CM146, Winter 2024
Problem Set 3: Deep learning, SVM, Kernels
Due March 1, 2024, 11:59pm PST

Sophia Sharif

Submission instructions

- Submit your solutions electronically on the course Gradescope site as PDF files.
- If you plan to typeset your solutions, please use the LaTeX solution template. If you must submit scanned handwritten solutions, please use a black pen on blank white paper and a high-quality scanner app.
- For questions involving math and derivation, please provide important intermediate steps and box the final answer clearly.

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.

Solution: Let $\phi(\boldsymbol{x})$ be a vector where the i-th entry is 1 if the i-th word in the English dictionary occurs in $\boldsymbol{x}$, and 0 otherwise. Then $k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^T \phi(\boldsymbol{z})$ is the number of unique words that occur in both $\boldsymbol{x}$ and $\boldsymbol{z}$. Since there exists a feature map $\phi(\cdot)$ such that $k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^T \phi(\boldsymbol{z})$ and $k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^T \phi(\boldsymbol{z}) = \phi(\boldsymbol{z})^T \phi(\boldsymbol{x}) = k(\boldsymbol{z}, \boldsymbol{x})$, the function $k(\boldsymbol{x}, \boldsymbol{z})$ is a kernel.

(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:

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

Solution: Let $k_1(\boldsymbol{x}, \boldsymbol{z}) = \frac{\boldsymbol{x}}{||\boldsymbol{x}||} \cdot \frac{\boldsymbol{z}}{||\boldsymbol{z}||}$ (which is a kernel by the scaling rule) and $k_2(\boldsymbol{x}, \boldsymbol{z}) = 1$ (which is a kernel for the feature vector $\phi(\boldsymbol{x}) = 1$). Then, $k_3(\boldsymbol{x}, \boldsymbol{z}) = k_1(\boldsymbol{x}, \boldsymbol{z}) + k_2(\boldsymbol{x}, \boldsymbol{z}) = \frac{\boldsymbol{x}}{||\boldsymbol{x}||} \cdot \frac{\boldsymbol{z}}{||\boldsymbol{z}||} + 1$ is a kernel by the sum rule. Finally, $k(\boldsymbol{x}, \boldsymbol{z}) = k_3(\boldsymbol{x}, \boldsymbol{z})^3 = \left(1 + \left(\frac{\boldsymbol{x}}{||\boldsymbol{x}||}\right) \cdot \left(\frac{\boldsymbol{z}}{||\boldsymbol{z}||}\right)\right)^3$ is a kernel by the product rule.

Parts of this assignment are adapted from course material by Tommi Jaakola (MIT), and Andrew Ng (Stanford), and Jenna Wiens (UMich).

(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?

Solution: The feature map $\phi_\beta(\cdot)$ is given by

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

for any value of $\beta > 0$. To see that this feature vector corresponds to the kernel $k_\beta(\boldsymbol{x}, \boldsymbol{z}) = (1 + \beta \boldsymbol{x} \cdot \boldsymbol{z})^3$, we can verify that $\phi_\beta(\boldsymbol{x})^T \phi_\beta(\boldsymbol{z}) = (1 + \beta \boldsymbol{x} \cdot \boldsymbol{z})^3$:

$$\begin{aligned}
\phi_\beta(\boldsymbol{x})^T \phi_\beta(\boldsymbol{z}) &= 1 + 3\beta x_1 z_1 + 3\beta x_2 z_2 + 3\beta^2 x_1^2 z_1^2 + 6\beta^2 x_1 z_1 x_2 z_2 + 3\beta^2 x_2^2 z_2^2 \\
&\quad + \beta^3 x_1^3 z_1^3 + 3\beta^3 x_1^2 z_1^2 x_2 z_2 + 3\beta^3 x_1 z_1 x_2^2 z_2^2 + \beta^3 x_2^3 z_2^3 \\
&= 1 + 3\beta(x_1 z_1 + x_2 z_2) + 3\beta^2(x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2) \\
&\quad + \beta^3(x_1^3 z_1^3 + 3x_1^2 z_1^2 x_2 z_2 + 3x_1 z_1 x_2^2 z_2^2 + x_2^3 z_2^3) \\
&= 1 + 3\beta(x_1 z_1 + x_2 z_2) + 3\beta^2(x_1 z_1 + x_2 z_2)^2 + \beta^3(x_1 z_1 + x_2 z_2)^3 \\
&= 1 + 3\beta \boldsymbol{x} \cdot \boldsymbol{z} + 3\beta^2(\boldsymbol{x} \cdot \boldsymbol{z})^2 + \beta^3(\boldsymbol{x} \cdot \boldsymbol{z})^3 \\
&= (1 + \beta \boldsymbol{x} \cdot \boldsymbol{z})^3
\end{aligned}$$

The parameter β scales the importance of the dot product $\boldsymbol{x} \cdot \boldsymbol{z}$ in the feature map. The alignment of the feature vectors in the feature space is more important when $\beta > 1$, and less important when $\beta < 1$. When $\beta = 1$, the feature map is the same as the feature map for the kernel $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^3$.

