)\cdot k_4(x,z)\end{aligned}$$

(c) Problem 2c

Solution:

The footnote in the directions state that we “may use any external program to expand the cubic”, so I used Wolfram Alpha to get the expanded form in the steps below.

$$
\begin{aligned}
k_\beta(x,z) &= (1+\beta x\cdot z)^3 \\
&= (1+\beta(x_1z_1+x_2z_2))^3 \\
&= (1+\beta x_1z_1+\beta x_2z_2)^3
\end{aligned}
$$

(expanded using Wolfram Alpha)

$$
\begin{aligned}
&= 1+3\beta x_1z_1+3\beta x_2z_2+3\beta^2x_1^2z_1^2+3\beta^2x_2^2z_2^2+6\beta^2x_1x_2z_1z_2 \\
&\quad +3\beta^3x_1^2x_2z_1^2z_2+3\beta^3x_1x_2^2z_1z_2^2+\beta^3x_1^3z_1^3+\beta^3x_2^3z_2^3 \\
&= \begin{bmatrix} 1 \\ \sqrt{3\beta}\cdot x_1 \\ \sqrt{3\beta}\cdot x_2 \\ \sqrt{3\beta^2}\cdot x_1^2 \\ \sqrt{3\beta^2}\cdot x_2^2 \\ \sqrt{6\beta^2}\cdot x_1x_2 \\ \sqrt{3\beta^3}\cdot x_1^2x_2 \\ \sqrt{3\beta^3}\cdot x_1x_2^2 \\ \sqrt{\beta^3}\cdot x_1^3 \\ \sqrt{\beta^3}\cdot x_2^3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \sqrt{3\beta}\cdot z_1 \\ \sqrt{3\beta}\cdot z_2 \\ \sqrt{3\beta^2}\cdot z_1^2 \\ \sqrt{3\beta^2}\cdot z_2^2 \\ \sqrt{6\beta^2}\cdot z_1z_2 \\ \sqrt{3\beta^3}\cdot z_1^2z_2 \\ \sqrt{3\beta^3}\cdot z_1z_2^2 \\ \sqrt{\beta^3}\cdot z_1^3 \\ \sqrt{\beta^3}\cdot z_2^3 \end{bmatrix}
\end{aligned}
$$

From the above analysis, we see that:

$$\phi(x) = \begin{bmatrix} 1 \\ \sqrt{3\beta} \cdot x_1 \\ \sqrt{3\beta} \cdot x_2 \\ \sqrt{3\beta^2} \cdot x_1^2 \\ \sqrt{3\beta^2} \cdot x_2^2 \\ \sqrt{6\beta^2} \cdot x_1 x_2 \\ \sqrt{3\beta^3} \cdot x_1^2 x_2 \\ \sqrt{3\beta^3} \cdot x_1 x_2^2 \\ \sqrt{\beta^3} \cdot x_1^3 \\ \sqrt{\beta^3} \cdot x_2^3 \end{bmatrix}$$

What are the similarities and differences from the kernel $k(x,z) = (1 + x \cdot z)^3$?

$k(x,z) = k_\beta(x,z)$ with $\beta = 1$

Thus, the kernels are the same except for the fact that β is fixed at $\beta = 1$ for $k(x,z)$, in contrast to the fact that β can be any value $\beta > 0$ for $k_\beta(x,z)$.

What role does the parameter β play?

β has a larger exponent for higher order product terms of x_1 and x_2, so it creates a larger coefficient in front of higher order terms compared to lower order ones. The effect is that β weights higher ordered terms more heavily in the training process, so they will have a greater weight when predicting labels compared to lower ordered terms, at least in the early stages of the training process.

Problem 3: SVM

(a) Problem 3a

Solution:

$\theta* = \left[-\frac{a}{a^2+e^2} \quad -\frac{e}{a^2+e^2}\right]^T$, as is found through the following analysis…

1. The classification boundary must pass through the origin, so…
   Margin $\leq$ the distance from the origin to point (a, e)
2. We seek to maximize the margin, which occurs when…
   Margin $=$ the distance from the origin to point (a, e)
3. (2) is true when the decision boundary is perpendicular to the straight line passing from the origin to point (a, e).
4. By (3), the decision boundary is perpendicular to the vector $[a \quad e]^T$
5. By (4), $\theta*$ is parallel to the vector $[a \quad e]^T$
6. Since $y = -1$, $\theta*$ points in the opposite direction of $[a \quad e]^T$
7. By (5) and (6), $\theta* = [-Ca \quad -Ce]^T$, where C is some scaling factor found using the constraint.
8. We compute C as follows:

$$\begin{aligned} y_1\theta^T x_1 &\geq 1 \\ (-1)(-Ca^2 - Ce^2) &\geq 1 \\ Ca^2 + Ce^2 &\geq 1 \\ Ca^2 + Ce^2 &= 1 \quad \text{(margin is optimized on equality)} \\ C &= \frac{1}{a^2+e^2} \end{aligned}$$

9. By (7) and (8), $\theta* = [-Ca \quad -Ce]^T = \left[-\frac{a}{a^2+e^2} \quad -\frac{e}{a^2+e^2}\right]^T$

$\therefore$ From the argument above, $\theta* = \left[-\frac{a}{a^2+e^2} \quad -\frac{e}{a^2+e^2}\right]^T$

(b) Problem 3b

Solution:

$\theta^* = [-1 \quad 2]^T$ and $\gamma = \dfrac{1}{\sqrt{5}}$, as found below…

The boundary must pass between x_1 and x_2 in order to classify them correctly.

