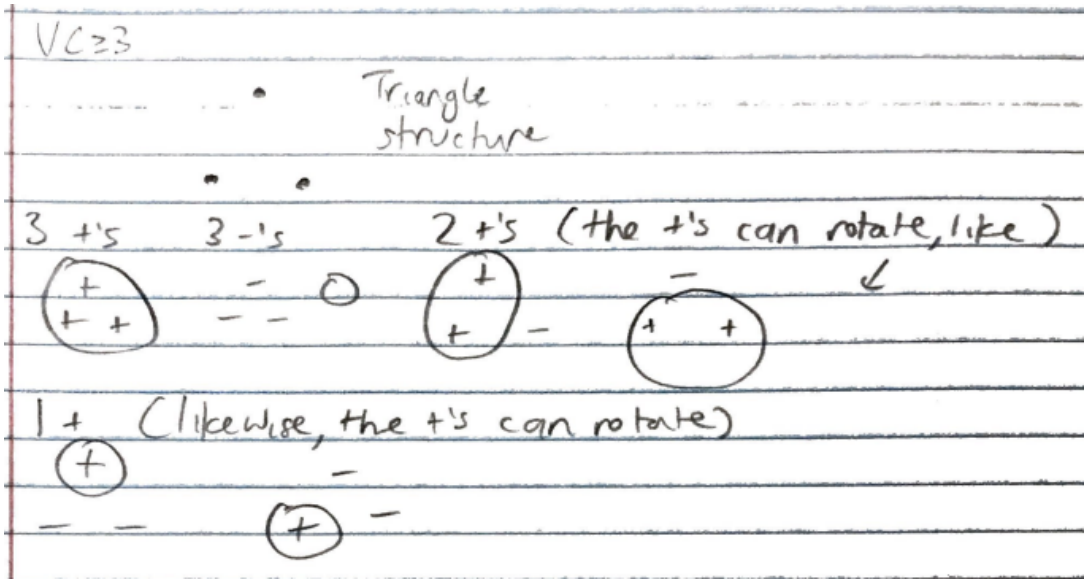
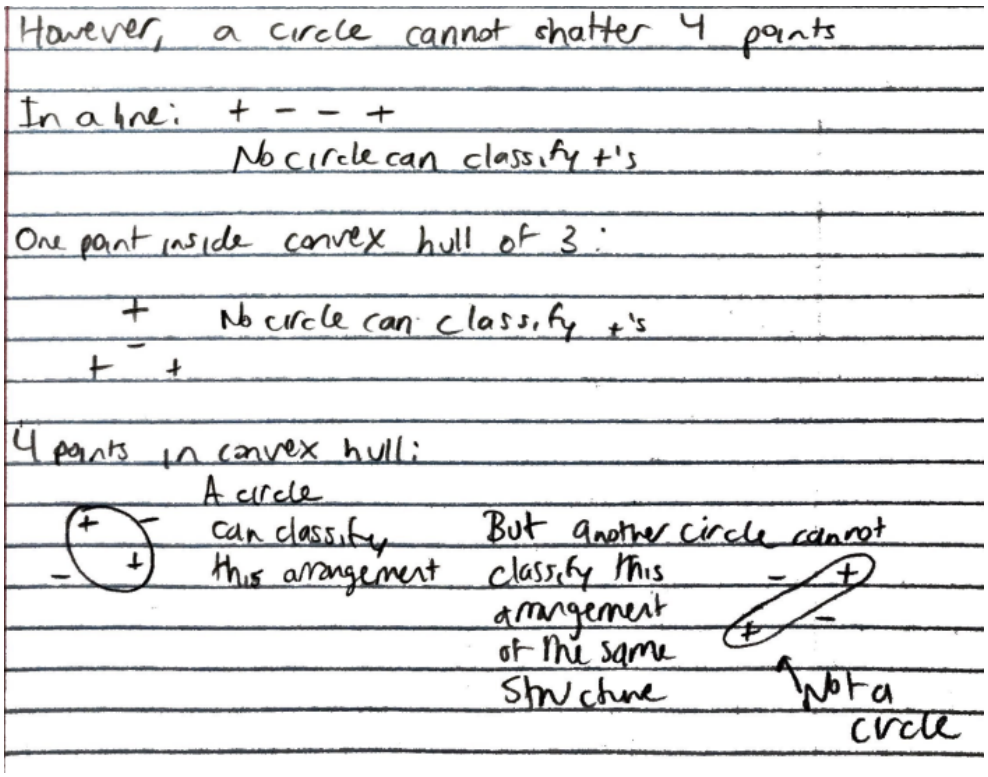
1 VC-Dimension [8 pts]

(a) Consider the space of instances X corresponding to all points in the x, y plane. What is the VC-dimension of the hypothesis space defined by H_c = circles in the x, y plane, with points inside the circle are classified as positive examples? Justify your answer (e.g. with one or more diagrams).

VC=3 (can shatter any 3 points that form a triangle, so VC >= 3)



However, cannot shatter 4 points:



If 4 points are in a line, labeled +,-,-,+, a circle cannot correctly classify the ones on the end as positive while also classifying the 2 middle ones as negative. Next, if 1 point is inside the convex hull of the other 3, and the one on the inside is -, then a circle cannot classify the outer 3 as positive while classifying the inside one as negative. Finally, if 4 points, labeled 1,2,3,4, make a convex hull, and are labeled clockwisely +,-,+,-, one circle may be able to classify 1 and 3 correctly, but another circle cannot classify if 2 and 4 are + correctly.

