CM146, Fall 2020
Problem Set 3: VC Dimension and Neural Networks
Due Nov. 30 at 11:59 pm

Ian Conceicao

Monday, November 30th 2020

1 VC Dimension [16 pts]

For the following problems, we classify a point x to either a positive label $+1$ or a negative label -1 by a hypothesis set $\mathcal{H}$.

(a) *Positive ray classifier.* Consider $x \in \mathbb{R}$ and $\mathcal{H} = \{sign(x - b) \mid b \in \mathbb{R}\}$. That is, the label is $+1$ if x is greater than b otherwise -1. [8 pts]

i. For N points, prove that there are at most $N + 1$ label outcomes that can be generated by $\mathcal{H}$. For example, when we have 4 points, $x_1 \leq x_2 \leq x_3 \leq x_4$, there are at most 5 label outcomes can be generated by $\mathcal{H}$, as shown by the following.

x_1	x_2	x_3	x_4
$+1$	$+1$	$+1$	$+1$
-1	$+1$	$+1$	$+1$
-1	-1	$+1$	$+1$
-1	-1	-1	$+1$
-1	-1	-1	-1

Solution: Assume we are given N outcomes: $x_1 \leq x_2 ... \leq x_N$. We can fix all points positive, and 1 by 1 we can we can make x_i negative, from $i = 1$ to $i = N$, inclusive. This the same as moving the b value of the line between different points. This will take N iterations. In total we have a base case plus N, or $N + 1$ different sets of outcomes. We can conclude: for N points, we have $N + 1$ possible different label outcomes.

ii. What is the VC dimension of $\mathcal{H}$?

Solution: The VC dimension of $\mathcal{H}$ is 1.
We can see 1 is shatterable in the following example: Place $x_1 = 2$. If the target of x_1 is positive, we can set $b = 1$. If the target of x_1 is negative, we can set $b = 3$.
We can also show that when $N = 2$ there is no possible way to shatter the points. Let's divide it into two cases:
Case 1: $x_1 = x_2$. Here we can imagine both points at the same point on a number line. If their labels are different, we cannot set a b value to separate them. **Case 2:** $x_1 > x_2$. Here we can image x_1 on the right of x_2 on the number line. When x_2 is positive and x_1 is negative, there is no line that will correctly the label the data, because everything to the right of the line b will be labeled positive, and everything to the left will be labeled negative. Setting $b > x_1, x_2$ would incorrectly label x_2 as negative. Setting b: $x_2 < b < x_1$ would incorrectly label both. Setting $b < x_1, x_2$ would incorrectly label x_1

Part of this assignment is adapted from the course materials by Hsuan-Tien Lin (National Taiwan University).

as positive. This case is analogous to $x_1 < x_2$.
Because $\mathcal{H}$ can shatter 1 point, but not any 2 points, we can conclude $\mathcal{H}$ has a VC dimension of 1.
We can also prove the VC dimension of $\mathcal{H}$ another way. For the first value of N that the growth function is less than 2^N, the VC dimension is N-1. We found in part (i) the growth function is N+1, so we can make the following table:

N	Growth (N+1)	2^N
1	2	2
2	3	4
3	4	8

The first N value where 2^N is greater than the growth function is $N = 2$, therefore the VC dimension of $\mathcal{H}$ is $\boxed{1}$.

(b) *Positive interval classifier.* Consider $x \in \mathbb{R}$ and $\mathcal{H} = \{sign(\mathbb{1}(x \in [a, b]) - 0.5) \mid a, b \in \mathbb{R}, a \leq b\}$, where $\mathbb{1}(.)$ is the indicator function. That is, the label is $+1$ if x in the interval $[a, b]$ otherwise -1. [8 pts]

i. For N points, prove that there are at most $(\frac{N^2+N}{2} + 1)$ label outcomes that can be generated by $\mathcal{H}$.
Solution: Imagine N points on a number line. None of these points are the same. Therefore we have $N + 1$ "spaces" between these points. Spaces are defined as the set of points between 2 closest points, and the set of points between $-\infty$ and the smallest point, and ∞ and the greatest point. Therefore we $N + 1$ spaces. We can choose to place 2 lines in these $N + 1$ spaces. This can be modelled with the combination formula: $C(n, r) = \frac{n!}{r!(n-r)!}$
We are picking between $N + 1$ spaces with 2 lines, so we set $n = N + 1$ and $r = 2$:
$C(N + 1, 2) = \frac{(N+1)!}{2!((N+1)-2)!}$
$= \frac{(N+1)!}{2(N-1)!}$
$= \frac{(N+1)(N)(N-1)}{2(N-1)!}$
$= \frac{(N+1)(N))}{2}$
$= \frac{N^2+N}{2}$
Because combination works without replacement, we need to model the situation where the 2 lines choose the same "space" and all points are features are labeled negative. Adding 1 to our term we get:
$\frac{N^2+N}{2} + 1$

ii. What is the VC dimension of $\mathcal{H}$?
Solution: Once again we can make a table of the growth function and 2^N to get the VC dimension:

N	Growth $(\frac{N^2+N}{2} + 1)$	2^N
1	2	2
2	4	4
3	7	8
4	11	16

The first N where the growth function is less than 2^N is $N = 3$. Therefore the VC dimension is $\boxed{2}$.