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?

Solution: Our only data point is $\boldsymbol{x} = (a, e)^T$ with label $y = -1$, so our constraint is $-\boldsymbol{\theta}^T \boldsymbol{x} = 1$ or $\theta_1 a + \theta_2 e = -1$. Maximizing the margin γ means maximizing $||\boldsymbol{x}|| \cos \alpha$ (the

[1] You may use any external program to expand the cubic.

distance between $\boldsymbol{x}$ and the plane), where α is the angle between $\boldsymbol{\theta}$ and $\boldsymbol{x}$. The magin is maximized when $\alpha = 0$, which means $\boldsymbol{\theta}^*$ is parallel to x. Thus, we know $\boldsymbol{\theta}^* = \lambda \boldsymbol{x} = \langle \lambda a, \lambda e \rangle$. Substituting $\theta_1 = \lambda a$ and $\theta_2 = \lambda e$ into the constraint, we get $-1 = \lambda a^2 + \lambda e^2$, so $\lambda = -\frac{1}{a^2+e^2}$. Thus, $\boldsymbol{\theta}^* = -\frac{1}{a^2+e^2}\langle a, e \rangle$.

(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?

Solution: Our two constraints are $\boldsymbol{\theta}^T \boldsymbol{x}_1 \geq 1$ and $-\boldsymbol{\theta}^T \boldsymbol{x}_2 \geq 1$, or $\theta_1 + \theta_2 \geq 1$ and $\theta_1 \leq -1$. Our objective function minimizes $||\boldsymbol{\theta}||$, so we choose $\theta_1 = -1$ to optimally satisfy $\theta_1 \leq -1$. Substituting $\theta_1 = -1$ into $\theta_1 + \theta_2 \geq 1$, we get $\theta_2 \geq 2$. Thus, $\boldsymbol{\theta}^* = \langle -1, 2 \rangle$. The margin is $\gamma = \frac{1}{||\boldsymbol{\theta}||} = \frac{1}{\sqrt{5}}$.

(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.

Solution: Our constraints are now $y_1(\boldsymbol{\theta}^T \boldsymbol{x}_1 + b) \geq 1$ and $y_2(\boldsymbol{\theta}^T \boldsymbol{x}_2 + b) \geq 1$, or $\theta_1 + \theta_2 + b \geq 1$ and $\theta_1 + b \leq -1$. Our objective function minimizes $||\boldsymbol{\theta}||$, so we choose $\theta_1 + b = -1$ to optimally satisfy $\theta_1 + b \leq -1$. Substituting this into $\theta_1 + \theta_2 + b \geq 1$, we get $\theta_2 \geq 2$, which implies that $\theta_2 = 2$. We again wish to minimize $||\boldsymbol{\theta}||$, so we choose $\theta_1 = 0$ to satisfy $\theta_1 + b = -1$. Thus, $\boldsymbol{\theta}^* = \langle 0, 2 \rangle$ and $b^* = -1$. The margin is $\gamma = \frac{1}{||\boldsymbol{\theta}||} = \frac{1}{2}$. Adding the offset parameter b allows the decision boundary to be shifted away from the origin, so the margin is larger.

