Problem Set 3: Deep learning, Learning theory, Kernels

Bonnie Liu
UID: 005300989
Discussion 1A (Ulzee An)

Due Date: February 28, 2022

For this problem set, I collaborated with Henry Li, Hannah Zhong, David Xiong, Justin He, Yunqiu (Rachel) Han, and Nicholas Dean.

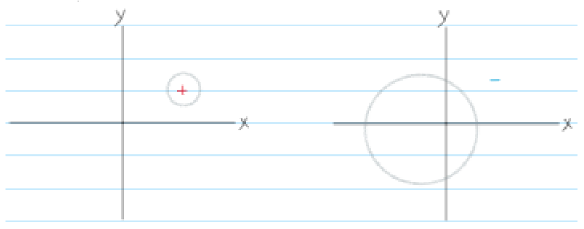
1 VC-Dimension

1.1 VC-dimension of circles in the x, y plane

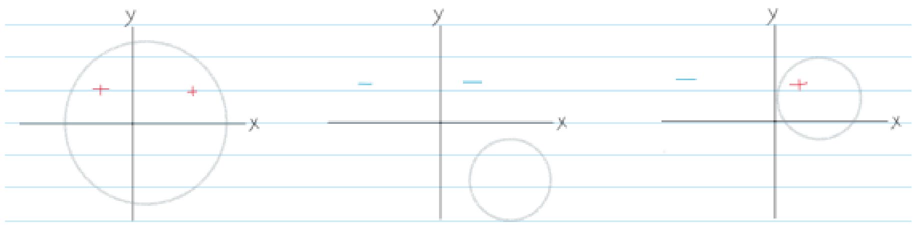
The VC-dimension of hypothesis space H over instance space X is the size of the largest finite subset of X that is shattered by H. A set S of examples is shattered by a set of functions H if for every partition of the examples in S into positive and negative examples, there is a function in H that gives exactly these labels to the examples.

Here, X is all points in the x, y plane, and H_c is circles in the x, y plane with points inside circle classified as positive examples. Let's try to find the VC-dimension of H_c.

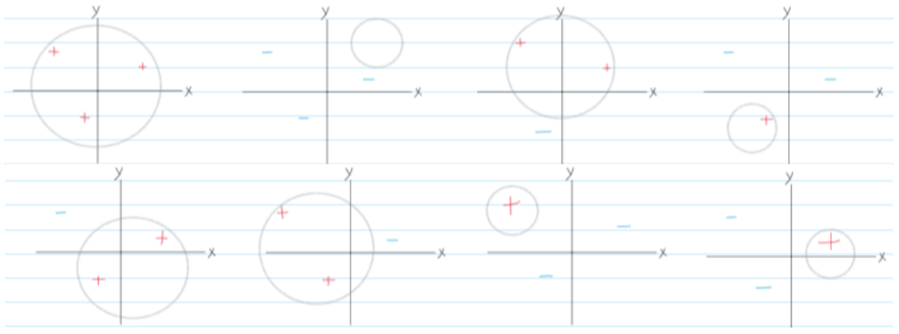
Below are all possible labelings of 1 point:



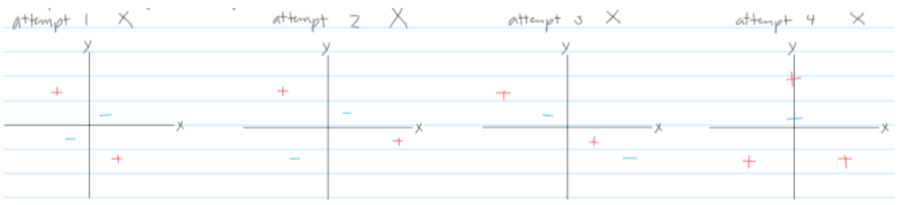
Below are all possible labelings of 2 points:



Below are all possible labelings of 3 points:



Below are our attempts of labeling 4 points:



Therefore, $VC(H_c) = 3$.

1.2 Effect of $|H|$ increasing on VC-dimension

1.2.1 Hypothesis subsets

Let two hypothesis classes H_1 and H_2 satisfy $H_1 \subseteq H_2$. H_1 is a subset of H_2, so $|H_1| \leq |H_2|$. Everything hypothesis H_1 has, H_2 has as well. It follows then that every finite subset of instance space X that can be shattered by H_1 can be also be shattered by H_2. Therefore, $VC(H_1) \leq VC(H_2)$.

1.2.2 Hypothesis unions

Let $H_1 = H_2 \cup H_3$. The statement $VC(H_1) \leq VC(H_2) + VC(H_3)$ is false. The following is a counterexample: Suppose H_2 is a set containing the hypothesis that only predicts 0 and H_3 is a set containing the hypothesis that only predicts 1. Then $VC(H_2) = VC(H_3) = 0$ since neither of these can shatter a single point. It follows that $VC(H_2) + VC(H_3) = 0$. Since $H_1 = H_2 \cup H_3$, H_1 consists of the hypothesis that predicts 0 and the hypothesis that predicts 1. This is sufficient to shatter one point, so $VC(H_1) = 1 > VC(H_2) + VC(H_3) = 0$.

