Problem Set 3

Graded

Student

BRANDON LO

Total Points

63 / 64 pts

Question 1

Kernels **8** / 8 pts

1.1 a **2** / 2 pts

✔ **– 0 pts** Correct

– 2 pts Wrong final answer

– 1 pt partially correct solution

1.2 b **3** / 3 pts

✔ **– 0 pts** Correct

– 3 pts not attempted

1.3 c **3** / 3 pts

✔ **– 0 pts** Correct

– 3 pts not attempted

– 2 pts Incomplete mapping function

– 1 pt Incorrect or missing interpretation of Beta

– 1 pt minor error

Question 2

SVM **8** / 8 pts

2.1 **a** **2** / 2 pts

✔ **– 0 pts** Correct

– 1 pt Wrong final answer

– 1 pt Wrong argument or no justification

– 2 pts no answer

– 0.5 pts negative sign is missing

– 0.5 pts minor error

2.2 **b** **2** / 2 pts

✔ **– 0 pts** Correct

– 1 pt Wrong vector

– 1 pt Wrong/no argument

– 1 pt wrong margin

– 0.5 pts a minor error in the final answer

– 2 pts no answer

2.3 **c** **4** / 4 pts

✔ **– 0 pts** Correct

– 1 pt Wrong final answer for normal vector

– 1 pt Wrong bias

– 1 pt Wrong/missing comparison of margin

– 4 pts no answer

– 1 pt Wrong/no approach to compute the normal vector

– 1 pt small error

Question 3

Implementation: Digit Recognizer **47** / 48 pts

3.1 **a) Visualization of images** **2** / 2 pts

✔ **– 0 pts Correct**: Shows three training examples with all different images

– 1 pt Partial: Images given, but some have the same label

– 1 pt Partial: Code given, but no images

– 2 pts Incorrect: No attempt

3.2 **b) Convert arrays to tensors** **3** / 3 pts

✔ **– 0 pts** Correct

– 3 pts not attempt

3.3 **c) Dataloaders** **5** / 5 pts

✔ **– 0 pts** Correct

– 5 pts not attempted

3.4 **d) OneLayerNetwork** **5** / 5 pts

✔ **– 0 pts** Correct

– 5 pts not attempted

– 1 pt implementation error

3.5 **e) Create OneLayerNetwork, Criterion, Optimizer** **2** / 2 pts

✔ **– 0 pts** Correct

– 2 pts not attempted

3.6 **f) Implement training** **8** / 8 pts

✔ **– 0 pts** Correct

– 8 pts not attempted

3.7 **g) TwoLayerNetwork** **5** / 5 pts

✔ **– 0 pts** Correct

– 5 pts not attempted

– 2 pts missing activation / major error

– 1 pt minor error

3.8 **h) Create TwoLayerNetwork, Criterion, Optimizer** **2** / 2 pts

✔ **– 0 pts** Correct

– 2 pts not attempted

– 1 pt minor error

3.9 **i) Loss Plot** **3** / 3 pts

✔ **– 0 pts** Correct

– 3 pts not attempted

– 1 pt minor error

– 2 pts major error

3.10 **j) Accuracy Plot** **3** / 3 pts

✔ **– 0 pts** **Correct**:
`OneLaverNet`'s training and validation accuracy slowly increase.
`TwoLaverNet`'s training and validation accuracy increase with dips—generally lower than `OneLayerNet`.

– 1 pt **Partial**: Minor issues; potentially incorrect/missing nonlinearity between `TwoLayerNet`'s layers

– 1 pt **Partial:** Analysis given, but no graph

– 2 pts **Partial:** Major issues with graph

– 3 pts **Incorrect**: No attempt

3.11 **k) Test Accuracies** **2** / 3 pts

– 0 pts **Correct**: Within 1%
`OneLaverNet`: 96.0%
`TwoLayerNet`: 89.3%

– 0.5 pts **Partial**: Between 1-2% off on either or both nets

✔ **– 1 pt** **Partial**: Between 2-6% off on either or both nets

– 2 pts **Partial**: Greater than 6% off on either or both nets

– 3 pts **Incorrect**: No attempt

3.12 l) Try with Adam optimizer instead — **7** / 7 pts

✔ **– 0 pts** **Correct:**

Plots:

- Loss plot has the training and validation curves for both nets decrease asymptotically
- Accuracy plot increasing asymptotically; validation for `TwoLayerNet` somewhat jagged

Accuracies (within 1%):

- `OneLayerNet`: 96.7%
- `TwoLayerNet`: 97.3%

Description of findings:

- Adam significantly improved `TwoLayerNet`'s performance
- While `OneLayerNet` also improved, `TwoLayerNet` improved much more

– 1 pt **Partial**: Loss plot has issues

– 1 pt **Partial**: Accuracy plot has issues

– 1 pt **Partial**: Accuracy off by 1-2% for either or both nets

– 2 pts **Partial**: Accuracy off by greater than 2% for either or both nets or not given