2 Bound of VC dimension [16 pts]

Assume the VC dimension of an empty set is zero. Now, we have two hypothesis sets $\mathcal{H}_1$ and $\mathcal{H}_2$.

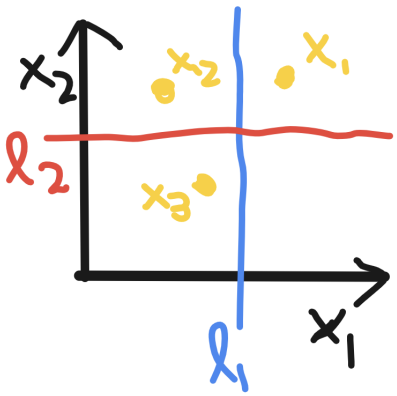
(a) Let $\mathcal{H}_3 = \mathcal{H}_1 \cap \mathcal{H}_2$. Show that $VC(\mathcal{H}_1) \geq VC(\mathcal{H}_3)$. [6 pts]

Solution: The VC dimension of a hypothesis set, H, is the max subset of size d of instance space x that can be shattered by H. To shatter a subset of size d, we must have a model in the hypothesis set, H, say H_i that can shatter d. Therefore if H cannot shatter d, then H must be missing some hypothesis, H_i that can shatter d. Because of this, we can say: if H cannot shatter d, then any subset of H cannot shatter d. Looking at the problem, $\mathcal{H}_3$ is the intersection of two subsets of $\mathcal{H}_1$ and $\mathcal{H}_2$. We already showed that the maximum shatterable subset of instance space x, or VC dimension, does not increase when we take a subset of a hypothesis space. Therefore, the VC dimension of $\mathcal{H}_3$ cannot be larger than the VC dimension of $\mathcal{H}_2$ and it cannot be larger than the VC dimension of $\mathcal{H}_1$.
We can further show this with a proof by contradiction. Assume $\mathcal{H}_3$ can shatter a subset of instance space x of size d. Then the VC dimension of $\mathcal{H}_3$ is d or greater. Let's also assume d is greater than the VC dimension of $\mathcal{H}_2$. Now we have it established that $VC(\mathcal{H}_1) < VC(\mathcal{H}_3)$. Let's call the specific model that can shatter this subset x, H_i. Because all elements of $\mathcal{H}_3$ come from a subset of $\mathcal{H}_2$ due to the intersection operation, this model, H_i must be in $\mathcal{H}_2$. Because this model can shatter a size of d, the VC dimension of $\mathcal{H}_2$ is d or greater. This contradicts our original claim that $VC(\mathcal{H}_1) < VC(\mathcal{H}_3)$, therefore: $VC(\mathcal{H}_1) \geq VC(\mathcal{H}_3)$

Proven.

(b) Let $\mathcal{H}_3 = \mathcal{H}_1 \cup \mathcal{H}_2$. Give an example to show that $VC(\mathcal{H}_1) + VC(\mathcal{H}_2) < VC(\mathcal{H}_3)$ is possible. [10 pts]

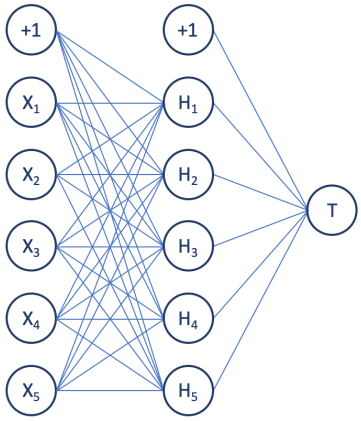
Solution: Assume our instance space is: X_1, X_2, X_3. $\mathcal{H}_1$ contains only one hypothesis, a linear classifier, l_1: $w_1 * x_1 + b$ and $\mathcal{H}_2$ contains only one hypothesis, a linear classifier, l_2: $w_1 * x_2 + b$. Both hypothesis spaces have VC's of 2, because they both only contain 1, 1 term linear classifier. However, when the output of each classifier is "or'd" or "and"'d together, the two lines can shatter 3 points:



These new lines would come together to make a 2 layer neural network. The first levels are the first linear classifiers, and the next level is trained on how to combine them. It can combine them to either perform an "or" or an "and" operation. Because H_3 can shatter 3 points, this new model would have a VC of 3, which is greater than the VC dimensions of H_2 and H_1.