(b) This problem investigates a few properties of the VC dimension, mostly relating to how $VC(H)$ increases as the set H increases. For each part of this problem, you should state whether the given statement is true, and justify your answer with either a formal proof or a counter-example.

i. Let two hypothesis classes H_1 and H_2 satisfy $H_1 \subseteq H_2$. Prove or disprove: $VC(H_1) \leq VC(H_2)$.

ii. Let $H_1 = H_2 \cup H_3$. Prove or disprove: $VC(H_1) \leq VC(H_2) + VC(H_3)$.

i. **True**. If H1 = H2, then VC(H1) = VC(H2). If H1 is a proper subset of H2, then VC(H1) < VC(H2), since it is impossible to have more VC dimensions with a proper subset of H2 than H2 itself. Yes. All hypotheses that belong to H1 are in H2 so the shattered subset of H2 is at least as of H1.

ii. **False**. VC(H2) + VC(H3) < VC(H1) is possible. Let H2 be the set of hypotheses that always classifies a label as positive, and H3 be the set of hypotheses that always classifies a label as negative. VC(H2) = 0, since H2 cannot shatter a single point because H2 would not be able to classify it as negative. LIkewise, VC(H3) = 0, since H3 cannot shatter a single point because it would not be able to classify it as positive.

However, H1 = H2 ∪ H3, so H1 is the set of hypotheses that always predicts negative or always predicts positive; H1 = {0, 1}. So, VC(H1) = 1, since it can label a single point negative or positive, but it cannot shatter 2 points (since it will either label every point positive or negative, which would be incorrect for when there is a positive and negative point).

So VC(H1) > VC(H2) + VC(H3), since 1 > 0.

2 Kernels [8 pts]

(a) For any two documents x and z, define $k(x, z)$ to equal the number of unique words that occur in both x and 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.

Yes, this function is a kernel.

$k(x,z)$ is a kernel function. $k(x,z) = k(z,x)$ since the intersection of sets of words in x and z, and z and x is the same. $k(x,z) = ||x \cap z|| = k(z,x)$

To show $k(x,z)$ is a kernel, we use the Mercer Theorem. $k(x,z)$ is the size of the intersection of words in the documents.

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

To prove K is positive semi definite, we show all eigenvalues of K are nonnegative

$$\det(K - \lambda I) = 0$$

$$\det\begin{pmatrix} ||x|| - \lambda & ||x \cap z|| \\ ||x \cap z|| & ||z|| - \lambda \end{pmatrix} = (||x|| - \lambda)(||z|| - \lambda) - ||x \cap z||^2 = 0$$

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

To satisfy the equality:

We know $||x|| \geq ||x \cap z||$ and $||z|| \geq ||x \cap z||$

Also, $||x \cap z|| \leq \min(||x||, ||z||)$, since there cannot be more words in a subset than one of the sets the words are from

So, $||x \cap z||^2 \leq ||x||\,||z||$ for any x, z

We see that $\lambda \geq 0$ to satisfy the equality

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

Since all eigenvalues ≥ 0, K is PSD, so $k(x,z)$ is a kernel function

(b) 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:

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

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

$k(x,z) = xz$ is a kernel

Scaling:

$f(x) = \frac{1}{||x||}$ $f(z) = \frac{1}{||z||}$,

so $k_1(x,z) = f(x)\,k(x,z)\,f(z) = \frac{x}{||x||}\cdot\frac{z}{||z||}$ is kernel

Sum:

$k'(x,z) = 1$ is a kernel, since $K = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ Both $\lambda = 0$, K is PSD

$\det(K - \lambda I) = \det\begin{pmatrix} 1-\lambda & 1 \\ 1 & 1-\lambda \end{pmatrix} = (1-\lambda)^2 - 1 = \lambda^2 - 2\lambda = 0$

$\lambda = 0 \quad \lambda = 2$

so, $k_2(x,z) = k'(x,z) + k_1(x,z) = 1 + \frac{x}{||x||}\cdot\frac{z}{||z||}$ is a kernel

Product:

$k_3(x,z) = k_2(x,z)\cdot k_2(x,z)\cdot k_2(x,z)$

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

(c) 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?

$k_\beta(x, z) = (1 + \beta\, x \cdot z)^3, \quad \beta > 0$

$\phi_\beta(\cdot) = ?$

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

$= (1 + \beta x_1 z_1 + \beta x_2 z_2)(1 + \beta x_1 z_1 + \beta x_2 z_2 + \beta x_1 z_1 + \beta^2 x_1^2 z_1^2 + \beta^2 x_1 x_2 z_1 z_2 + \beta x_2 z_2 + \beta^2 x_1 x_2 z_1 z_2 + \beta^2 x_2^2 z_2^2)$

$= (1 + \beta x_1 z_1 + \beta x_2 z_2)(1 + 2\beta x_1 z_1 + 2\beta x_2 z_2 + 2\beta^2 x_1 x_2 z_1 z_2 + \beta^2 x_1^2 z_1^2 + \beta^2 x_2^2 z_2^2)$