– 1 pt **Partial**: Description of findings doesn't mention new optimizer

– 2 pts **Partial**: Description of findings missing or incorrect

– 7 pts **Incorrect**: No attempt

Question assigned to the following page: 1.1

1 Kernels [8 pts]

(a) **(2 pts)** For any two documents $\boldsymbol{x}$ and $\boldsymbol{z}$, define $k(\boldsymbol{x}, \boldsymbol{z})$ to equal the number of unique words that occur in both $\boldsymbol{x}$ and $\boldsymbol{z}$ (i.e., the size of the intersection of the sets of words in the two documents). Is this function a kernel? Give justification for your answer.

$$K(x,z) = \begin{bmatrix} \|x \cap x\| & \|x \cap z\| \\ \|z \cap x\| & \|z \cap z\| \end{bmatrix} = \begin{bmatrix} \|x\| & \|x \cap z\| \\ \|x \cap z\| & \|z\| \end{bmatrix}$$

Eigenvalues of K:

$$0 = (\|x\| - \lambda)(\|z\| - \lambda) - \|x \cap z\|^2$$

$$0 = \lambda^2 + \|x\|\|z\| - \lambda\|z\| - \lambda\|x\| - \|x \cap z\|^2$$

$$0 = \lambda^2 + \|x\|\|z\| - \lambda(\|z\| + \|x\|) - \|x \cap z\|^2$$

$$\lambda = \frac{-(-(\|z\| + \|x\|)) \pm \sqrt{-(\|z\| + \|x\|)^2 - 4(\|x\|\|z\| - \|x \cap z\|^2}}{2}$$

$$\|x\|\|z\| \geq \|x \cap z\|^2$$

We can therefore say that all eigenvalues of $K(x,z)$ are non-negative, so it is positive semi-definite. Therefore, $K(x,z)$ is a kernel function

Question assigned to the following page: 1.2

(b) **(3 pts)** One way to construct kernels is to build them from simpler ones. We have seen various "construction rules", including the following: Assuming $k_1(\boldsymbol{x}, \boldsymbol{z})$ and $k_2(\boldsymbol{x}, \boldsymbol{z})$ are kernels, then so are

- (scaling) $f(\boldsymbol{x})k_1(\boldsymbol{x}, \boldsymbol{z})f(\boldsymbol{z})$ for any function $f(\boldsymbol{x}) \in \mathbb{R}$
- (sum) $k(\boldsymbol{x}, \boldsymbol{z}) = k_1(\boldsymbol{x}, \boldsymbol{z}) + k_2(\boldsymbol{x}, \boldsymbol{z})$
- (product) $k(\boldsymbol{x}, \boldsymbol{z}) = k_1(\boldsymbol{x}, \boldsymbol{z})k_2(\boldsymbol{x}, \boldsymbol{z})$

Using the above rules and the fact that $k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x} \cdot \boldsymbol{z}$ is (clearly) a kernel, show that the following is also a kernel:

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

We know that if $k(x,z) = x \cdot z$ is a kernel, by the scaling property:

$$k_1(x,z) = \frac{1}{||x||}\frac{1}{||z||}k(x,z) = \frac{x \cdot z}{||x||\,||z||}$$

We know that 1 is a kernel function, using the sum property:

$1 + \frac{x \cdot z}{||x||\,||z||}$ is also a kernel

Using product property:

$$\left(1 + \frac{x \cdot z}{||x||\,||z||}\right)\left(1 + \frac{x \cdot z}{||x||\,||z||}\right)\left(1 + \frac{x \cdot z}{||x||\,||z||}\right) = \left(1 + \frac{x \cdot z}{||x||\,||z||}\right)^3$$

is also a kernel.

Therefore, $\left(1 + \left(\frac{x}{||x||}\right) \cdot \left(\frac{z}{||z||}\right)\right)^3$ is a kernel.

Question assigned to the following page: 1.3

(c) **(3 pts)** Given vectors $\boldsymbol{x}$ and $\boldsymbol{z}$ in $\mathbb{R}^2$, define the kernel $k_\beta(\boldsymbol{x}, \boldsymbol{z}) = (1 + \beta \boldsymbol{x} \cdot \boldsymbol{z})^3$ for any value $\beta > 0$. Find the corresponding feature map $\phi_\beta(\cdot)$[1]. What are the similarities/differences from the kernel $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^3$, and what role does the parameter β play?

$$(1+\beta x\cdot z)^3 = 1+3\beta x\cdot z + 3\beta^2(x\cdot z)^2 + \beta^3(x\cdot z)^3$$