3 Neural Network [20 pts]

We design a neural network to implement the XOR operation of X_1, X_2, X_3, X_4, X_5. We use $+1$ to represent `true` and -1 to represent `false`. Consider the following neural network.



We use w_{ij} to represent the weight between X_i and H_j, and use w_{0j} to represent the weight between the first layer bias term $(+1)$ and H_j. We use v_i to represent the weight between H_i and T, and use v_0 to represent the weight between the second layer bias term $(+1)$ and T.

Now, let $X_i \in \{+1, -1\}$, $H_j = sign\left(\sum_{i=0}^{5} w_{ij} X_i\right)$, and $T = sign\left(\sum_{i=0}^{5} v_i H_i\right)$.

(a) Specify w_{ij} such that H_j is $+1$ if there are at least j positive values among X_1, X_2, X_3, X_4, X_5, otherwise -1. If there are multiple acceptable weights, you only need to write down one of them. [8 pts]

Solution: First, we can set the non-bias terms to equal weights of 1:

i. for i in $1, 2, ...5$:

   i. $w_{ij} = 1$

Now we can observe the summation of these weights under the most minimal negative situation:

j	Minimum Negative Combination	Sum	Proposed Bias W_{0j}
1	5 Negative, 0 Positive	-5	4.5
2	4 Negative, 1 Positive	-3	2.5
3	3 Negative, 2 Positive	-1	0.5
4	2 Negative, 3 Positive	1	-1.5
5	1 Negative, 4 Positive	3	-3.5

By observing the sum of these minimal negative scenarios, we can propose a bias for each j to make such a value the lowest possible negative combination, at -0.5. By analyzing the table we can make the following rule to make the bias terms:

i. $W_{0j} = 6.5 - 2j$

(b) Given w_{ij} and H_j defined as above, specify v_i such that the whole neural network behaves like the XOR operation of X_1, X_2, X_3, X_4, X_5. If there are multiple acceptable weights, you only need to write down one of them. [8 pts]

Solution: H_i tells us if there are at least i positives. We want to return 1 if there are an odd number of positives and -1 if there are an even number of positives. Therefore I chose to assign positive values to odd weights and negative values to even weights. I also realized the earlier weights were more important than the later ones and through trial and error I created:

i. $w_o = -10$
ii. $w_1 = 13$
iii. $w_2 = -8$
iv. $w_3 = 7$
v. $w_4 = -6$
vi. $w_5 = 5$

I verified the results by creating the table:

	0 Positive	1 Positive	2 Positive	3 Positive	4 Positive	5 Positive
w_0	-10	-10	-10	-10	-10	-10
$w_1 * H_1$	-13	13	13	13	13	13
$w_2 * H_2$	8	8	-8	-8	-8	-8
$w_3 * H_3$	-7	-7	-7	7	7	7
$w_4 * H_4$	6	6	6	6	-6	-6
$w_5 * H_5$	-5	-5	-5	-5	-5	5
Sum	-4	5	-11	3	-9	1

We can see the weights do indeed work because each sum is greater than 0 if and only if an odd number of x_i are positive.

(c) Justify why the output of the neural network behaves like the XOR operation of X_1, X_2, X_3, X_4, X_5. [4 pts]

Solution: An XOR operation of X_1, X_2, X_3, X_4, X_5 returns true if and only if an odd number of X_i's are positive. Our neural network models this relationship by first, telling us for each i, if there are at least i positive. The H_i layer does this. Then the output looks at all of the H_is', and assigns positive weights to to the H_i's with odd i's, and negative weights to the H_i's with even i's. It also gives a negative bias term: w_0, that the positive terms will have to overcome. And because the earlier H_i's are more 'important' (for example we can have 1 positive and 3 positive will be wrong and 5 positive will be wrong), the earlier terms are weighted with more magnitude than the first ones. For further proof that each level works, view the tables in each previous section.

4 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 the row pixels. 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 : `Fall2020-CS146-HW3.ipynb`
- data : `hw3_train.csv`, `hw3_valid.csv`, `hw3_test.csv`

Please use your *@g.ucla.edu* email id to access the code and data. Similar to *HW-1*, 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. For colab usage demo, check out the Discussion recordings for Week 2 in CCLE. The notebook has marked blocks where you need to code.