The margin is maximized when the distance from x_1 to the margin is equivalent to the distance from x_2 to the margin, where both distances satisfy the constraints and minimize $||\theta||$.

Since both x_1 and x_2 must satisfy their constraints, and the margin is maximized when the constraints satisfy equality, we generate 2 equations as follows…

$$y_1\theta^T x_1 \geq 1$$
$$\theta_1 + \theta_2 \geq 1$$
$$\theta_1 + \theta_2 = 1 \text{ (margin maximized on equality)}$$

$$y_2\theta^T x_2 \geq 1$$
$$(-1)\theta_1 \geq 1$$
$$\theta_1 \leq -1$$
$$\theta_1 = -1 \text{ (margin maximized on equality)}$$

Solving the system of equations…

$$\theta_1 = -1$$

$$-1 + \theta_2 = 1 \longrightarrow \theta_2 = 2$$

Therefore,

$$\theta^* = [-1 \quad 2]^T$$

The margin is equivalent to the distance from either point to the line, which we can calculate as…

$$\gamma = \frac{y_1\left[\theta^T x_1\right]}{||\theta||} = \frac{1}{||\theta||} = \frac{1}{\sqrt{(-1)^2 + 2^2}} = \frac{1}{\sqrt{5}}$$

(c) Problem 3c

Solution:

$\theta^* = [0 \quad 2]^T$, $b^* = -1$, and $\gamma = \dfrac{1}{2}$, as shown below…

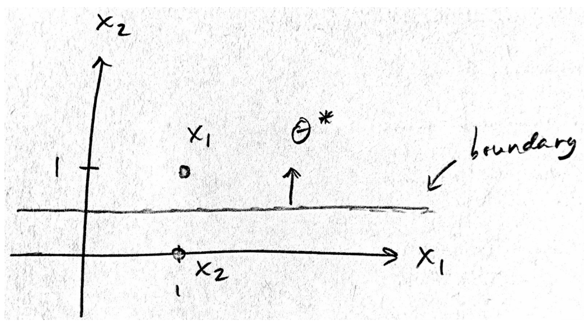
1. The two points have opposite labels, so the boundary must pass between them.
2. The margin is maximized when the boundary is equidistant from both points, so the boundary passes through the midpoint of the straight line between x_1 and x_2, and…

   margin $\leq$ distance from midpoint to x_1 = distance from midpoint to x_2

3. From (2), the margin is maximized when…

   margin $=$ distance from midpoint to x_1 = distance from midpoint to x_2

4. (3) occurs when the boundary is perpendicular to the straight line passing between x_1 and x_2
5. By (4), θ^* is parallel to the straight line passing between x_1 and x_2, pointing in the direction of the positive example x_1.
6. By (1)-(5), the boundary geometrically appears as follows:



7. By (5), $\theta^* = \left[C(1-1) \quad C(1-0)\right]^T = [0 \quad C]^T$ for some scaling factor C.
8. We generate a system of equations from the constraints on x_1 and x_2…

$$y_1(\theta^T x_1 + b) \geq 1$$
$$C + b \geq 1$$
$$C + b = 1 \text{ (margin maximized on equality)}$$

$$y_2(\theta^T x_2 + b) \geq 1$$
$$-b \geq 1$$
$$-b = 1 \text{ (margin maximized on equality)}$$
$$b = -1$$

9. Solving the system of equations, we get…

$$b = -1$$
$$C - 1 = 1 \quad \longrightarrow \quad C = 2$$

From the above analysis,

$$\theta* = [0 \quad 2]^T$$
$$b* = -1$$

The margin is…

$$\gamma = \frac{y_1(\theta^T x_1 + b)}{||\theta||} = \frac{2-1}{\sqrt{0^2 + 2^2}} = \frac{1}{2}$$

“Compare your solutions with and without offset”…

The solution with the offset has a better (i.e. greater) margin of $\frac{1}{2}$ compared to the prior margin of $\frac{1}{\sqrt{5}}$.

This makes sense because the new flexibility of the $b*$ value allows us to reposition the boundary in such a way that it is perpendicular to the straight line passing from x_1 to x_2, which visually and geometrically yields a boundary with a greater margin than any boundary passing through the origin.
The classifier and margin changed from the previous question because the decision boundary no longer passes through the origin and is at a different angle relative to the x_1-x_2 plane, and the margin has increased.