2 Kernels

2.1 Unique words

A kernel function k is a bivariate function that satisfies the following two properties: $k(\boldsymbol{x_m}, \boldsymbol{x_n}) = k(\boldsymbol{x_n}, \boldsymbol{x_m})$ and $k(\boldsymbol{x_m}, \boldsymbol{x_n}) = \phi(\boldsymbol{x_m})^T \phi(\boldsymbol{x_n})$ for some function ϕ.

Since $k(\boldsymbol{x}, \boldsymbol{z})$ is defined to equal the number of unique words that occur in both $\boldsymbol{x}$ and $\boldsymbol{z}$ given that $\boldsymbol{x}$ and $\boldsymbol{z}$ are two documents, k satisfies the first property because intersection is commutative.

For the second property, we need to find ϕ such that $k(\boldsymbol{x_m}, \boldsymbol{x_n}) = \phi(\boldsymbol{x_m})^T \phi(\boldsymbol{x_n})$. Let ϕ be the function that takes in a document $\boldsymbol{x}$ and returns a vector that corresponds to all the possible words. The ith element of $\phi(\boldsymbol{x})$ is 1 if the word corresponding to position i is present in $\boldsymbol{x}$ and 0 otherwise. Then $\phi(\boldsymbol{x_m})^T \phi(\boldsymbol{x_n})$ will give us the number of unique words that appear in both $\boldsymbol{x_m}$ and $\boldsymbol{x_n}$.

Because k satisfies both properties, this function is a kernel.

2.2 Construction rules

$(1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}))^3$ can be written as the product of three terms, so we can use the product construction rule two times.

$$\begin{aligned}(1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}))^3 &= (1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}))(1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}))^2 \\ &= (1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}))(1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}))(1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}))\end{aligned}$$

Now we want to show that $1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||})$ is a kernel. If we show that 1 and $(\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||})$ are both kernels, we can use the sum construction rule. Let's start with 1.

If $k(\boldsymbol{x_m}, \boldsymbol{x_n}) = 1$, then $k(\boldsymbol{x_m}, \boldsymbol{x_n}) = k(\boldsymbol{x_n}, \boldsymbol{x_m}) = 1$. Let ϕ be the function that maps its input to 1 regardless of what the input is. Then $k(\boldsymbol{x_m}, \boldsymbol{x_n}) = \phi(\boldsymbol{x_m})^T \phi(\boldsymbol{x_n}) = 1 \cdot 1 = 1$. Thus, $k(\boldsymbol{x_m}, \boldsymbol{x_n}) = 1$ is a kernel.

Now let's show that $(\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||})$ is a kernel as well.

$$\begin{aligned}(\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}) &= \frac{\boldsymbol{x} \cdot \boldsymbol{z}}{||\boldsymbol{x}||||\boldsymbol{z}||} \\ &= \frac{1}{||\boldsymbol{x}||||\boldsymbol{z}||}(\boldsymbol{x} \cdot \boldsymbol{z}) \\ &= \frac{1}{||\boldsymbol{x}||}(\boldsymbol{x} \cdot \boldsymbol{z})\frac{1}{||\boldsymbol{z}||}\end{aligned}$$

Since we know that $\boldsymbol{x} \cdot \boldsymbol{z}$ is a kernel, we can use the scaling construction rule since taking the magnitude of a vector is our function $f(\boldsymbol{x})$ such that $f(\boldsymbol{x}) \in \mathbb{R}$. Thus, it follows that $(1 + (\frac{\boldsymbol{x}}{||\boldsymbol{x}||}) \cdot (\frac{\boldsymbol{z}}{||\boldsymbol{z}||}))^3$ is a kernel.

2.3 Finding feature map given kernel

Since $\boldsymbol{x}, \boldsymbol{z} \in \mathbb{R}^2$, $(1 + \beta \boldsymbol{x} \cdot \boldsymbol{z})^3 = 1 + 3\beta x_1 z_1 + 3\beta^2 x_1^2 z_1^2 + \beta^3 x_1^3 z_1^3 + 3\beta x_2 z_2 + 3\beta^2 x_2^2 z_2^2 + \beta^3 x_2^3 z_2^3 + 6\beta^2 x_1 x_2 + 3\beta^3 x_1^2 x_2 + 3\beta^3 x_1 x_2^2$. Then

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

The similarities to the kernel $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^3$ is that both of their feature maps have 10 features, and if we remove the β terms from the feature map we found, we get the feature map of $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^3$. The differences lie in the β terms of our feature map. Thus, the parameter β plays the role of changing the magnitude of the kernel function.