$= 1 + 2\beta x_1 z_1 + 2\beta x_2 z_2 + 2\beta^2 x_1 x_2 z_1 z_2 + \beta^2 x_1^2 z_1^2 + \beta^2 x_2^2 z_2^2$
$+ \beta x_1 z_1 + 2\beta^2 x_1^2 z_1^2 + 2\beta^2 x_1 x_2 z_1 z_2 + 2\beta^3 x_1^2 x_2 z_1^2 z_2 + \beta^3 x_1^3 z_1^3 + \beta^3 x_1 z_1 x_2^2 z_2^2$
$+ \beta x_2 z_2 + 2\beta^2 x_1 z_1 x_2 z_2 + 2\beta^2 x_2^2 z_2^2 + 2\beta^3 x_1 z_1 x_2^2 z_2^2 + \beta^3 x_1^2 z_1^2 x_2 z_2 + \beta^3 x_2^3 z_2^3)$

$= 1 + 3\beta x_1 z_1 + 3\beta x_2 z_2 + 6\beta^2 x_1 z_1 x_2 z_2 + 3\beta^2 x_1^2 z_1^2 + 3\beta^2 x_2^2 z_2^2 + 3\beta^3 x_1^2 z_1^2 x_2 z_2 + 3\beta^3 x_1 z_1 x_2^2 z_2^2 + \beta^3 x_1^3 z_1^3 + \beta^3 x_2^3 z_2^3$

$= \phi(x)^T \cdot \phi(z)$

Dot product

Feature map $\rightarrow$

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

The above kernel and the same kernel where β = 1 are similar; the only difference is that when the constant β is present (β is not 1) , the transformation scales each point of the vector by the constant β.

B plays the role of a regularization term; for higher degree components (like for x_1^3) of the transformation vector, β is multiplied to a higher power (3/2) and it is a larger weight. For smaller degree components, β is a smaller weight.

3 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) 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?

Minimize $\frac{1}{2}\|\theta\|^2$ subject to $y_n \theta^T x_n \geq 1$

$y_n = -1$, $x = (a, e)^T$,

So, the constraint is $-\theta^T (a, e)^T \geq 1$

$\theta^T (a, e)^T \leq -1$

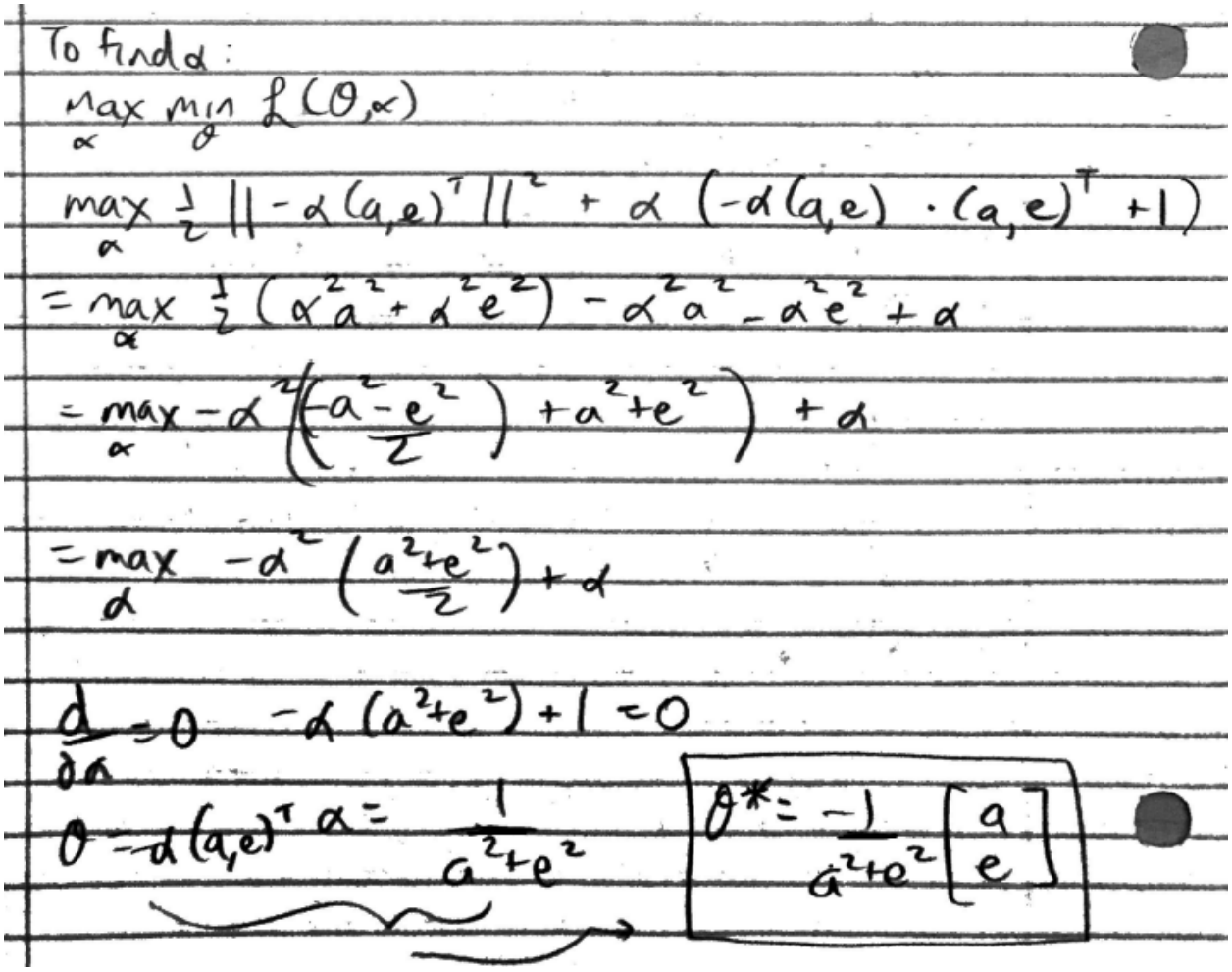
Dual form:

$\max_{\alpha} \min_{\theta} \mathcal{L}(\theta, \alpha)$ $\quad \theta^T (a, e)^T + 1 \leq 0$

$\mathcal{L}(\theta, \alpha) = \frac{1}{2}\|\theta\|^2 + \alpha(\theta^T (a, e)^T + 1)$

To find θ

$\frac{dL}{d\theta} = \theta + \alpha (a, e)^T = 0 \quad \theta = -\alpha (a, e)^T$



(b) 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?