Problem 4: Implementation: Digit Recognizer

(a) Problem 4a

Solution:

Code in “main()” to specify the correct directory path:

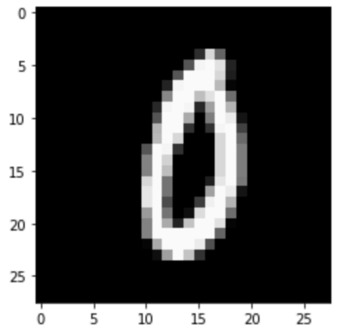
```
### ========== TODO : START ========== ###
data_directory_path =  "/content/drive/My Drive/schoolwork/3rd_year/winter/cs_m146/hw_3"
### ========== TODO : END ========== ###
```

Code in “main()” to randomly select three training examples with different labels and print their images:

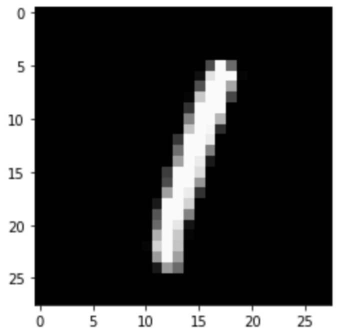
```
### ========== TODO : START ========== ###
### part a: print out three training images with different labels

# randomly select one training example from each label choice
for label in np.sort(np.unique(y_train)):
  img_index = np.random.choice(np.where(y_train == label)[0])
  plot_img(X_train[img_index])

### ========== TODO : END ========== ###
```

Training example images:

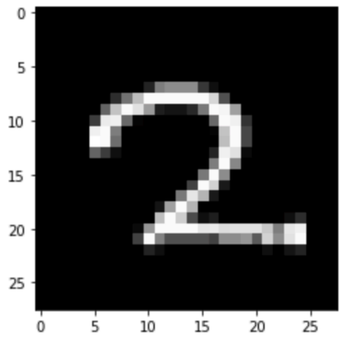
For label 0:



For label 1:



For label 2:



(b) Problem 4b

Solution:

Code in “main()” to convert np arrays to tensors:

```
### ========== TODO : START ========== ###
### part b: convert numpy arrays to tensors
X_train = torch.from_numpy(X_train)
y_train = torch.from_numpy(y_train)
X_valid = torch.from_numpy(X_valid)
y_valid = torch.from_numpy(y_valid)
X_test = torch.from_numpy(X_test)
y_test = torch.from_numpy(y_test)
```

(c) Problem 4c

Solution:

Code in “main(…)” to prepare the loaders:

```
### ========== 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 = DataLoader(TensorDataset(X_train, y_train), batch_size=10)
valid_loader = DataLoader(TensorDataset(X_valid, y_valid), batch_size=10)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=10)

### ========== TODO : END ========== ###
```

(d) Problem 4d

Solution:

Code in “OneLayerNetwork.__init__(…)”:

```
### ========== TODO : START ========== ###
### part d: implement OneLayerNetwork with torch.nn.Linear
self.linear1 = torch.nn.Linear(784, 3)

### ========== TODO : END ========== ###
```

Code in “OneLayerNetwork.forward(…)”:

```
### ========== TODO : START ========== ###
### part d: implement the foward function
outputs = self.linear1(x)

### ========== TODO : END ========== ###
```

(e) Problem 4e

Solution:

Code in “main(…)”:

```
### ========== 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 ========== ###
```

(f) Problem 4f

Solution:

Code in “train(…)”:

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

The results from running the given “train(…)” function call in “main(…)” are, in fact, similar to those shown in the directions. This is evident from comparing the output in the directions to the first 4 lines of the call’s output from “main(…)”:

```
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 |
```

(g) Problem 4g

Solution:

Code in “TwoLayerNetwork.__init__(…)”:

```
### ========== TODO : START ========== ###
### part g: implement TwoLayerNetwork with torch.nn.Linear
self.linear1 = torch.nn.Linear(784, 400)
self.linear2 = torch.nn.Linear(400, 3)
### ========== TODO : END ========== ###
```

Code in “TwoLayerNetwork.forward(…)”:

```
### ========== TODO : START ========== ###
### part g: implement the foward function
sigmoid = torch.nn.Sigmoid()
layer_1_outputs = sigmoid(self.linear1(x))
outputs = self.linear2(layer_1_outputs)
### ========== TODO : END ========== ###
```

(h) Problem 4h

Solution:

Code in “main(…)” to prepare for TwoLayerNetwork training:

```
### ========== 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 ========== ###
```

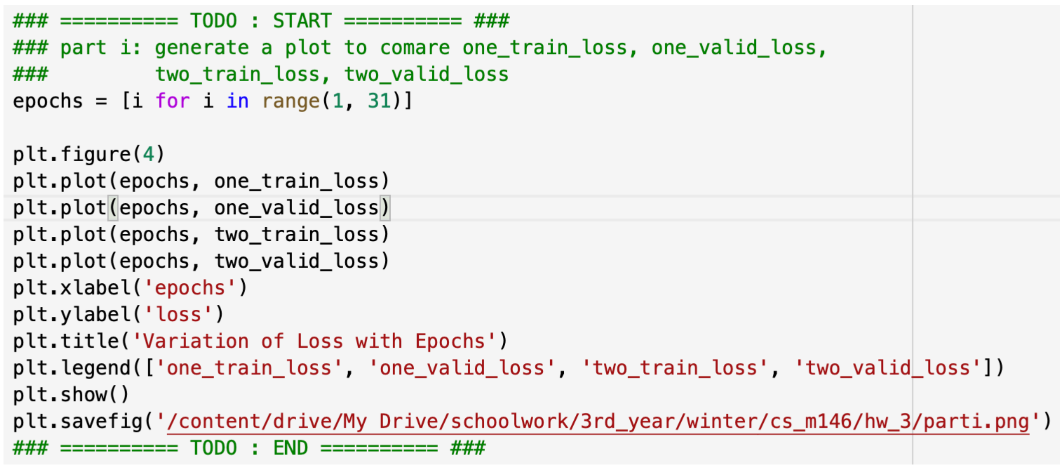
First few entries of the output of code in “main(…)”:

```
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 |
```