3 SVM

3.1 Single training vector

Following the advice of Campuswire post 345, I started by examining the case where $\boldsymbol{x} = (1\ 1)^T$. Then $\boldsymbol{\theta} = (-3\ 1)^T$ satisfies the constraints since $y_n \boldsymbol{\theta}^T \boldsymbol{x_n} = 2 \geq 1$, and $\frac{1}{2}||\boldsymbol{\theta}||^2 = \frac{\sqrt{10}}{2}$. Another option for $\boldsymbol{\theta}$ is $\boldsymbol{\theta} = (-2\ 1)^T$. If $\boldsymbol{\theta} = (-2\ 1)^T$, $y_n \boldsymbol{\theta}^T \boldsymbol{x_n} = 1 \geq 1$, and $\frac{1}{2}||\boldsymbol{\theta}||^2 = \frac{\sqrt{5}}{2}$.

Now let's examine the case where $\boldsymbol{x} = (2\ 2)^T$. Then $\boldsymbol{\theta} = (-2\ 1)^T$ satisfies the constraints since $y_n \boldsymbol{\theta}^T \boldsymbol{x_n} = 2 \geq 1$ and $\frac{1}{2}||\boldsymbol{\theta}||^2 = \frac{\sqrt{5}}{2}$. Another option for $\boldsymbol{\theta}$ is $\boldsymbol{\theta} = (-1\ 0.5)^T$. If $\boldsymbol{\theta} = (-1\ 0.5)^T$, $y_n \boldsymbol{\theta}^T \boldsymbol{x_n} = 1 \geq 1$, and $\frac{1}{2}||\boldsymbol{\theta}||^2 = \frac{\sqrt{5}}{4}$.

Based on these two examples, we can see that the optimization function $\frac{1}{2}||\boldsymbol{\theta}||^2$ is minimized when our constraint $y_n \boldsymbol{\theta}^T \boldsymbol{x_n}$ equals 1.

To solve for $\boldsymbol{\theta}^*$, I used Lagrange multipliers. Let $f(\theta_1, \theta_2) = \frac{1}{2}(\theta_1^2 + \theta_2^2)$ be our objective function, which we want to minimize. Let $g(\theta_1, \theta_2) = -a\theta_1 - e\theta_2 = 1$ be our constraint. Then

$$\nabla g(\theta_1, \theta_2) = \begin{pmatrix} -a \\ -e \end{pmatrix}$$

and

$$\nabla f(\theta_1, \theta_2) = \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix}$$

Using Lagrange multipliers, we know that $\nabla f(\theta_1, \theta_2) = \lambda \nabla g(\theta_1, \theta_2)$. Plugging this in, we get the following:

$$\begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} = \lambda \begin{pmatrix} -a \\ -e \end{pmatrix}$$

This means $\theta_1 = -\lambda a$ and $\theta_2 = -\lambda e$. Our third equation comes from our constraint: $-a\theta_1 - e\theta_2 = 1$. Solving this system of linear equations gives us the following:

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

3.2 Two training examples

Our approach for this problem is similar to the one in the problem above, except this time, we have two constraints instead of one. Our two constraints are $g(\theta_1, \theta_2) = \theta_1 + \theta_2 = 1$ and $h(\theta_1, \theta_2) = -\theta = 1$.

$$\nabla g(\theta_1, \theta_2) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\nabla h(\theta_1, \theta_2) = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

Using Lagrange multipliers, we know that $\nabla f(\theta_1, \theta_2) = \lambda \nabla g(\theta_1, \theta_2) + \mu \nabla h(\theta_1, \theta_2)$. Plugging this in, we get the following:

$$\begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} = \lambda \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \mu \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

This means $\theta_1 = \lambda - \mu$ and $\theta_2 = -\lambda$. Our third and fourth equations come from our constraint: $\theta_1 + e\theta_2 = 1$ and $-\theta_1 = 1$. Solving this system of linear equations gives us the following:

$$\boldsymbol{\theta}^* = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$

The margin γ is $\frac{1}{\sqrt{5}}$ because the margin given a weight vector is the minimum distance from our decision boundary defined by our weight vector to any data point $(\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), ..., (\boldsymbol{x_n}, y_n)$. Since we only have two data points, we can calculate the distances between them and the decision boundary fairly easily.

$$\begin{aligned}\frac{y_1(\boldsymbol{\theta}^T\boldsymbol{x_1})}{||\boldsymbol{\theta}||_2} &= \frac{1(-1\ 2)(1\ 1)^T}{\sqrt{1^2+2^2}}\\ &= \frac{1}{\sqrt{5}}\end{aligned}$$

$$\begin{aligned}\frac{y_2(\boldsymbol{\theta}^T\boldsymbol{x_2})}{||\boldsymbol{\theta}||_2} &= \frac{-1(-1\ 2)(1\ 0)^T}{\sqrt{1^2+2^2}}\\ &= \frac{1}{\sqrt{5}}\end{aligned}$$

As expected, the distance between each point and the decision boundary is the same since we only have two points.