Expand: $1+3\beta x_1 z_1 + 3\beta x_2 z_2 + 3\beta^2(x_1^2 z_1^2 + 2x_1 x_2 z_1 z_2 + x_2^2 z_2^2)$
$+\beta^3(x_1^3 x_2^3 + 3x_1^2 x_2^2 z_1 z_2 + 3x_1 x_2 z_1^2 z_2^2 + z_1^3 z_2^3)$

$$\phi = \begin{bmatrix} 1 \\ \sqrt{3\beta}\, x_1 \\ \sqrt{3\beta}\, x_2 \\ \sqrt{3\beta}\, x_1^2 \\ \sqrt{6\beta}\, x_1 x_2 \\ \sqrt{3\beta}\, x_2^2 \\ \sqrt{\beta^3}\, x_1^3 \\ \sqrt{3\beta^3}\, x_1^2 x_2 \\ \sqrt{3\beta^3}\, x_1 x_2^2 \\ \sqrt{\beta^3}\, x_2^3 \end{bmatrix}$$

The kernel of $k(x,z) = (1+x\cdot z)^3$ is similar in that you have to find the dot product of x and z, but different in that it would not have the β term as $\beta = 1$ in this scenario. β increases as the degree of the transformation vector increases, so it can be used as a regularization parameter.

Question assigned to the following page: 2.1

2 SVM [8 pts]

Suppose we are looking for a maximum-margin linear classifier *through the origin*, i.e. $b = 0$ (also hard margin, i.e., no slack variables). In other words, we minimize $\frac{1}{2}||\boldsymbol{\theta}||^2$ subject to $y_n \boldsymbol{\theta}^T \boldsymbol{x}_n \geq 1, n = 1, \ldots, N$.

(a) **(2 pts)** Given a single training vector $\boldsymbol{x} = (a, e)^T$ with label $y = -1$, what is the $\boldsymbol{\theta}^*$ that satisfies the above constrained minimization?

We use Lagrange multipliers: $f(\theta) = \frac{1}{2}\|\theta\|^2$

$g(\theta) = y_n \theta^T x_n \geq 1$

For a single training vector $x = (a\ e)^T$ with label $y = -1$:

$f(\theta) = \frac{1}{2}\|\theta\|^2$

$g(\theta) = -\theta^T x_n \geq 1$

The Lagrangian: $L = \frac{1}{2}\|\theta\|^2 - \lambda(-\theta^T x_n - 1) = \frac{1}{2}\|\theta\|^2 + \lambda(\theta^T x_n + 1)$

To find the minimum, we take the derivative:

$\frac{\partial L}{\partial \theta} = \theta + \lambda x_n = 0$

$\theta^* = -\lambda x_n$

Substituting and taking the derivative to maximize λ:

$L = \frac{1}{2}\|-\lambda x_n\|^2 + \lambda((-\lambda x_n)^T x_n + 1)$

$= \frac{1}{2}\lambda^2\|x_n\|^2 - \lambda^2\|x_n\|^2 + \lambda = -\frac{1}{2}\lambda^2\|x_n\|^2 + \lambda$

$\frac{\partial L}{\partial \lambda} = -\lambda\|x_n\|^2 + 1 = 0$

$\lambda^* = \frac{1}{\|x_n\|^2}$

Substituting back into the θ^* equation: $\theta^* = -\frac{1}{\|x_n\|^2} x_n = -\frac{x}{\|x\|^2}$

$\theta^* = -\frac{1}{a^2 + e^2}\begin{bmatrix} a \\ e \end{bmatrix}$

Questions assigned to the following page: 2.2 and 2.3

(b) **(2 pts)** Suppose we have two training examples, $\boldsymbol{x}_1 = (1,1)^T$ and $\boldsymbol{x}_2 = (1,0)^T$ with labels $y_1 = 1$ and $y_2 = -1$. What is $\boldsymbol{\theta}^*$ in this case, and what is the margin γ?

$f(\theta) = \frac{1}{2}\|\theta\|^2$

$g_1(\theta) = \theta^T x_1 \geq 1$

$g_2(\theta) = \theta^T x_2 \leq -1$

$\theta_1 + \theta_2 \geq 1$ and $\theta_1 \leq -1$

We pick $\theta_1 = 2, \theta_2 = -1$

Margin: $\frac{1}{\|\theta\|_2} = \frac{\sqrt{5}}{5}$

$\theta^* = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$, $\gamma = \frac{1}{\sqrt{5}}$

(c) **(4 pts)** Suppose we now allow the offset parameter b to be non-zero. How would the classifier and the margin change in the previous question? What are $(\boldsymbol{\theta}^*, b^*)$ and γ? Compare your solutions with and without offset.

$\theta_1 + \theta_2 + b \geq 1$ and $\theta_1 + b \leq -1$

We set $b = -1$ to minimize the magnitude of θ

$(\theta^*, b^*) = ([0,2]^T, -1)$, and $\gamma = 0.5$

The magnitude is smaller than without the offset, but the margin is larger. This is because there are more hyperplanes available so the classifier can find a better one which has a smaller magnitude and larger margin.