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 ==========

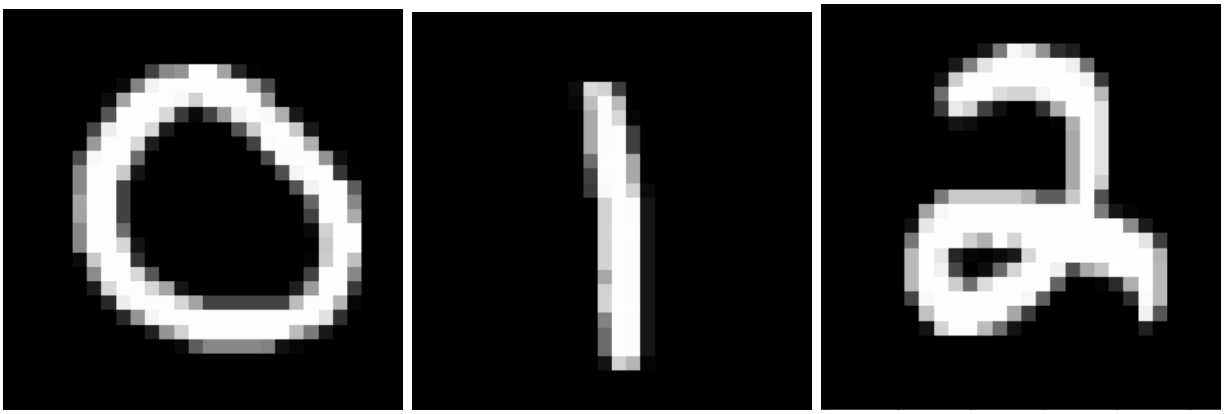
Note: For parts (b)-(h), you are expected to copy-paste your code as a part of the solution in the submission pdf. Tip: If you are using LaTeX, check out the Minted

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.

Solution:



(a) $y = 0$ (b) $y = 1$ (c) $y = 2$

Figure 1: Three training examples with different labels.

(b) **(3 pts)** The loaded examples are numpy arrays. Convert the numpy arrays to PyTorch tensors.

Solution:

```
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) **(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`.

Solution:

```
train_dataset = TensorDataset(X_train, y_train)
valid_dataset = TensorDataset(X_valid, y_valid)
test_dataset = TensorDataset(X_test, y_test)
```

```
train_loader = DataLoader(train_dataset, batch_size=10, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=10)
test_loader = DataLoader(test_dataset, batch_size=10)
```

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`.

Solution:

```
class OneLayerNetwork(torch.nn.Module):
def __init__(self):
    super(OneLayerNetwork, self).__init__()
    self.model = torch.nn.Linear(784, 3)

def forward(self, x):
    outputs = self.model(x)
    return outputs
```

(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`.

Solution:

```
model = OneLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.0005)
```

(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 ...
...
```

Solution:

```
for epoch in range(1, epochs):
  model.train()
  for batch_X, batch_y in train_loader:
    optimizer.zero_grad()
    output = model(batch_X)
    loss = criterion(output, batch_y)
    loss.backward()
    optimizer.step()
```

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 $\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top \mathbf{x}_n)$.

Solution:

```
class TwoLayerNetwork(torch.nn.Module):
    def __init__(self):
        super(TwoLayerNetwork, self).__init__()
        self.W1 = torch.nn.Linear(784, 400)
        self.sigma = torch.nn.Sigmoid()
        self.W2 = torch.nn.Linear(400, 3)

    def forward(self, x):
        layer1 = self.W1(x)
        activation = self.sigma(layer1)
        outputs = self.W2(activation)
        return outputs
```

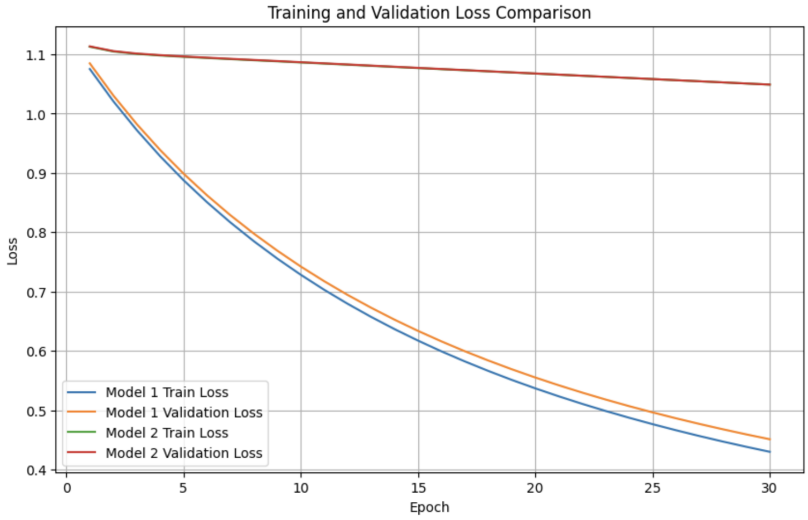
(h) **(2 pts)** Create an instance of `TwoLayerNetwork`, set up a criterion with `torch.nn.CrossEntropyLoss`, and set up a SGD optimizer with learning rate 0.0005 by using `torch.optim.SGD`. Then train `TwoLayerNetwork`.