3.3 Offset parameter b

Our approach for this problem is similar to the one in the problem above, except this time, we have three inputs to our functions instead of two. Our objective function is $\nabla f(\theta_1, \theta_2, b) = \frac{1}{2}(\theta_1^2 + \theta_2^2)$, and our two constraints are $g(\theta_1, \theta_2, b) = \theta_1 + \theta_2 + b = 1$ and $h(\theta_1, \theta_2, b) = \theta_1 + b = -1$.

$$\nabla g(\theta_1, \theta_2, b) = \begin{pmatrix}1\\1\\1\end{pmatrix}$$

$$\nabla h(\theta_1, \theta_2, b) = \begin{pmatrix}1\\0\\1\end{pmatrix}$$

Using Lagrange multipliers, we know that $\nabla f(\theta_1, \theta_2, b) = \lambda \nabla g(\theta_1, \theta_2, b) + \mu \nabla h(\theta_1, \theta_2, b)$. Plugging this in, we get the following:

$$\begin{pmatrix}\theta_1\\\theta_2\\0\end{pmatrix} = \lambda\begin{pmatrix}1\\1\\1\end{pmatrix} + \mu\begin{pmatrix}1\\0\\1\end{pmatrix}$$

This means that $\theta_1 = \lambda + \mu$, $\theta_2 = \lambda$, and $0 = \lambda + \mu$. Our fourth and fifth equations come from our constraints: $\theta_1 + \theta_2 + b = 1$ and $\theta_1 + b = -1$. Solving this system of linear equations gives us the following:

$$\boldsymbol{\theta}^* = \begin{pmatrix}0\\2\end{pmatrix}$$

and

$$b^* = -1$$

The margin γ is $\frac{1}{2}$ because

$$\begin{aligned}\frac{y_1(\boldsymbol{\theta}^T \boldsymbol{x_1} + b)}{||\boldsymbol{\theta}||_2} &= \frac{1[(0\ 2)(1\ 1)^T - 1]}{\sqrt{0^2+2^2}} \\ &= \frac{1}{2}\end{aligned}$$

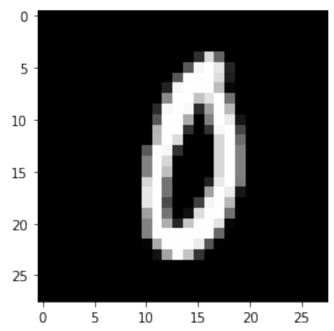
$$\begin{aligned}\frac{y_2(\boldsymbol{\theta}^T \boldsymbol{x_2} + b)}{||\boldsymbol{\theta}||_2} &= \frac{-1[(0\ 2)(1\ 0)^T - 1]}{\sqrt{0^2+2^2}} \\ &= \frac{1}{2}\end{aligned}$$

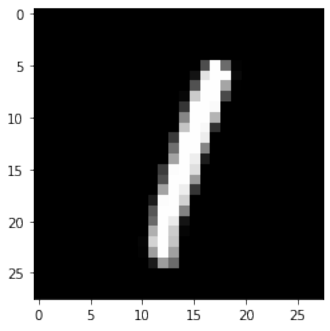
With offset, our margin is bigger than without offset. With offset, our classifier dictates a decision boundary that is the horizontal line $y = \frac{1}{2}$. Previously, without offset, our classifier dictated a decision boundary that had the equation $y = \frac{1}{2}x$.

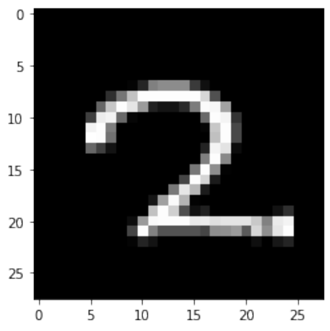
4 Implementation: Digit Recognizer

4.1 Data Visualization and Preparation: plot_img function

```
### part a: print out three training images with different labels
idx_0 = np.random.choice([i for i in range(X_train.shape[0]) if y_train[i] == 0])
idx_1 = np.random.choice([i for i in range(X_train.shape[0]) if y_train[i] == 1])
idx_2 = np.random.choice([i for i in range(X_train.shape[0]) if y_train[i] == 2])
plot_img(X_train[idx_0])
plot_img(X_train[idx_1])
plot_img(X_train[idx_2])
```







4.2 Data Visualization and Preparation: convert numpy arrays to tensors

```
### 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)
```

4.3 Data Visualization and Preparation: prepare dataloaders

```
### 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([(X_train[i], y_train[i]) \
                           for i in range(X_train.shape[0])], batch_size=10)
valid_loader = DataLoader([(X_valid[i], y_valid[i]) \
                           for i in range(X_valid.shape[0])], batch_size=10)
test_loader = DataLoader([(X_test[i], y_test[i]) \
                          for i in range(X_test.shape[0])], batch_size=10)
```