$x_1 = (1,1)^T$, $x_2 = (1,0)^T$

$y_1 = 1$ $\quad$ $y_2 = -1$

θ^*, $\gamma = ?$

Constraints

$y_1 \theta^T x_1 \geq 1 \rightarrow \theta^T (1,1)^T \geq 1 \rightarrow -\theta^T (1,1)^T + 1 \leq 0$

$y_2 \theta^T x_2 \geq 1 \rightarrow -\theta^T (1,0)^T \geq 1 \rightarrow \theta^T (1,0)^T + 1 \leq 0$

$$\mathcal{L}(\theta, \alpha_1, \alpha_2) = \frac{1}{2}\|\theta\|^2 + \alpha_1 (-\theta^T (1,1)^T + 1) + \alpha_2 (\theta^T (1,0)^T + 1)$$

$$\frac{d\mathcal{L}}{\partial \theta} = \theta - \alpha_1 (1,1)^T + \alpha_2 (1,0)^T = 0$$

$$\theta = \alpha_1 (1,1)^T - \alpha_2 (1,0)^T = \begin{bmatrix} \alpha_1 - \alpha_2 \\ \alpha_1 \end{bmatrix}$$

$$\max_{\alpha} \min_{\theta} \mathcal{L}(\theta, \alpha_1, \alpha_2)$$

$$= \max_{\alpha} \frac{1}{2} \left\| \begin{bmatrix} \alpha_1 - \alpha_2 \\ \alpha_1 \end{bmatrix} \right\|^2 + \alpha_1 \left(- \begin{bmatrix} \alpha_1 - \alpha_2 \\ \alpha_1 \end{bmatrix}^T (1,1)^T + 1 \right) + \alpha_2 \left(\begin{bmatrix} \alpha_1 - \alpha_2 \\ \alpha_1 \end{bmatrix}^T \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 1 \right)$$

$$= \frac{1}{2}(\alpha_1^2 - 2\alpha_1\alpha_2 + \alpha_2^2 + \alpha_1^2) + \alpha_1(\alpha_2 - \alpha_1 - \alpha_1) + \alpha_1 + \alpha_1\alpha_2 - \alpha_2^2 + \alpha_2$$

$$= \alpha_1^2 - \alpha_1\alpha_2 + \frac{1}{2}\alpha_2^2 + \alpha_1\alpha_2 - 2\alpha_1^2 + \alpha_1 + \alpha_1\alpha_2 - \alpha_2^2 + \alpha_2$$

$$= -\alpha_1^2 + \alpha_1\alpha_2 - \frac{1}{2}\alpha_2^2 + \alpha_1 + \alpha_2$$

$$\frac{\partial \mathcal{L}}{\partial \alpha_1} = -2\alpha_1 + \alpha_2 + 1 = 0$$

$$\frac{\partial \mathcal{L}}{\partial \alpha_2} = \alpha_1 - \alpha_2 + 1 = 0$$

$\rightarrow -\alpha_1 + 2$

$\alpha_1 = 2$, $\alpha_2 = 3$

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

So, $\text{margin} = \frac{1}{\sqrt{(-1)^2 + 2^2}} = \frac{1}{\sqrt{5}}$

(c) 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.

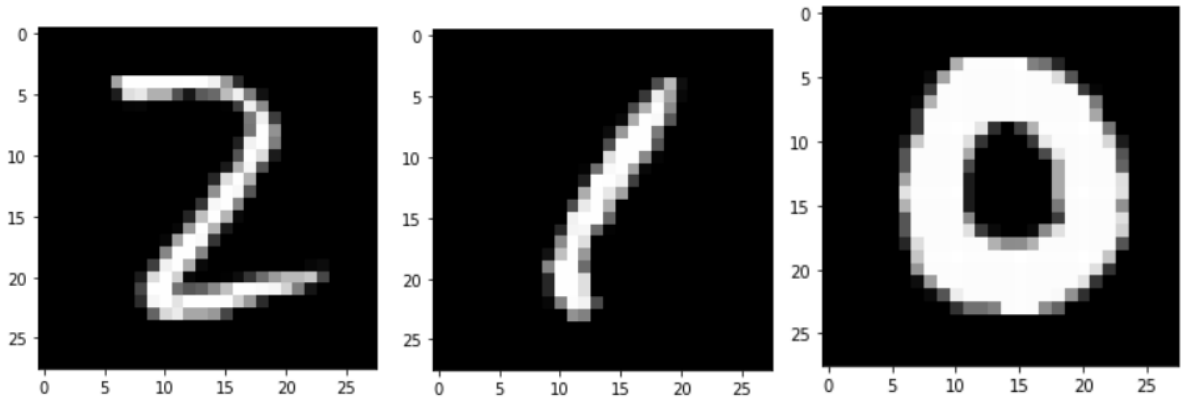
Now that we have an offset term, the boundary is now clearly a horizontal line $x_2 = 0.5$. There is no dependence of x_1, so when we find θ*, we can set $w_1 = 0$. $\theta^* = [0, w_2]$. For $x_1 = (1,1)^T$, the constraint is $w_2 + b \geq 1$, and for $x_2 = (1,0)^T$, the constraint is $-b \geq 1 \rightarrow b \leq -1$. So, $b = -1$, $w_1 = 0$, and $w_2 = 2$. **$\theta^* = [0, 2]$, $b^* = -1$, and the margin $\gamma = 1/2$**. (horizontal line $x_2 = 0.5$ has distance 1/2 from both $[1,1]^T$ and $[1, 0]^T$).