(i) Problem 4i

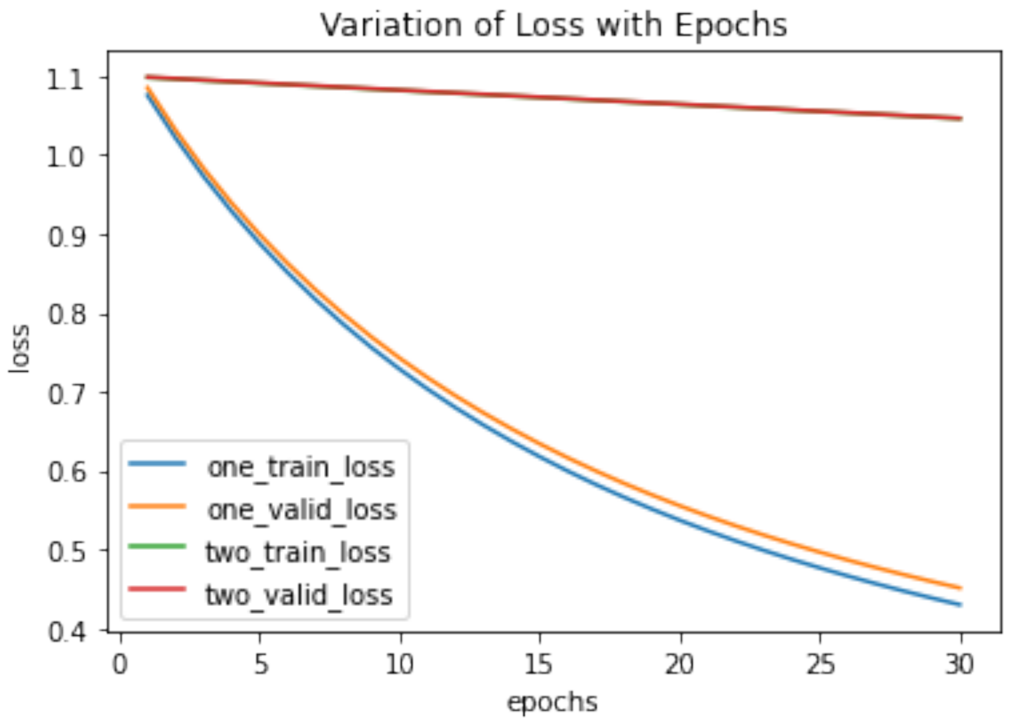
Solution:

Code in “main(…)” to generate the plot:



```
### ========== TODO : START ========== ###
### part i: generate a plot to comare one_train_loss, one_valid_loss,
###         two_train_loss, two_valid_loss
epochs = [i for i in range(1, 31)]

plt.figure(4)
plt.plot(epochs, one_train_loss)
plt.plot(epochs, one_valid_loss)
plt.plot(epochs, two_train_loss)
plt.plot(epochs, two_valid_loss)
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('Variation of Loss with Epochs')
plt.legend(['one_train_loss', 'one_valid_loss', 'two_train_loss', 'two_valid_loss'])
plt.show()
plt.savefig('/content/drive/My Drive/schoolwork/3rd_year/winter/cs_m146/hw_3/parti.png')
### ========== TODO : END ========== ###
```

Plot generated by the code in “main(…)”:



“Describe your findings”:

The initial loss values (around epoch 0) of the 1 and 2-layer neural nets are approximately equivalent because both models misclassify many points.

The loss values decrease for both models, but with different rates and characteristics. The train and valid loss for 1-layer decreases in more of an exponential fashion, and the value of both loss curves for the 1-layer network are strictly less than the values of the 2-layer curves at each epoch.

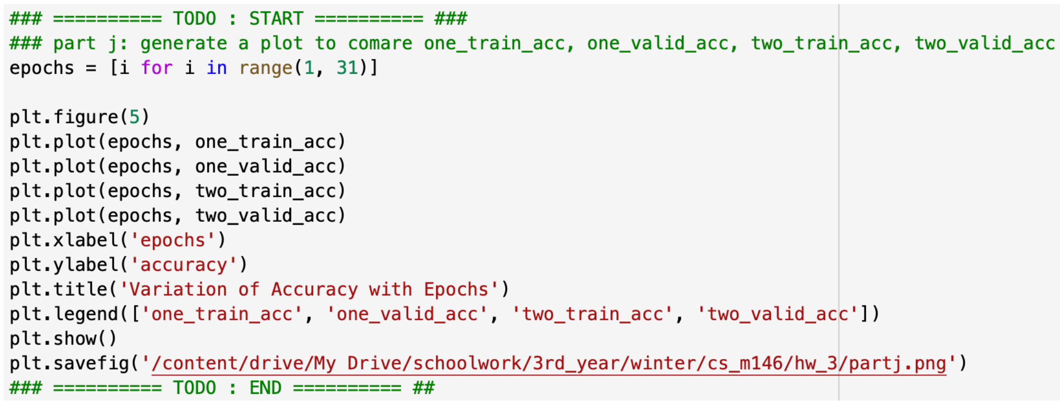
The 2-layer curves appear to decrease in more of a linear fashion, at least based on the epoch range analyzed in this plot. The rate of decrease for both 2-layer curves is much less than that of the 1-layer curves at every epoch value in the displayed range.