4.4 One-Layer Network: constructor and forward function

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

        ### ========== TODO : START ========== ###
        ### part d: implement OneLayerNetwork with torch.nn.Linear
        self.linear = 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.linear.forward(x)
        ### ========== TODO : END ========== ###
        return outputs
```

4.5 One-Layer Network: instantiation, criterion, and SGD optimizer

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

4.6 One-Layer Network: training process

```
### 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()
```

Using the above code, I obtained values very close to the ones listed in the spec.

4.7 Two-Layer Network: constructor and forward function

```
class TwoLayerNetwork(torch.nn.Module):
    def __init__(self):
        super(TwoLayerNetwork, self).__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 ========== ###

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

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

4.8 Two-Layer Network: instantiation, criterion, SGD optimizer, and training

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

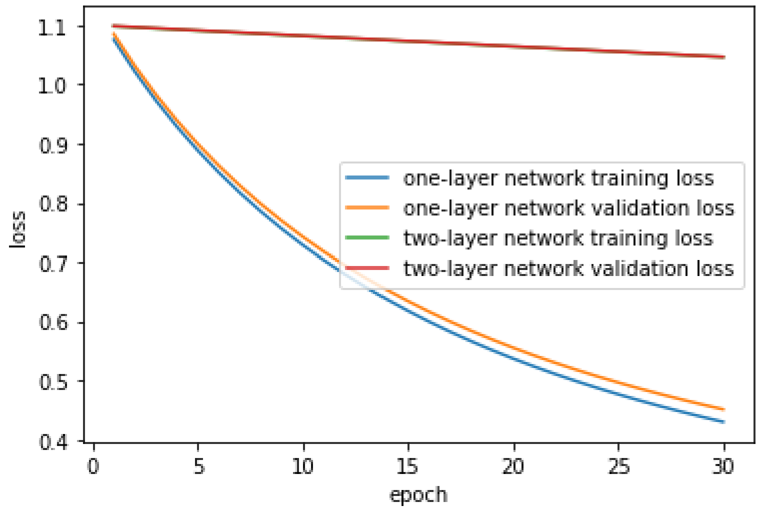
Using the code above, I obtained the following from training the two-layer network.

```
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!
```

4.9 Performance Comparison: loss vs. epoch

```
### part i: generate a plot to compare one_train_loss, one_valid_loss,
###         two_train_loss, two_valid_loss
plt.figure()
plt.plot(range(1, 31), one_train_loss, label="one-layer network training loss")
plt.plot(range(1, 31), one_valid_loss, label="one-layer network validation loss")
plt.plot(range(1, 31), two_train_loss, label="two-layer network training loss")
plt.plot(range(1, 31), two_valid_loss, label="two-layer network validation loss")
plt.legend()
plt.xlabel("epoch")
plt.ylabel("loss")
```

The above code results in the following graph:

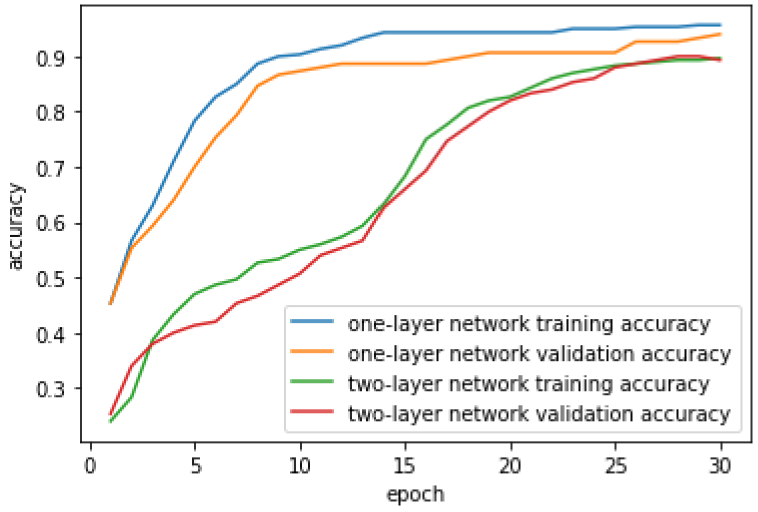


As we can see in the graph above, the loss decreased with each iteration. The one-layer network training loss was less than its validation loss. The two-layer network training loss and validation loss only decreased by about 0.5 each, suggesting that our model is running into the vanishing gradient problem since it's not converging to a solution fast enough.