Solution:

```
model_two = TwoLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_two.parameters(), lr=0.0005)
```

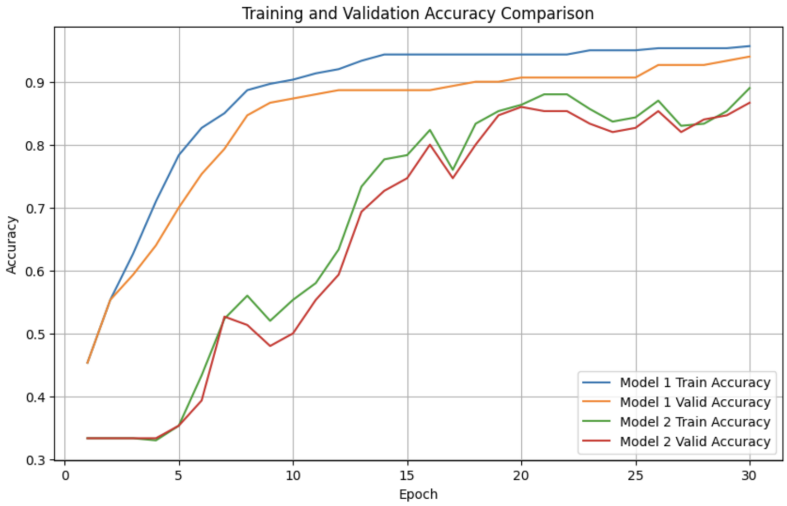
Performance Comparison [16 pts]

(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.



Solution: For the one-layer network, both the training and validation loss decrease as the number of epochs increases. The training loss is slightly higher than the validation loss, which is expected. For the two-layer network, the decrease in both training and validation loss is much more gradual.

(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.



Solution: As the number of epochs increases, the training accuracy of both the one-layer and two-layer networks generally increases. The validation accuracy is slightly higher than the training accuracy, which is expected. The one-layer network performs significantly better than the two-layer network.

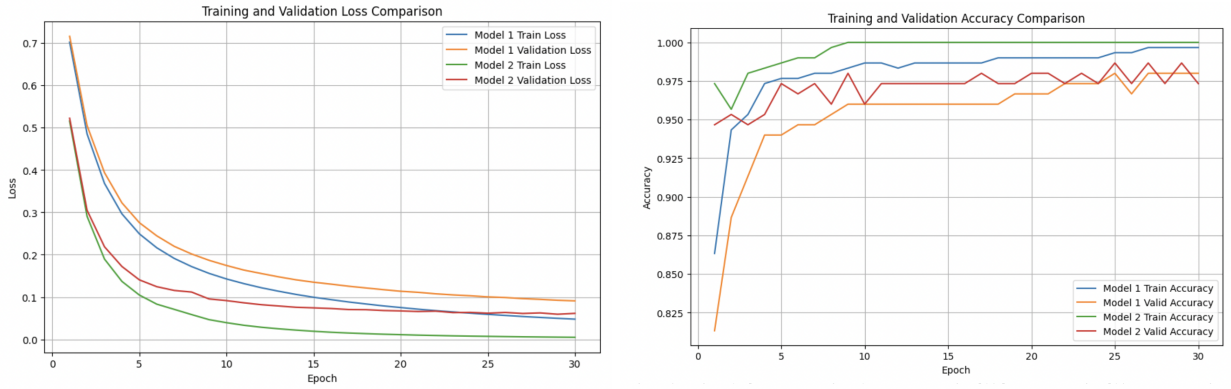
(k) **(3 pts)** Calculate and report the test accuracy of both the one-layer network and the two-layer network. How can you improve the performance of the two-layer network ?

Solution: The accuracy of the one-layer network is 96.00%, while the accuracy for the two-layer network is 89.33%. The two-layer network likely performs worse because of the sigmoid activation function causing the vanishing gradient problem. The sigmoid squashes values into a small range leading to small gradients, which can cause the network to learn slowly. We can improve the performance of the two-layer network by using a different activation function like the ReLU.

(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.

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

Solution: As the number of epochs increases, the training and validation loss for both the one-layer and two-layer networks decrease. The training accuracy of both networks generally increases as the number of epochs increases. The training accuracy is slightly higher than the validation accuracy, which is expected. The accuracy of the one-layer network is 96.67%, while the accuracy for the two-layer network is 96.00%. The loss for both models decreases faster when using the Adam optimizer. The two-layer network performs better than the one-layer network, likely because the Adam optimizer is better at handling the vanishing gradient problem.



(a) Loss vs. Epochs

(b) Accuracy vs. Epochs

Figure 2: Loss and accuracy for the one-layer and two-layer networks using the Adam optimizer.