With the offset term, the margin ($\gamma = 1/2$) is larger than the margin without the offset term ($\gamma = 1/\text{sqrt}(5)$), so the offset term provides a better classifier.

4 Implementation: Digit Recognizer [48 pts]

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

```
### ========== TODO : START ========== ###
### part a: print out three training images with different labels
plot_img(X_train[50])
plot_img(X_train[130])
plot_img(X_train[150])
### ========== TODO : END ========== ###
```



(b) The loaded examples are numpy arrays. Convert the numpy arrays to tensors. You do not need to submit anything for this part.[3 pts]

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

(c) 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. [5 pts]

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_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)
valid_loader = DataLoader(valid_dataset, batch_size=10)
test_loader = DataLoader(test_dataset, batch_size=10)
### ========== TODO : END ========== ###
```

(d) 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 sigmoid function here since we will use `torch.nn.CrossEntropyLoss` later. [5 pts]

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`.

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

        ### ========== TODO : START ========== ###
        ### part d: implement OneLayerNetwork with torch.nn.Linear
        # in_features = 784, out_features = 3
        self.one_layer = 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.one_layer(x)
        ### ========== TODO : END ========== ###
        return outputs
```

(e) 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` [2 pts]

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.parameters(), lr=0.0005)

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

(f) 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. [8 pts]

```
Start training OneLayerNetwork...
| epoch  1 | train loss 1.075387 | train acc 0.453333 | valid loss ...
| epoch  2 | train loss 1.021301 | train acc 0.563333 | valid loss ...
| epoch  3 | train loss 0.972599 | train acc 0.630000 | valid loss ...
| epoch  4 | train loss 0.928335 | train acc 0.710000 | valid loss ...
...
```

```
### ========== TODO : START ========== ###
### part f: implement the training process
y_pred = model.forward(batch_X)   # forward pass
optimizer.zero_grad()             # initialize gradients to zero
loss = criterion(y_pred, batch_y) # compute loss
loss.backward()                   # loss backward
optimizer.step()                  # update model parameters
### ========== TODO : END ========== ###
```

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

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

```python
class TwoLayerNetwork(torch.nn.Module):
    def __init__(self):
        super(TwoLayerNetwork, self).__init__()
        ### ========== TODO : START ========== ###
        ### part g: implement TwoLayerNetwork with torch.nn.Linear
        self.first_layer = torch.nn.Linear(784, 400)
        self.second_layer = 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
        layer_1 = self.first_layer(x)
        sigmoid = torch.nn.Sigmoid()
        layer_1 = sigmoid(layer_1)
        outputs = self.second_layer(layer_1)
        ### ========== TODO : END ========== ###
        return outputs
```

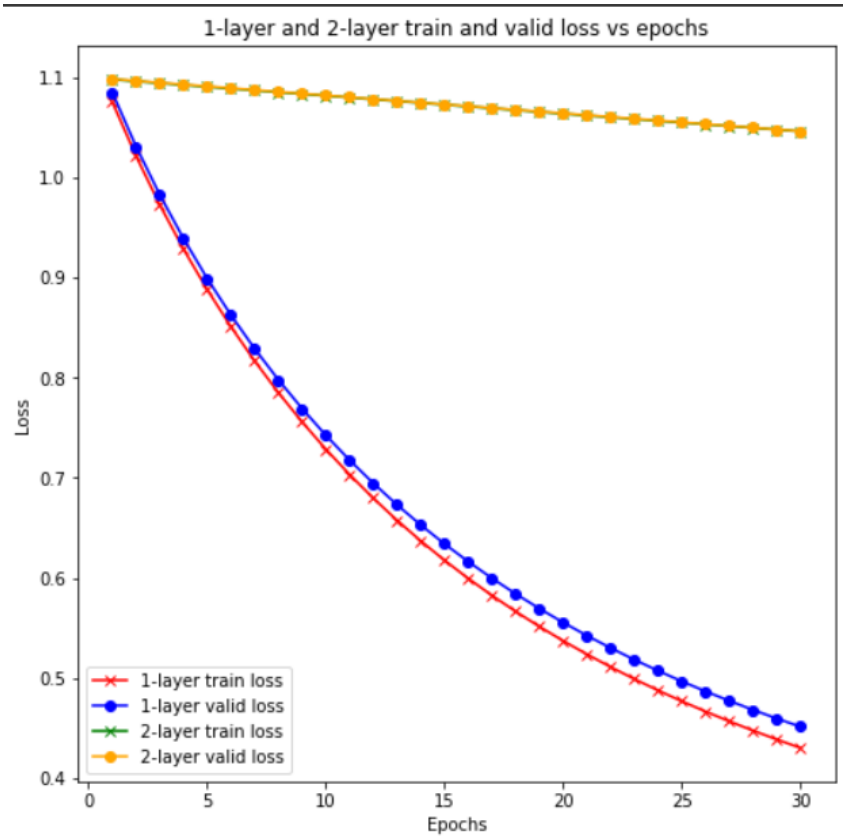
(h) Create an instance of `TwoLayerNetwork`, set up a criterion with `torch.nn.CrossEntropyLoss`, and set up a SGD optimizer with learning rate 0.0005 by using `torch.optim.SGD`. Then train `TwoLayerNetwork`. [2 pts]