4.10 Performance Comparison: accuracy vs. epoch

```
### part j: generate a plot to compare one_train_acc, one_valid_acc,
###         two_train_acc, two_valid_acc
plt.figure()
plt.plot(range(1, 31), one_train_acc, label="one-layer network training accuracy")
plt.plot(range(1, 31), one_valid_acc, label="one-layer network validation accuracy")
plt.plot(range(1, 31), two_train_acc, label="two-layer network training accuracy")
plt.plot(range(1, 31), two_valid_acc, label="two-layer network validation accuracy")
plt.legend()
plt.xlabel("epoch")
plt.ylabel("accuracy")
```

The above code results in the following graph:



As we can see in the graph above, the training accuracies of both the one-layer network and the two-layer network are generally higher than that of their validation counterparts. This makes sense since we trained our neural networks on the training data. The one-layer network performed better than the two-layer network in terms of accuracy. This means that increasing the number of layers in our neural networks does not automatically mean we get a higher-performing neural network.

4.11 Performance Comparison: test accuracy

```
### part k: calculate the test accuracy
one_test_acc = evaluate_acc(model_one, test_loader)
two_test_acc = evaluate_acc(model_two, test_loader)
print("One-Layer Network Test Accuracy: " + str(one_test_acc))
print("Two-Layer Network Test Accuracy: " + str(two_test_acc))
```

The above code resulted in the following results:

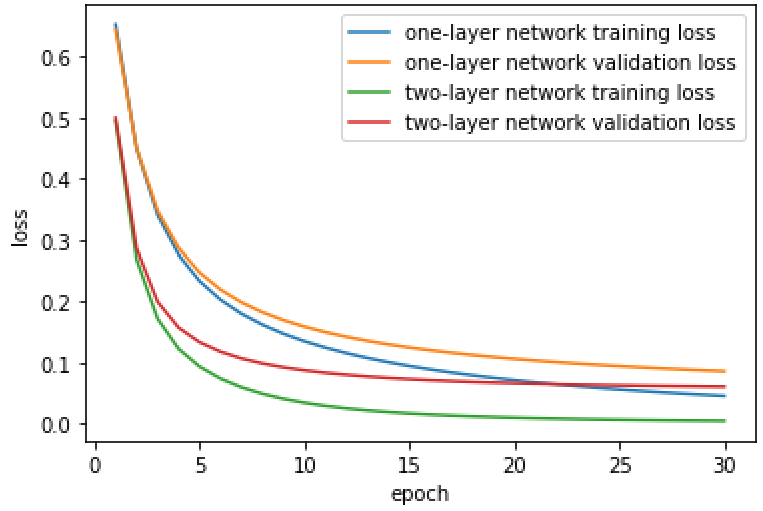
```
One-Layer Network Test Accuracy: tensor(0.9600)
Two-Layer Network Test Accuracy: tensor(0.9000)
```

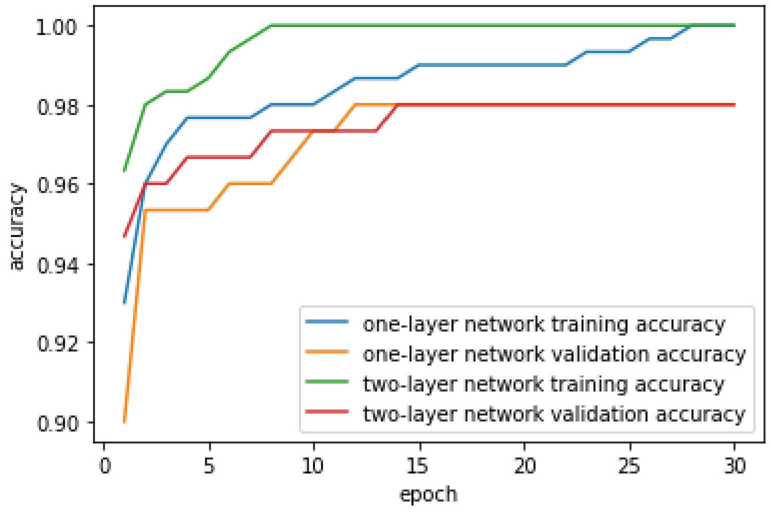
We can improve the performance of our two-layer network by changing the activation function from sigmoid to something that suffers a little less from the vanishing gradient problem, such as the ReLU function. Something else we can do to bypass the vanishing gradient issue is initializing our weights differently. We can also try increasing the number of neurons in the hidden layer. Another thing we can try is increasing our learning rate so that our model will converge to a solution faster. Right now, our two-layer network doesn't seem to have converged yet.