Questions assigned to the following page: 3.1 and 3.2

3 Implementation: Digit Recognizer [48 pts]

In this exercise, you will implement a digit recognizer in pytorch. Our data contains pairs of 28×28 images $\mathbf{x}_n$ and the corresponding digit labels $y_n \in \{0, 1, 2\}$. For simplicity, we view a 28×28 image $\mathbf{x}_n$ as a 784-dimensional vector by concatenating all the pixels together. In other words, $\mathbf{x}_n \in \mathbb{R}^{784}$. Your goal is to implement two digit recognizers (`OneLayerNetwork` and `TwoLayerNetwork`) and compare their performances.

---

code and data

- code : `CS146_Winter2024_PS3.ipynb`
- data : `ps3_train.csv`, `ps3_valid.csv`, `ps3_test.csv`

Please use your *@g.ucla.edu* email id to access the code and data. Similar to *PS1*, copy the colab notebook to your drive and make the changes. Mount the drive appropriately and copy the shared data folder to your drive to access via colab. The notebook has marked blocks where you need to code.

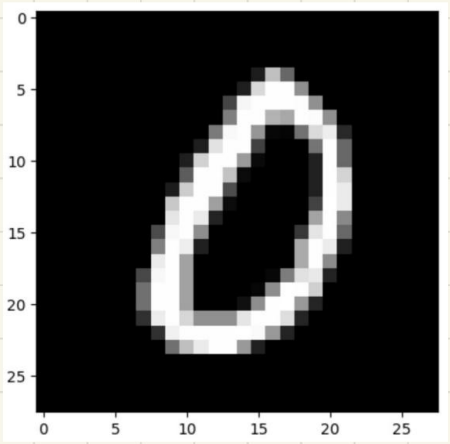
========== TODO : START ==========

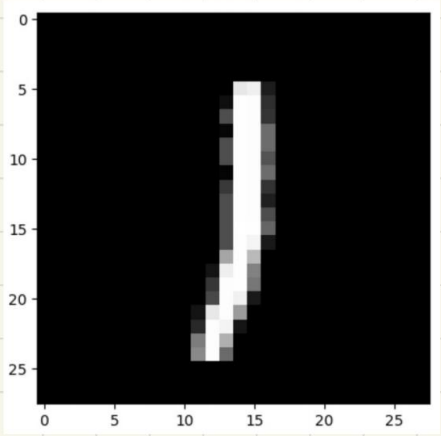
========== TODO : END ==========

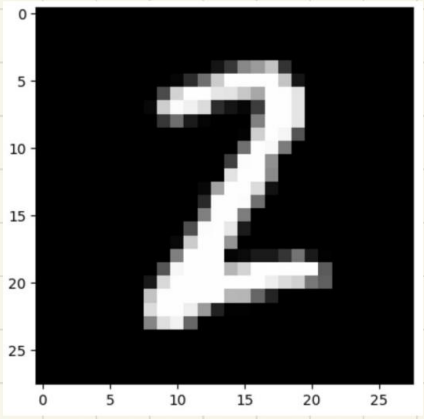
Note: For parts (b)-(h), you are expected to copy-paste your code as a part of the solution in the submission pdf. Tip: If you are using LaTeX, check out the Minted package (example) for code highlighting.

Data Visualization and Preparation [10 pts]

(a) **(2 pts)** Select three training examples with *different labels* and print out the images by using `plot_img` function. Include those images in your report.







(b) **(3 pts)** The loaded examples are numpy arrays. Convert the numpy arrays to PyTorch 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)
### ========== TODO : END ========== ###
```

Questions assigned to the following page: 3.3, 3.4, and 3.5

(c) **(5 pts)** Prepare `train_loader`, `valid_loader`, and `test_loader` by using `TensorDataset` and `DataLoader`. We expect to get a batch of pairs $(\mathbf{x}_n, y_n)$ from the dataloader. Please set the batch size to 10 for all dataloaders and shuffle to True for `train_loader`. We need to set shuffle to True for `train_loader` so that we are training on randomly selected minibatches of data.

You can refer https://pytorch.org/docs/stable/data.html for more information about `TensorDataset` and `DataLoader`.