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

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

(i) 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. [3 pts]

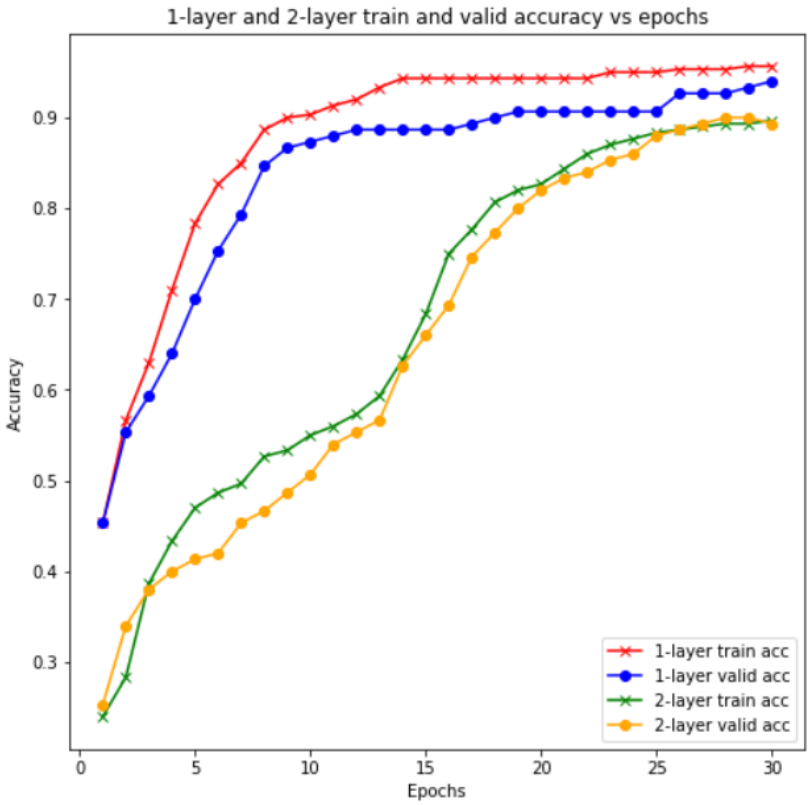
```
### ========== TODO : START ========== ###
### part i: generate a plot to comare one_train_loss, one_valid_loss, two_train_loss, two_valid_loss
epochs = np.arange(1, 31)
plt.figure(figsize=[10,10])
plt.plot(epochs, one_train_loss, color='red', label='1-layer train loss', marker='x')
plt.plot(epochs, one_valid_loss, color='blue', label='1-layer valid loss', marker='o')
plt.plot(epochs, two_train_loss, color='green', label='2-layer train loss', marker='x')
plt.plot(epochs, two_valid_loss, color='orange', label='2-layer valid loss', marker='o')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('1-layer and 2-layer train and valid loss vs epochs')
plt.legend()
plt.show()
### ========== TODO : END ========== ###
```



For each of the 1-layer and 2-layer networks, the train loss and validation loss are nearly the same, which tells us there is no overfitting (from what we can see in the plot).

The train and valid loss for the 1-layer network decreases in an exponential decay fashion, while the loss for the 2-layer network decreases more slowly, in a linear fashion. The loss decreases faster for the 1-layer network because it has a much smaller number of parameters compared to the 2-layer network, so it takes longer for the loss to decrease for the 2-layer network. SInce we only use 30 epochs for both the 1-layer and 2-layer network, the loss for the 1-layer network decreases a lot, while the loss for the 2-layer network hardly decreases in comparison.

(j) 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. [3 pts]



We can see similar findings from the previous plot. Both the training and validation accuracy increase quickly towards close to 100% (around 95%) for the 1-layer network; the accuracies for the 2-layer network increase more slowly and do not get as close to 100% as the 1-layer network accuracies do (instead, they seem to plateau at around 90% after 25-27 epochs).

(k) 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 ? [3 pts]

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

```
Test accuracy of 1-layer network:  0.9599999785423279
Test accuracy of 2-layer network:  0.8999999761581421
```

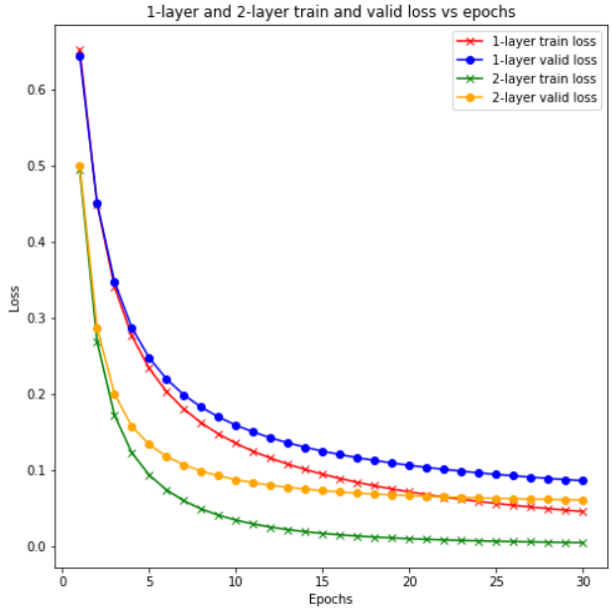
The test accuracy of the 1-layer network is 0.96, which is higher than the test accuracy of the 2-layer network, which is 0.9. This is because the 2-layer network has a higher number of parameters, so it needs more epochs to achieve the same accuracy as the 1-layer network's accuracy.