Lastly, the two curves for 2-layer approximately overlap, while the valid loss curve for 1-layer is strictly less than the train loss curve for 1-layer. This makes sense because the model is being updated via SGD strictly based on the training dataset, so the parameters are naturally more fitted to the training set (i.e. have lesser loss on the training dataset) compared to the validation set.

(j) Problem 4j

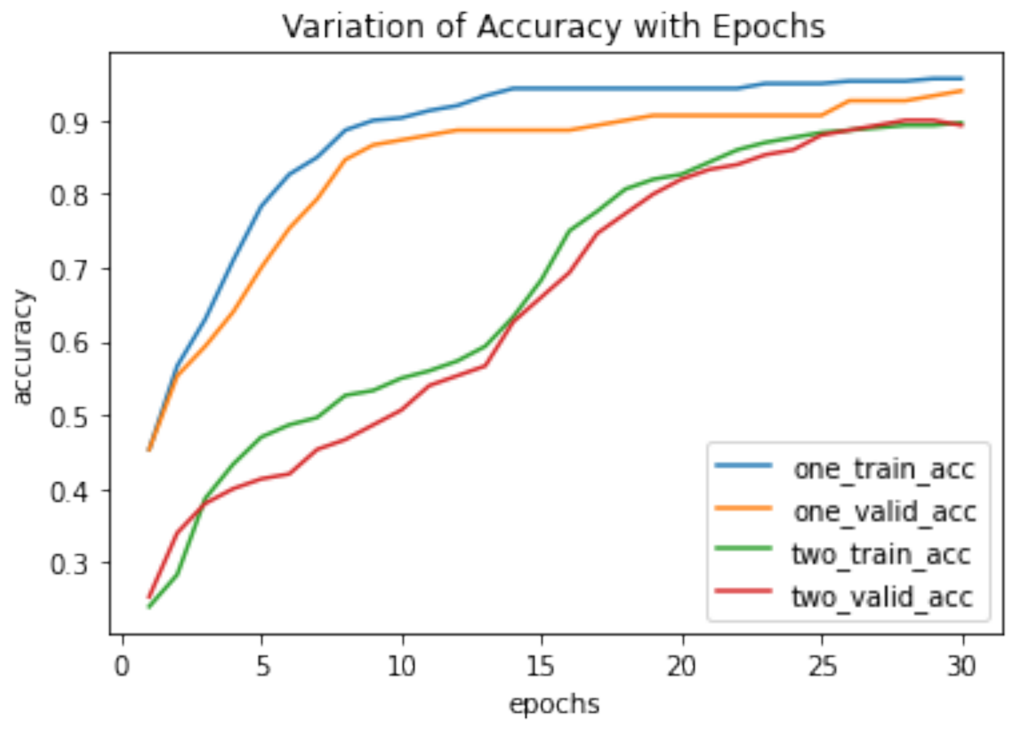
Solution:

Code in “main(…)” to generate the plot:



```
### ========== TODO : START ========== ###
### part j: generate a plot to comare one_train_acc, one_valid_acc, two_train_acc, two_valid_acc
epochs = [i for i in range(1, 31)]

plt.figure(5)
plt.plot(epochs, one_train_acc)
plt.plot(epochs, one_valid_acc)
plt.plot(epochs, two_train_acc)
plt.plot(epochs, two_valid_acc)
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.title('Variation of Accuracy with Epochs')
plt.legend(['one_train_acc', 'one_valid_acc', 'two_train_acc', 'two_valid_acc'])
plt.show()
plt.savefig('/content/drive/My Drive/schoolwork/3rd_year/winter/cs_m146/hw_3/partj.png')
### ========== TODO : END ========== ##
```

Plot generated by code in “main(…)”:



"Describe your findings":

The two accuracy curves for 1-layer are strictly greater in accuracy value compared to the two accuracy curves for 2-layer, indicating that the 1-layer network has greater accuracy on both the training and validation set at every epoch in the given range.

The validation accuracy for 1-layer is strictly less than the training accuracy for 1-layer at every epoch value, which makes sense because the model is being updated via SGD strictly based on the training dataset, so the parameters are naturally more fitted to the training set (i.e. have greater accuracy on the training set) compared to the validation set.

For the same reason, the validation accuracy for 2-layer is less than the training accuracy for 2-layer at almost all epoch values.

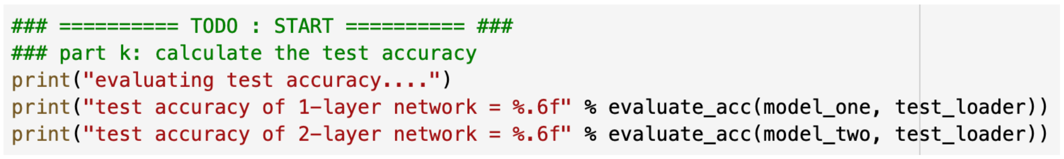
The accuracy curves for 1-layer increase at a much greater rate at early epoch values compared to the accuracy curves for 2-layer. The 1-layer curves increase rapidly, then approximately asymptotically approach a high accuracy value, while the accuracy curves for 2-layer increase in a relatively linear fashion for the given range of epoch values, beginning to taper off asymptotically for the later epoch values.