```
### ========== 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 = ...
### valid_loader = ...
### test_loader = ...
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 ========== ###
```

One-Layer Network [15 pts]

For one-layer network, we consider a *784–3* network. In other words, we learn a 784×3 weight matrix $\mathbf{W}$. Given a $\mathbf{x}_n$, we can compute the probability vector $\mathbf{p}_n = \text{Softmax}(\mathbf{W}^\top \mathbf{x}_n)$, where $\mathbf{p}_{n,c}$ denotes the probability of class c. Then, we focus on the *cross entropy loss*

$$-\sum_{n=1}^{N}\sum_{c=1}^{C} \mathbf{1}(c = y_n) \log(\mathbf{p}_{n,c})$$

where N is the number of examples, C is the number of classes, and $\mathbf{1}$ is the indicator function.

(d) **(5 pts)** Implement the constructor of `OneLayerNetwork` with `torch.nn.Linear` and implement the `forward` function to compute the outputs of the single fully connected layer i.e. $\mathbf{W}^\top \mathbf{x}_n$. Notice that we do not compute the softmax function here since we will use `torch.nn.CrossEntropyLoss` later. The bias term is included in `torch.nn.Linear` by default, do *not* disable that option.

You can refer to https://pytorch.org/docs/stable/generated/torch.nn.Linear.html for more information about `torch.nn.Linear` and refer to https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html for more information about using `torch.nn.CrossEntropyLoss`.

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

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

(e) **(2 pts)** Create an instance of `OneLayerNetwork`, set up a criterion with `torch.nn.CrossEntropyLoss`, and set up a SGD optimizer with learning rate 0.0005 by using `torch.optim.SGD`.

You can refer to https://pytorch.org/docs/stable/optim.html for more information about `torch.optim.SGD`.

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

Questions assigned to the following page: 3.6, 3.7, and 3.8

(f) **(8 pts)** Implement the training process. This includes forward pass, initializing gradients to zeros, computing loss, loss.backward, and updating model parameters. If you implement everything correctly, after running the `train` function in main, you should get results similar to the following.

```
Start training OneLayerNetwork...
| epoch  1 | train loss 1.075380 | train acc 0.453333 | valid loss ...
| epoch  2 | train loss 1.021195 | train acc 0.553333 | valid loss ...
| epoch  3 | train loss 0.972527 | train acc 0.626667 | valid loss ...
| epoch  4 | train loss 0.928296 | train acc 0.710000 | valid loss ...
...
```

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

Two-Layer Network [7 pts]

For two-layer network, we consider a *784–400–3* network. In other words, the first layer will consist of a fully connected layer with 784×400 weight matrix $\mathbf{W}_1$ and a second layer consisting of 400×3 weight matrix $\mathbf{W}_2$. Given a $\mathbf{x}_n$, we can compute the probability vector $\mathbf{p}_n = \text{Softmax}(\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top \mathbf{x}_n))$, where $\sigma(.)$ is the element-wise sigmoid function. Again, we focus on the *cross entropy loss*, hence the network will implement $\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top \mathbf{x}_n)$ (note the softmax will be taken care of implicitly in our loss). The bias term is included in `torch.nn.Linear` by default, do *not* disable that option.

(g) **(5 pts)** Implement the constructor of `TwoLayerNetwork` with `torch.nn.Linear` and implement the `forward` function to compute $\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top \mathbf{x}_n)$.

```
### ========== TODO : START ========== ###
### part g: implement TwoLayerNetwork with torch.nn.Linear
self.twoLayerNetwork_1 = torch.nn.Linear(784, 400)
self.twoLayerNetwork_2 = torch.nn.Linear(400, 3)
### ========== TODO : END ========== ###
```

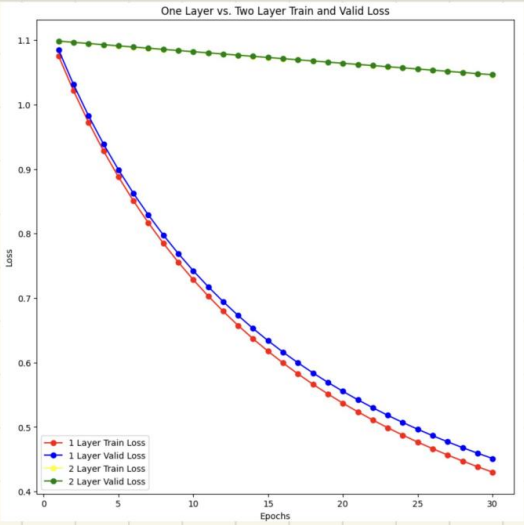
```
### ========== TODO : START ========== ###
### part g: implement the foward function
sig = torch.nn.Sigmoid()
layer_1 = self.twoLayerNetwork_1(x)
layer_1 = sig(layer_1)
outputs = self.twoLayerNetwork_2(layer_1)
### ========== TODO : END ========== ###
```

(h) **(2 pts)** Create an instance of `TwoLayerNetwork`, set up a criterion with `torch.nn.CrossEntropyLoss`, and set up a SGD optimizer with learning rate 0.0005 by using `torch.optim.SGD`. Then train `TwoLayerNetwork`.