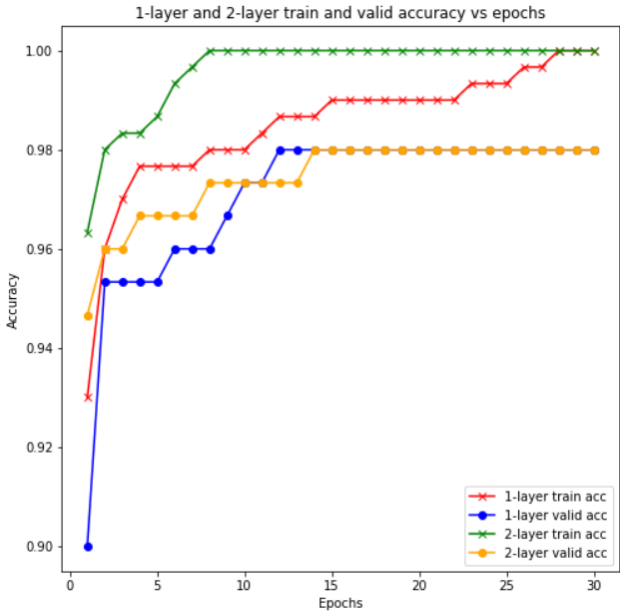
(l) Replace the SGD optimizer with the Adam optimizer and do the experiments again. Show the loss figure, the accuracy figure, and the test accuracy. Include the figures in the report and describe your findings. [7 pts]