4.12 Performance Comparison: Adam optimizer

```
### part l: replace the SGD optimizer with the Adam optimizer and do the experiments again
print("\n=======================================")
print("Running experiments with Adam optimizer")
print("=======================================\n")

model_one = OneLayerNetwork()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_one.parameters(), lr=0.0005)

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

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

print("Start training TwoLayerNetwork...")
results_two = train(model_two, criterion, optimizer, train_loader, valid_loader)
print("Done!")

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(range(1, 31), one_train_loss, label="one-layer network training loss")
plt.plot(range(1, 31), one_valid_loss, label="one-layer network validation loss")
plt.plot(range(1, 31), two_train_loss, label="two-layer network training loss")
plt.plot(range(1, 31), two_valid_loss, label="two-layer network validation loss")
plt.legend()
plt.xlabel("epoch")
plt.ylabel("loss")

plt.figure()
plt.plot(range(1, 31), one_train_acc, label="one-layer network training accuracy")
plt.plot(range(1, 31), one_valid_acc, label="one-layer network validation accuracy")
plt.plot(range(1, 31), two_train_acc, label="two-layer network training accuracy")
plt.plot(range(1, 31), two_valid_acc, label="two-layer network validation accuracy")
plt.legend()
plt.xlabel("epoch")
plt.ylabel("accuracy")

one_test_acc = evaluate_acc(model_one, test_loader)
two_test_acc = evaluate_acc(model_two, test_loader)
print("One-Layer Network Test Accuracy: " + str(one_test_acc))
print("Two-Layer Network Test Accuracy: " + str(two_test_acc))
```

The above code resulted in the following:





```
One-Layer Network Test Accuracy: tensor(0.9733)
Two-Layer Network Test Accuracy: tensor(0.9667)
```

As seen from our results above, test accuracy for both networks improved when we used the Adam optimizer instead of the SGD optimizer, but the one-layer network still performed slightly better than the two-layer network. As expected, the training accuracies of both networks were greater than their respective validation accuracies. When we used the Adam optimizer, the training loss curve and the validation loss curve for the two-layer network more closely resembled a convex shape rather than the previous linear shape we saw when we used the SGD optimizer.