```
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 |
```

Question assigned to the following page: 3.9

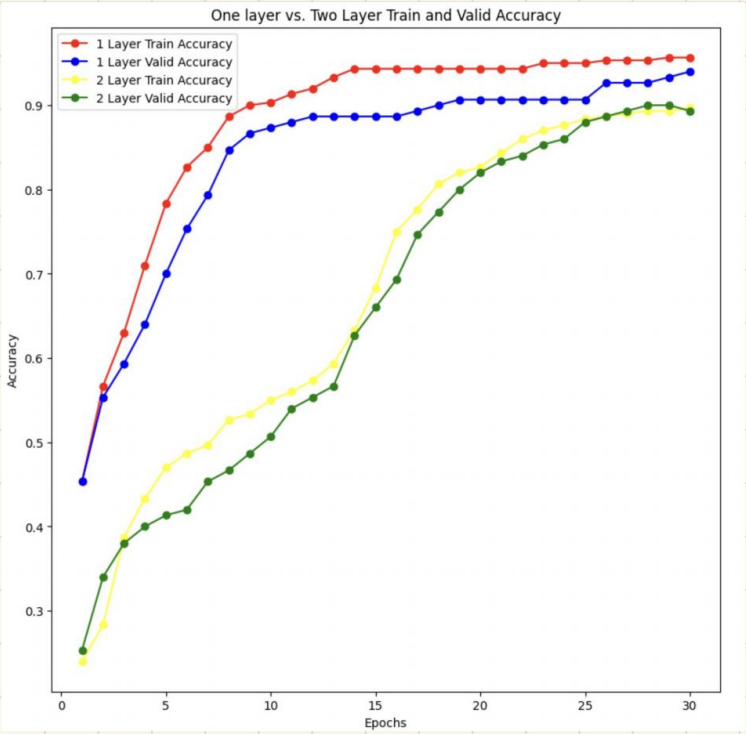
(i) **(3 pts)** Generate a plot depicting how `one_train_loss`, `one_valid_loss`, `two_train_loss`, `two_valid_loss` varies with epochs. Include the plot in the report and describe your findings.



The decline in loss for the training and validation set is notably slower in the 2 layer model compared to the 1 layer one. This is due to the error gradient being back-propagated through the model with every adjustment. There is also more parameters for the 2-layer network, Back propagation uses chain rule which can result in small derivatives which only minorly adjust weights. Additionally, SGD only uses a small number of samples so it takes longer for loss to decrease. 1 layer has fewer parameters which allows it to optimize easier and converge faster. Both 1 and 2 layer have similar training and validation loss which indicates good generalization and no over-fitting.

Question assigned to the following page: 3.10

(j) **(3 pts)** Generate a plot depicting how `one_train_acc`, `one_valid_acc`, `two_train_acc`, `two_valid_acc` varies with epochs. Include the plot in the report and describe your findings.



The increase in training and validation accuracies increase faster for the 1 layer model than the 2 layer model. However, both accuracies became fairly high at near 90% or above past 25 epochs. At 30 epochs the 1 layer model finished with a higher accuracy than the 2 layer model.

Question assigned to the following page: 3.11

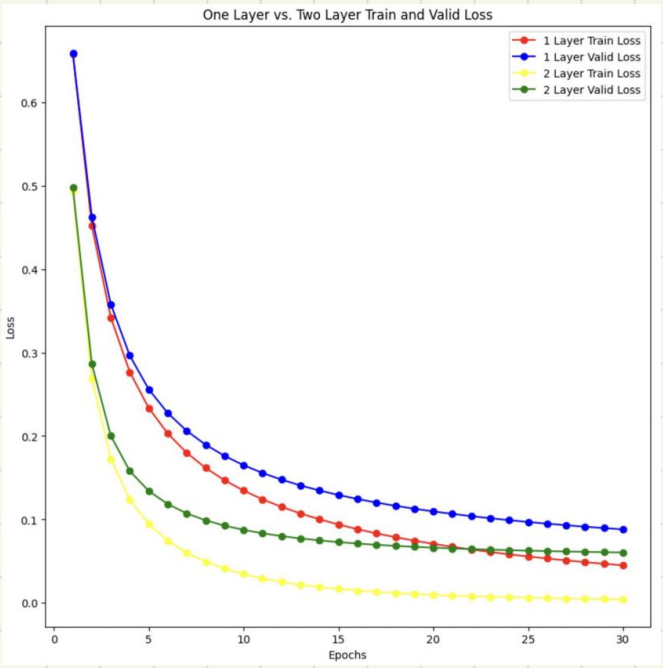
(k) **(3 pts)** Calculate and report the test accuracy of both the one-layer network and the two-layer network. How can you improve the performance of the two-layer network ?

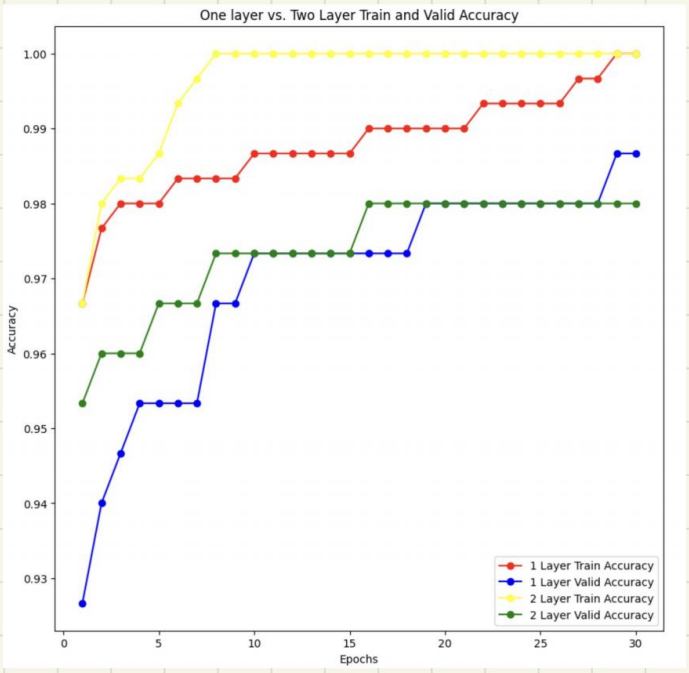
```
test accuracy of one layer network:  tensor(0.9733)
test accuracy of two layer network:  tensor(0.9667)
```

The accuracy of the single-layer network is higher than that of the two layer network. This could be due to the two layer network having more parameters which caused the training duration to extend beyond 30 epochs or overfitting. Therefore, adjusting these parameters could improve the performance of the two-layer network.

Question assigned to the following page: 3.12

(l) **(7 pts)** Replace the SGD optimizer with the Adam optimizer and do the experiments again. Show the loss figure, the accuracy figure, and the test accuracy. Include the figures in the report and describe your findings.