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

```
Start training OneLayerNetwork...
| epoch  1 | train loss 0.652176 | train acc 0.930000 | valid loss 0.643495 | valid acc 0.900000 |
| epoch  2 | train loss 0.448845 | train acc 0.960000 | valid loss 0.450157 | valid acc 0.953333 |
| epoch  3 | train loss 0.340191 | train acc 0.970000 | valid loss 0.347157 | valid acc 0.953333 |
| epoch  4 | train loss 0.275654 | train acc 0.976667 | valid loss 0.286565 | valid acc 0.953333 |
| epoch  5 | train loss 0.233069 | train acc 0.976667 | valid loss 0.247042 | valid acc 0.953333 |
| epoch  6 | train loss 0.202674 | train acc 0.976667 | valid loss 0.219164 | valid acc 0.960000 |
| epoch  7 | train loss 0.179708 | train acc 0.976667 | valid loss 0.198359 | valid acc 0.960000 |
| epoch  8 | train loss 0.161607 | train acc 0.980000 | valid loss 0.182174 | valid acc 0.960000 |
| epoch  9 | train loss 0.146879 | train acc 0.980000 | valid loss 0.169180 | valid acc 0.966667 |
| epoch 10 | train loss 0.134596 | train acc 0.980000 | valid loss 0.158487 | valid acc 0.973333 |
| epoch 11 | train loss 0.124151 | train acc 0.983333 | valid loss 0.149514 | valid acc 0.973333 |
| epoch 12 | train loss 0.115131 | train acc 0.986667 | valid loss 0.141863 | valid acc 0.980000 |
| epoch 13 | train loss 0.107242 | train acc 0.986667 | valid loss 0.135252 | valid acc 0.980000 |
| epoch 14 | train loss 0.100269 | train acc 0.986667 | valid loss 0.129476 | valid acc 0.980000 |
| epoch 15 | train loss 0.094054 | train acc 0.990000 | valid loss 0.124380 | valid acc 0.980000 |
| epoch 16 | train loss 0.088473 | train acc 0.990000 | valid loss 0.119847 | valid acc 0.980000 |
| epoch 17 | train loss 0.083429 | train acc 0.990000 | valid loss 0.115785 | valid acc 0.980000 |
| epoch 18 | train loss 0.078848 | train acc 0.990000 | valid loss 0.112121 | valid acc 0.980000 |
| epoch 19 | train loss 0.074666 | train acc 0.990000 | valid loss 0.108797 | valid acc 0.980000 |
| epoch 20 | train loss 0.070834 | train acc 0.990000 | valid loss 0.105767 | valid acc 0.980000 |
| epoch 21 | train loss 0.067309 | train acc 0.990000 | valid loss 0.102990 | valid acc 0.980000 |
| epoch 22 | train loss 0.064057 | train acc 0.990000 | valid loss 0.100436 | valid acc 0.980000 |
| epoch 23 | train loss 0.061046 | train acc 0.993333 | valid loss 0.098077 | valid acc 0.980000 |
| epoch 24 | train loss 0.058253 | train acc 0.993333 | valid loss 0.095891 | valid acc 0.980000 |
| epoch 25 | train loss 0.055655 | train acc 0.993333 | valid loss 0.093858 | valid acc 0.980000 |
| epoch 26 | train loss 0.053232 | train acc 0.996667 | valid loss 0.091962 | valid acc 0.980000 |
| epoch 27 | train loss 0.050969 | train acc 0.996667 | valid loss 0.090190 | valid acc 0.980000 |
| epoch 28 | train loss 0.048851 | train acc 1.000000 | valid loss 0.088529 | valid acc 0.980000 |
| epoch 29 | train loss 0.046865 | train acc 1.000000 | valid loss 0.086968 | valid acc 0.980000 |
| epoch 30 | train loss 0.045000 | train acc 1.000000 | valid loss 0.085498 | valid acc 0.980000 |
Done!
```

```
Start training TwoLayerNetwork...
| epoch  1 | train loss 0.493845 | train acc 0.963333 | valid loss 0.499119 | valid acc 0.946667 |
| epoch  2 | train loss 0.268077 | train acc 0.980000 | valid loss 0.286093 | valid acc 0.960000 |
| epoch  3 | train loss 0.171713 | train acc 0.983333 | valid loss 0.199556 | valid acc 0.960000 |
| epoch  4 | train loss 0.122286 | train acc 0.983333 | valid loss 0.156994 | valid acc 0.966667 |
| epoch  5 | train loss 0.093136 | train acc 0.986667 | valid loss 0.133069 | valid acc 0.966667 |
| epoch  6 | train loss 0.073462 | train acc 0.993333 | valid loss 0.117480 | valid acc 0.966667 |
| epoch  7 | train loss 0.059191 | train acc 0.996667 | valid loss 0.106445 | valid acc 0.966667 |
| epoch  8 | train loss 0.048472 | train acc 1.000000 | valid loss 0.098250 | valid acc 0.973333 |
| epoch  9 | train loss 0.040236 | train acc 1.000000 | valid loss 0.091944 | valid acc 0.973333 |
| epoch 10 | train loss 0.033800 | train acc 1.000000 | valid loss 0.086957 | valid acc 0.973333 |
| epoch 11 | train loss 0.028701 | train acc 1.000000 | valid loss 0.082925 | valid acc 0.973333 |
| epoch 12 | train loss 0.024610 | train acc 1.000000 | valid loss 0.079609 | valid acc 0.973333 |
| epoch 13 | train loss 0.021291 | train acc 1.000000 | valid loss 0.076844 | valid acc 0.973333 |
| epoch 14 | train loss 0.018572 | train acc 1.000000 | valid loss 0.074512 | valid acc 0.980000 |
| epoch 15 | train loss 0.016322 | train acc 1.000000 | valid loss 0.072525 | valid acc 0.980000 |
| epoch 16 | train loss 0.014443 | train acc 1.000000 | valid loss 0.070818 | valid acc 0.980000 |
| epoch 17 | train loss 0.012861 | train acc 1.000000 | valid loss 0.069341 | valid acc 0.980000 |
| epoch 18 | train loss 0.011519 | train acc 1.000000 | valid loss 0.068053 | valid acc 0.980000 |
| epoch 19 | train loss 0.010371 | train acc 1.000000 | valid loss 0.066924 | valid acc 0.980000 |
| epoch 20 | train loss 0.009382 | train acc 1.000000 | valid loss 0.065928 | valid acc 0.980000 |
| epoch 21 | train loss 0.008525 | train acc 1.000000 | valid loss 0.065045 | valid acc 0.980000 |
| epoch 22 | train loss 0.007778 | train acc 1.000000 | valid loss 0.064258 | valid acc 0.980000 |
| epoch 23 | train loss 0.007123 | train acc 1.000000 | valid loss 0.063554 | valid acc 0.980000 |
| epoch 24 | train loss 0.006546 | train acc 1.000000 | valid loss 0.062922 | valid acc 0.980000 |
| epoch 25 | train loss 0.006035 | train acc 1.000000 | valid loss 0.062352 | valid acc 0.980000 |
| epoch 26 | train loss 0.005581 | train acc 1.000000 | valid loss 0.061836 | valid acc 0.980000 |
| epoch 27 | train loss 0.005174 | train acc 1.000000 | valid loss 0.061368 | valid acc 0.980000 |
| epoch 28 | train loss 0.004810 | train acc 1.000000 | valid loss 0.060942 | valid acc 0.980000 |
| epoch 29 | train loss 0.004482 | train acc 1.000000 | valid loss 0.060553 | valid acc 0.980000 |
| epoch 30 | train loss 0.004186 | train acc 1.000000 | valid loss 0.060198 | valid acc 0.980000 |
Done!
```





With Adam, the loss for both the 1-layer and 2-layer converge to 0 much faster with Adam than with SGD (with SGD, the loss does not even reach 0 with 30 epochs).

The biggest difference between the SGD and Adam optimizers is that the 2-layer network's loss converges much faster to 0 (both the 1-layer and 2-layer loss exponentially decay). In fact, the 2-layer loss converges faster to 0 than the 1-layer loss does. Additionally, the 2-layer training accuracy increases very fast to 1, faster than the 1-layer training accuracy, while the validation accuracy converges to 0.98 for both.

```
Test accuracy of 1-layer network:  0.9733333587646484
Test accuracy of 2-layer network:  0.9666666388511658
```

Finally, the test accuracy increased when using Adam (from 0.96 to 0.973 for the 1-layer and 0.90 to 0.96 for the 2-layer).

The Adam optimizer converges faster than the SGD does because Adam uses AdaGrad (which maintains a per-parameter adaptive learning rate) and momentum (which accelerates stochastic gradient descent in the relevant direction - increases the update speed if gradients are in same direction, decreases if in different direction). Adam allows the 2-layer network to converge faster so it can have similar results to the 1-layer network in 30 epochs, which was too few epochs for the SGD optimizer for 2-layer network.

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

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

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