Overall, the 1-layer network approaches high accuracy values much more quickly than the 2-layer, and it attains a final accuracy that is greater on both data sets for the given range of epoch values.

(k) Problem 4k

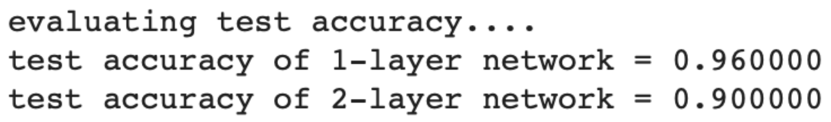
Solution:

Code in "main(…)" to report the test accuracy of both the one-layer network and the two-layer network:



```
### ========== TODO : START ========== ###
### part k: calculate the test accuracy
print("evaluating test accuracy....")
print("test accuracy of 1-layer network = %.6f" % evaluate_acc(model_one, test_loader))
print("test accuracy of 2-layer network = %.6f" % evaluate_acc(model_two, test_loader))
```

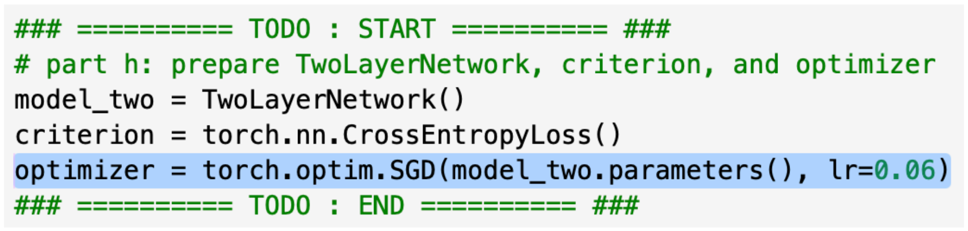
Output of code in "main(…)" to report the test accuracy:



```
evaluating test accuracy....
test accuracy of 1-layer network = 0.960000
test accuracy of 2-layer network = 0.900000
```

"How can we improve the performance of the 2-layer network?"

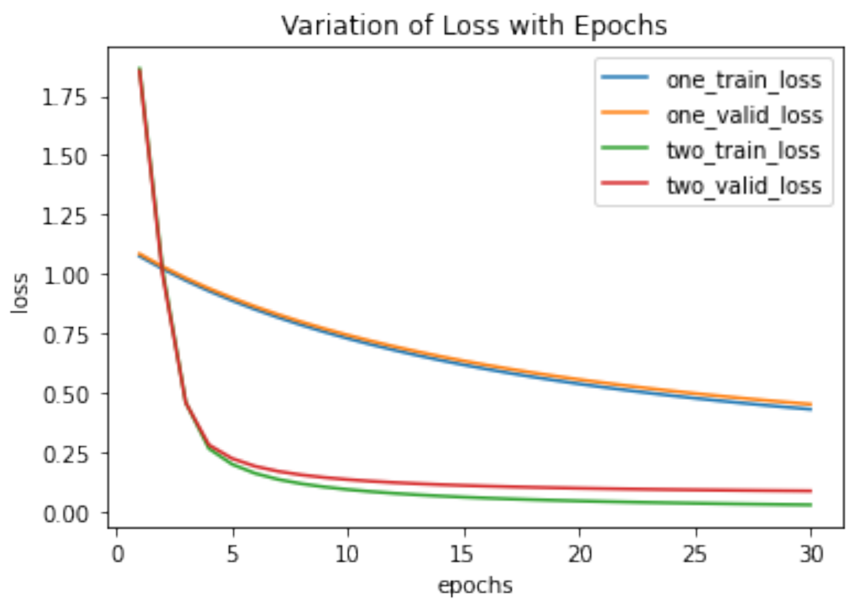
We can improve the performance by increasing the learning rate for the 2-layer network. I increased the learning rate from 0.0005 to 0.06, as shown in the highlighted line below:



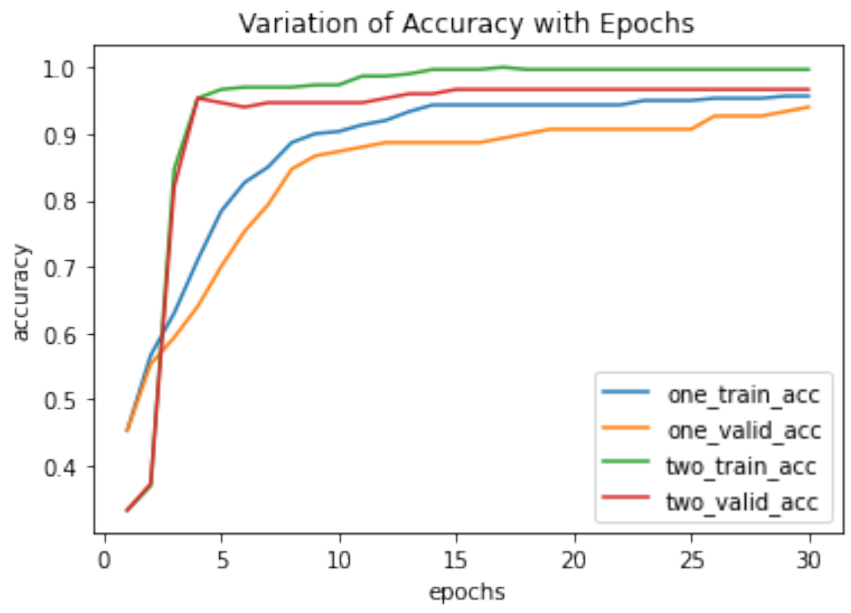
```
### ========== TODO : START ========== ###
# part h: prepare TwoLayerNetwork, criterion, and optimizer
model_two = TwoLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_two.parameters(), lr=0.06)
### ========== TODO : END ========== ###
```

The new results are as follows:

Loss figure:



Accuracy figure:



Test accuracy:

```
test accuracy of 1-layer network = 0.960000
test accuracy of Modified 2-layer network = 0.973333
```

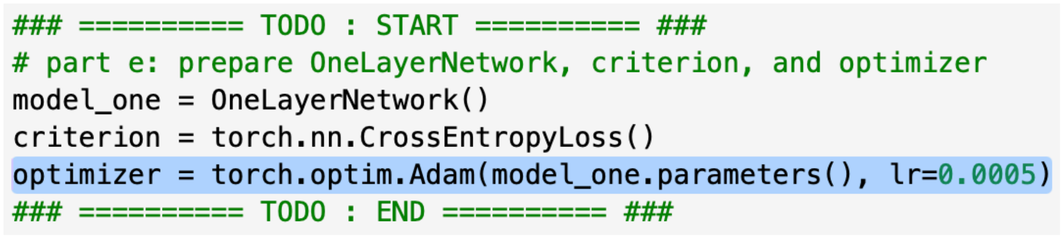
As seen in the images above, the test accuracy for the 2-layer network is now higher than that for the 1-layer network. The graphs also indicate that the performance for all epochs, both in loss and accuracy, is much greater than it was for the original learning rate.

This makes sense because the original graphs indicated that the loss was increasing very slowly and the accuracy was increasing very slowly for 2-layer. By speeding up the pace with which the 2-layer network approaches its final parameter values, we allow the network to achieve better parameter values (i.e. greater performance) in the same number of epochs.

(l) Problem 4l

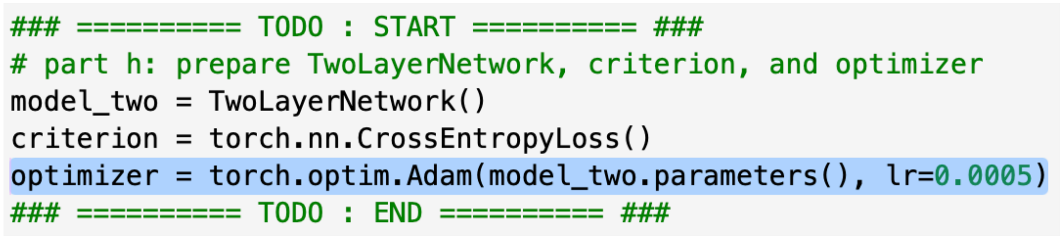
Solution:

Code changes from previous parts:
—> highlighted line below from part e:



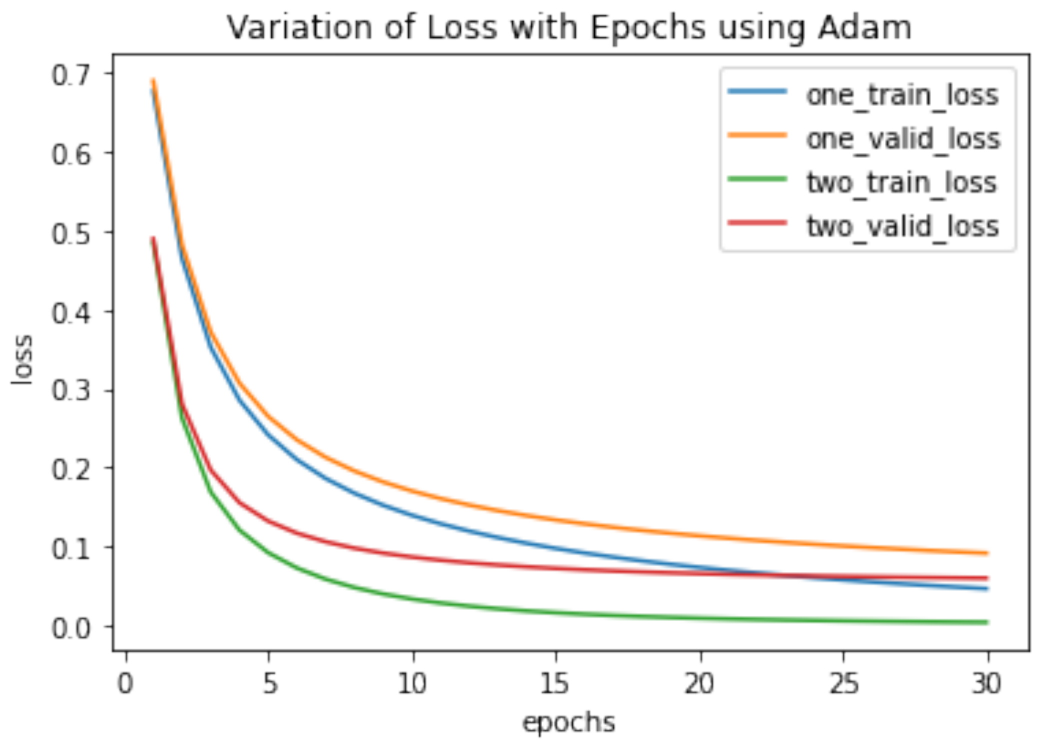
```
### ========== TODO : START ========== ###
# part e: prepare OneLayerNetwork, criterion, and optimizer
model_one = OneLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_one.parameters(), lr=0.0005)
### ========== TODO : END ========== ###
```