```
test accuracy of one layer network:  tensor(0.9667)
test accuracy of two layer network:  tensor(0.9667)
```

The train and validation loss converges faster with the Adam optimizer than the SGD optimizer. The loss for both two layer and one layer models is also much more close with the Adam optimizer.

The train and validation accuracy are much higher with the Adam optimizer than the SGD optimizer. 2 layer train accuracy is also higher in relation to the other accuracies and 1 layer validation accuracy is lower. All accuracies ended with higher than 97% accuracy after 30 epochs.

The test accuracies with Adam were both 0.9667, comparable to that of SGD.

Therefore, we can conclude that the Adam optimizer is better than SGD in this scenario.

Question assigned to the following page: 3.12

```
Start training OneLayerNetwork...
| epoch  1 | train loss 0.671099 | train acc 0.926667 | valid loss 0.678793 | valid acc 0.886667 |
| epoch  2 | train loss 0.461117 | train acc 0.970000 | valid loss 0.475818 | valid acc 0.926667 |
| epoch  3 | train loss 0.348042 | train acc 0.976667 | valid loss 0.366536 | valid acc 0.940000 |
| epoch  4 | train loss 0.280896 | train acc 0.980000 | valid loss 0.302011 | valid acc 0.940000 |
| epoch  5 | train loss 0.236744 | train acc 0.980000 | valid loss 0.259936 | valid acc 0.946667 |
| epoch  6 | train loss 0.205361 | train acc 0.980000 | valid loss 0.230302 | valid acc 0.953333 |
| epoch  7 | train loss 0.181743 | train acc 0.980000 | valid loss 0.208223 | valid acc 0.960000 |
| epoch  8 | train loss 0.163196 | train acc 0.983333 | valid loss 0.191070 | valid acc 0.960000 |
| epoch  9 | train loss 0.148154 | train acc 0.983333 | valid loss 0.177315 | valid acc 0.960000 |
| epoch 10 | train loss 0.135645 | train acc 0.986667 | valid loss 0.166008 | valid acc 0.960000 |
| epoch 11 | train loss 0.125036 | train acc 0.986667 | valid loss 0.156529 | valid acc 0.960000 |
| epoch 12 | train loss 0.115894 | train acc 0.986667 | valid loss 0.148453 | valid acc 0.966667 |
| epoch 13 | train loss 0.107913 | train acc 0.986667 | valid loss 0.141480 | valid acc 0.966667 |
| epoch 14 | train loss 0.100872 | train acc 0.986667 | valid loss 0.135391 | valid acc 0.966667 |
| epoch 15 | train loss 0.094605 | train acc 0.990000 | valid loss 0.130023 | valid acc 0.966667 |
| epoch 16 | train loss 0.088983 | train acc 0.990000 | valid loss 0.125251 | valid acc 0.973333 |
| epoch 17 | train loss 0.083909 | train acc 0.990000 | valid loss 0.120977 | valid acc 0.973333 |
| epoch 18 | train loss 0.079303 | train acc 0.990000 | valid loss 0.117124 | valid acc 0.973333 |
| epoch 19 | train loss 0.075101 | train acc 0.990000 | valid loss 0.113630 | valid acc 0.973333 |
| epoch 20 | train loss 0.071253 | train acc 0.990000 | valid loss 0.110446 | valid acc 0.973333 |
| epoch 21 | train loss 0.067715 | train acc 0.990000 | valid loss 0.107531 | valid acc 0.973333 |
| epoch 22 | train loss 0.064451 | train acc 0.993333 | valid loss 0.104849 | valid acc 0.973333 |
| epoch 23 | train loss 0.061431 | train acc 0.993333 | valid loss 0.102374 | valid acc 0.973333 |
| epoch 24 | train loss 0.058628 | train acc 0.993333 | valid loss 0.100081 | valid acc 0.973333 |
| epoch 25 | train loss 0.056022 | train acc 0.993333 | valid loss 0.097950 | valid acc 0.973333 |
| epoch 26 | train loss 0.053592 | train acc 0.996667 | valid loss 0.095964 | valid acc 0.973333 |
| epoch 27 | train loss 0.051322 | train acc 0.996667 | valid loss 0.094107 | valid acc 0.973333 |
| epoch 28 | train loss 0.049197 | train acc 0.996667 | valid loss 0.092367 | valid acc 0.973333 |
| epoch 29 | train loss 0.047204 | train acc 0.996667 | valid loss 0.090732 | valid acc 0.973333 |
| epoch 30 | train loss 0.045332 | train acc 1.000000 | valid loss 0.089194 | valid acc 0.980000 |
```