5 Code

```python
import os
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch.utils.data import TensorDataset, DataLoader
from PIL import Image

# To add your own Drive Run this cell.
from google.colab import drive
drive.mount('/content/drive')

######################################################################
# OneLayerNetwork
######################################################################

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

        ### ========== TODO : START ========== ###
        ### part d: implement OneLayerNetwork with torch.nn.Linear
        self.linear = 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.linear.forward(x)
        ### ========== TODO : END ========== ###
        return outputs

######################################################################
```

```
# TwoLayerNetwork
######################################################################

class TwoLayerNetwork(torch.nn.Module):
    def __init__(self):
        super(TwoLayerNetwork, self).__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 ========== ###

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

        ### ========== TODO : START ========== ###
        ### part g: implement the foward function
        transform = torch.nn.Sigmoid()
        hidden = transform(self.linear1.forward(x))
        outputs = self.linear2.forward(hidden)
        ### ========== TODO : END ========== ###
        return outputs

# load data from csv
# X.shape = (n_examples, n_features), y.shape = (n_examples, )
def load_data(filename):
    data = np.loadtxt(filename)
    y = data[:, 0].astype(int)
    X = data[:, 1:].astype(np.float32) / 255
    return X, y

# plot one example
# x.shape = (features, )
def plot_img(x):
    x = x.reshape(28, 28)
    img = Image.fromarray(x*255)
    plt.figure()
    plt.imshow(img)
    return

def evaluate_loss(model, criterion, dataloader):
    model.eval()
    total_loss = 0.0
    for batch_X, batch_y in dataloader:
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
```

```python
        total_loss += loss.item()

    return total_loss / len(dataloader)

def evaluate_acc(model, dataloader):
    model.eval()
    total_acc = 0.0
    for batch_X, batch_y in dataloader:
        outputs = model(batch_X)
        predictions = torch.argmax(outputs, dim=1)
        total_acc += (predictions==batch_y).sum()

    return total_acc / len(dataloader.dataset)

def train(model, criterion, optimizer, train_loader, valid_loader):
    train_loss_list = []
    valid_loss_list = []
    train_acc_list = []
    valid_acc_list = []
    for epoch in range(1, 31):
        model.train()
        for batch_X, batch_y in train_loader:
            ### ========== 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 ========== ###

        train_loss = evaluate_loss(model, criterion, train_loader)
        valid_loss = evaluate_loss(model, criterion, valid_loader)
        train_acc = evaluate_acc(model, train_loader)
        valid_acc = evaluate_acc(model, valid_loader)
        train_loss_list.append(train_loss)
        valid_loss_list.append(valid_loss)
        train_acc_list.append(train_acc)
        valid_acc_list.append(valid_acc)

        print(f"| epoch {epoch:2d} | train loss {train_loss:.6f} | train acc {train_acc:.6f

    return train_loss_list, valid_loss_list, train_acc_list, valid_acc_list

from re import X
```

```
######################################################################
# main
######################################################################

def main():

    # fix random seed
    np.random.seed(0)
    torch.manual_seed(0)

    # load data with correct file path

    ### ========== TODO : START ========== ###
    data_directory_path =  "/content/drive/My Drive/CS M146"
    ### ========== TODO : END ========== ###

    # X.shape = (n_examples, n_features)
    # y.shape = (n_examples, )
    X_train, y_train = load_data(os.path.join(data_directory_path, "ps3_train.csv"))
    X_valid, y_valid = load_data(os.path.join(data_directory_path, "ps3_valid.csv"))
    X_test, y_test = load_data(os.path.join(data_directory_path, "ps3_test.csv"))

    ### ========== TODO : START ========== ###
    ### part a: print out three training images with different labels
    idx_0 = np.random.choice([i for i in range(X_train.shape[0]) if y_train[i] == 0])
    idx_1 = np.random.choice([i for i in range(X_train.shape[0]) if y_train[i] == 1])
    idx_2 = np.random.choice([i for i in range(X_train.shape[0]) if y_train[i] == 2])
    plot_img(X_train[idx_0])
    plot_img(X_train[idx_1])
    plot_img(X_train[idx_2])

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

    print("Data preparation...")

    ### ========== 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 ========== ###

    ### ========== 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([(X_train[i], y_train[i]) \
                               for i in range(X_train.shape[0])], batch_size=10)
    valid_loader = DataLoader([(X_valid[i], y_valid[i]) \
                               for i in range(X_valid.shape[0])], batch_size=10)
    test_loader = DataLoader([(X_test[i], y_test[i]) \
                              for i in range(X_test.shape[0])], batch_size=10)

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

    ### ========== 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 ========== ###

    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=0.0005)

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

    print("Start training TwoLayerNetwork...")
    results_two = train(model_two, criterion, optimizer, train_loader, valid_loader)
    print("Done!")

    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

    ### ========== TODO : START ========== ###
    ### part i: generate a plot to compare one_train_loss, one_valid_loss, two_train_loss,
    plt.figure()
    plt.plot(range(1, 31), one_train_loss, label="one-layer network training loss")
    plt.plot(range(1, 31), one_valid_loss, label="one-layer network validation loss")
    plt.plot(range(1, 31), two_train_loss, label="two-layer network training loss")
    plt.plot(range(1, 31), two_valid_loss, label="two-layer network validation loss")
    plt.legend()
    plt.xlabel("epoch")
    plt.ylabel("loss")

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

    ### ========== TODO : START ========== ###
    ### part j: generate a plot to compare one_train_acc, one_valid_acc, two_train_acc, two_
    plt.figure()
    plt.plot(range(1, 31), one_train_acc, label="one-layer network training accuracy")
    plt.plot(range(1, 31), one_valid_acc, label="one-layer network validation accuracy")
    plt.plot(range(1, 31), two_train_acc, label="two-layer network training accuracy")
    plt.plot(range(1, 31), two_valid_acc, label="two-layer network validation accuracy")
    plt.legend()
    plt.xlabel("epoch")
    plt.ylabel("accuracy")
    ### ========== TODO : END ========== ##

    ### ========== TODO : START ========== ###
    ### part k: calculate the test accuracy
    one_test_acc = evaluate_acc(model_one, test_loader)
    two_test_acc = evaluate_acc(model_two, test_loader)
    print("One-Layer Network Test Accuracy: " + str(one_test_acc))
    print("Two-Layer Network Test Accuracy: " + str(two_test_acc))
    ### ========== TODO : END ========== ###

    ### ========== TODO : START ========== ###
    ### part l: replace the SGD optimizer with the Adam optimizer and do the experiments ag
    print("\n=======================================")
    print("Running experiments with Adam optimizer")
    print("=======================================\n")

    model_one = OneLayerNetwork()
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model_one.parameters(), lr=0.0005)

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

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

    print("Start training TwoLayerNetwork...")
    results_two = train(model_two, criterion, optimizer, train_loader, valid_loader)
    print("Done!")
```

```
    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(range(1, 31), one_train_loss, label="one-layer network training loss")
    plt.plot(range(1, 31), one_valid_loss, label="one-layer network validation loss")
    plt.plot(range(1, 31), two_train_loss, label="two-layer network training loss")
    plt.plot(range(1, 31), two_valid_loss, label="two-layer network validation loss")
    plt.legend()
    plt.xlabel("epoch")
    plt.ylabel("loss")

    plt.figure()
    plt.plot(range(1, 31), one_train_acc, label="one-layer network training accuracy")
    plt.plot(range(1, 31), one_valid_acc, label="one-layer network validation accuracy")
    plt.plot(range(1, 31), two_train_acc, label="two-layer network training accuracy")
    plt.plot(range(1, 31), two_valid_acc, label="two-layer network validation accuracy")
    plt.legend()
    plt.xlabel("epoch")
    plt.ylabel("accuracy")

    one_test_acc = evaluate_acc(model_one, test_loader)
    two_test_acc = evaluate_acc(model_two, test_loader)
    print("One-Layer Network Test Accuracy: " + str(one_test_acc))
    print("Two-Layer Network Test Accuracy: " + str(two_test_acc))

if __name__ == "__main__":
    main()
```