—> highlighted line below from part h:

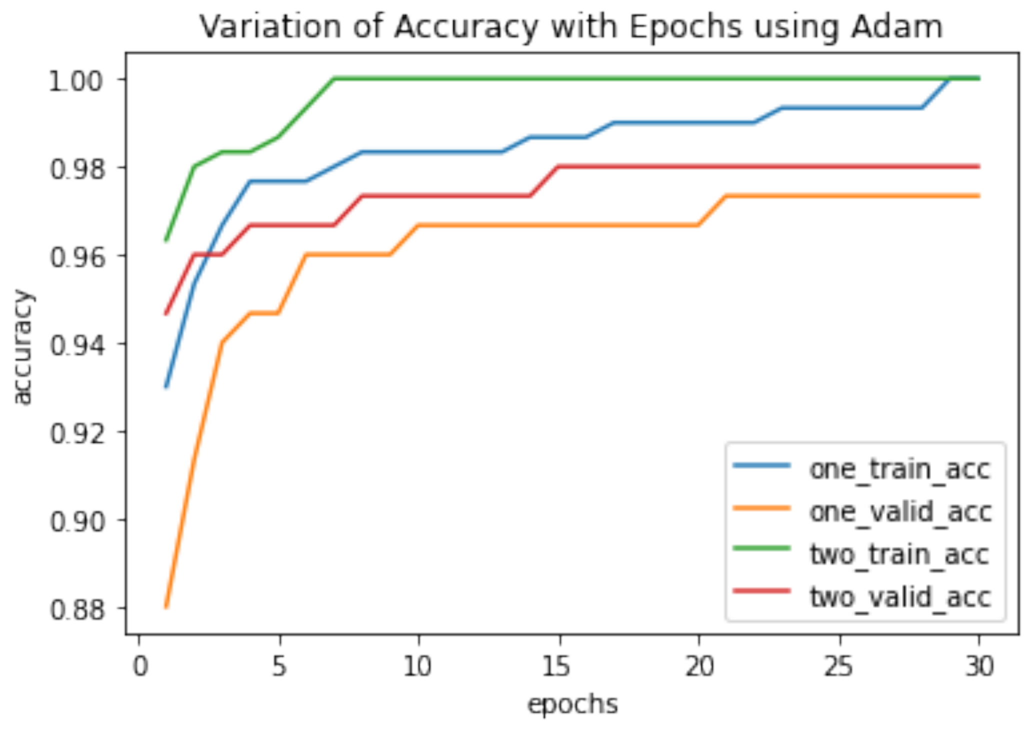


```
### ========== TODO : START ========== ###
# part h: prepare TwoLayerNetwork, criterion, and optimizer
model_two = TwoLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_two.parameters(), lr=0.0005)
### ========== TODO : END ========== ###
```

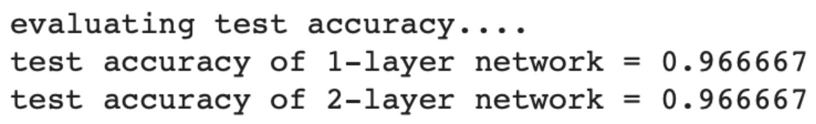
Loss figure:



Accuracy figure:



Test accuracy:



```
evaluating test accuracy....
test accuracy of 1-layer network = 0.966667
test accuracy of 2-layer network = 0.966667
```

“Describe your findings”:

For both 1-layer and 2-layer, the loss decreases much more quickly compared to the loss curves from SGD, and the accuracy increases much more quickly compared to the accuracy curves from SGD. The test accuracy values are also greater for both 1-layer and 2-layer compared to those achieve in SGD, and both networks converged to the same value of test accuracy.

For the 2-layer network, the relatively linear-shaped loss and accuracy curves from SGD now resemble the nonlinear shapes of the 1-layer loss and accuracy curves.

For both 1-layer and 2-layer, the Adam optimizer appears to approach the final parameter values very rapidly for early epoch values (i.e. it most likely uses a larger step size for early epoch values), then slows down as it approaches the final parameter values, so as to not overshoot or miss a close-to-optimal solution. Thus, it likely decreases the step size as the parameter values approach optimal.