```
Start training TwoLayerNetwork...
| epoch  1 | train loss 0.499326 | train acc 0.960000 | valid loss 0.503736 | valid acc 0.940000 |
| epoch  2 | train loss 0.271541 | train acc 0.980000 | valid loss 0.289969 | valid acc 0.960000 |
| epoch  3 | train loss 0.174737 | train acc 0.983333 | valid loss 0.202948 | valid acc 0.960000 |
| epoch  4 | train loss 0.124827 | train acc 0.983333 | valid loss 0.159689 | valid acc 0.960000 |
| epoch  5 | train loss 0.095259 | train acc 0.986667 | valid loss 0.135297 | valid acc 0.966667 |
| epoch  6 | train loss 0.075258 | train acc 0.990000 | valid loss 0.119334 | valid acc 0.966667 |
| epoch  7 | train loss 0.060722 | train acc 0.996667 | valid loss 0.108014 | valid acc 0.966667 |
| epoch  8 | train loss 0.049782 | train acc 1.000000 | valid loss 0.099592 | valid acc 0.973333 |
| epoch  9 | train loss 0.041361 | train acc 1.000000 | valid loss 0.093108 | valid acc 0.973333 |
| epoch 10 | train loss 0.034772 | train acc 1.000000 | valid loss 0.087982 | valid acc 0.973333 |
| epoch 11 | train loss 0.029545 | train acc 1.000000 | valid loss 0.083841 | valid acc 0.973333 |
| epoch 12 | train loss 0.025349 | train acc 1.000000 | valid loss 0.080439 | valid acc 0.973333 |
| epoch 13 | train loss 0.021942 | train acc 1.000000 | valid loss 0.077606 | valid acc 0.973333 |
| epoch 14 | train loss 0.019148 | train acc 1.000000 | valid loss 0.075218 | valid acc 0.973333 |
| epoch 15 | train loss 0.016835 | train acc 1.000000 | valid loss 0.073187 | valid acc 0.980000 |
| epoch 16 | train loss 0.014902 | train acc 1.000000 | valid loss 0.071445 | valid acc 0.980000 |
| epoch 17 | train loss 0.013274 | train acc 1.000000 | valid loss 0.069939 | valid acc 0.980000 |
| epoch 18 | train loss 0.011891 | train acc 1.000000 | valid loss 0.068628 | valid acc 0.980000 |
| epoch 19 | train loss 0.010707 | train acc 1.000000 | valid loss 0.067479 | valid acc 0.980000 |
| epoch 20 | train loss 0.009688 | train acc 1.000000 | valid loss 0.066468 | valid acc 0.980000 |
| epoch 21 | train loss 0.008804 | train acc 1.000000 | valid loss 0.065572 | valid acc 0.980000 |
| epoch 22 | train loss 0.008034 | train acc 1.000000 | valid loss 0.064775 | valid acc 0.980000 |
| epoch 23 | train loss 0.007358 | train acc 1.000000 | valid loss 0.064063 | valid acc 0.980000 |
| epoch 24 | train loss 0.006762 | train acc 1.000000 | valid loss 0.063424 | valid acc 0.980000 |
| epoch 25 | train loss 0.006234 | train acc 1.000000 | valid loss 0.062849 | valid acc 0.980000 |
| epoch 26 | train loss 0.005765 | train acc 1.000000 | valid loss 0.062329 | valid acc 0.980000 |
| epoch 27 | train loss 0.005345 | train acc 1.000000 | valid loss 0.061858 | valid acc 0.980000 |
| epoch 28 | train loss 0.004969 | train acc 1.000000 | valid loss 0.061429 | valid acc 0.980000 |
| epoch 29 | train loss 0.004630 | train acc 1.000000 | valid loss 0.061039 | valid acc 0.980000 |
| epoch 30 | train loss 0.004324 | train acc 1.000000 | valid loss 0.060682 | valid acc 0.980000 |
```