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

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

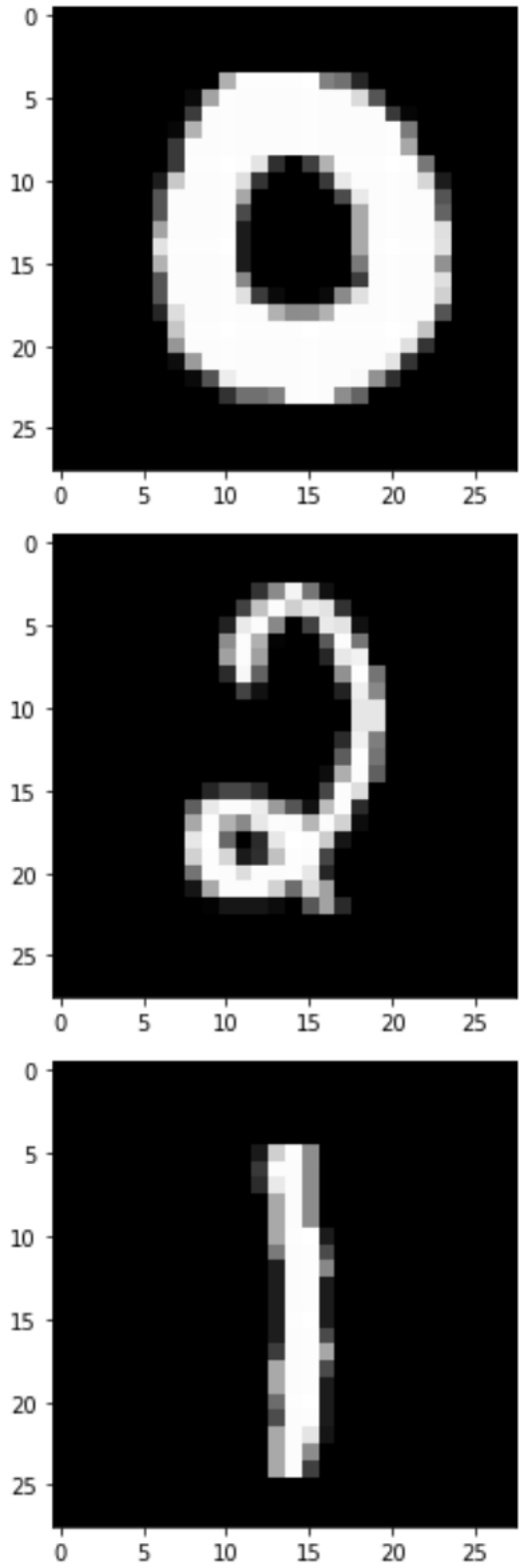
Note: For the questions requiring you to complete a piece of code, you are expected to copy-paste your code as a part of the solution in the submission pdf. Tip: If you are using LaTeX, check out the Minted package (example) for code highlighting.

Data Visualization and Preparation [10 pts]

(a) 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]

Solution:

```
plot_img(X_train[150])
print("Label for first example: {}".format(y_train[150]))
plot_img(X_train[27])
print("Label for second example: {}".format(y_train[27]))
plot_img(X_train[201])
print("Label for third example: {}".format(y_train[201]))
```



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

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) 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`.

Solution:

```
train_data = TensorDataset(X_train,y_train)
valid_data = TensorDataset(X_valid,y_valid)
test_data = TensorDataset(X_test,y_test)
train_loader = DataLoader(train_data, shuffle=True, batch_size=10)
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=10)
test_loader = DataLoader(test_data, shuffle=False, 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 $\mathbf{p}_n = \sigma(\mathbf{W}^\top \mathbf{x}_n)$, where $\sigma(.)$ is the element-wise sigmoid function and $\mathbf{p}_{n,c}$ denotes the probability of class c. Then, we focus on the *cross entropy loss*

$$-\sum_{n=0}^{N}\sum_{c=0}^{C} \mathbb{1}(c = y_n) \log(\mathbf{p}_{n,c})$$

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

(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`.

Solution: In the constructor:

```
self.linear = torch.nn.Linear(784, 3)
```

In the forward function:

```
    outputs = self.linear(x)
```

(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]

Solution:

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

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

(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 ...
...
```

Solution:

```
    optimizer.zero_grad()
    outputs = model(batch_X)
    loss = criterion(outputs, 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 $\mathbf{p}_n = \sigma(\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top \mathbf{x}_n))$, where $\sigma(.)$ is the element-wise sigmoid function. Again, we focus on the *cross entropy loss*, hence the network will impelement $\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top \mathbf{x}_n)$ (note the outer sigmoid will be taken care of implicitly in our loss).

(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]

Solution: In the constructor:

```
self.hidden = nn.Linear(784, 400)
self.output = nn.Linear(400, 3)
```

In the forward function:

```
hidden = self.hidden(x)
hidden = torch.sigmoid(hidden)
outputs = self.output(hidden)
```

(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]

Solution: Code:

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

Performance Comparison [16 pts]

(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]

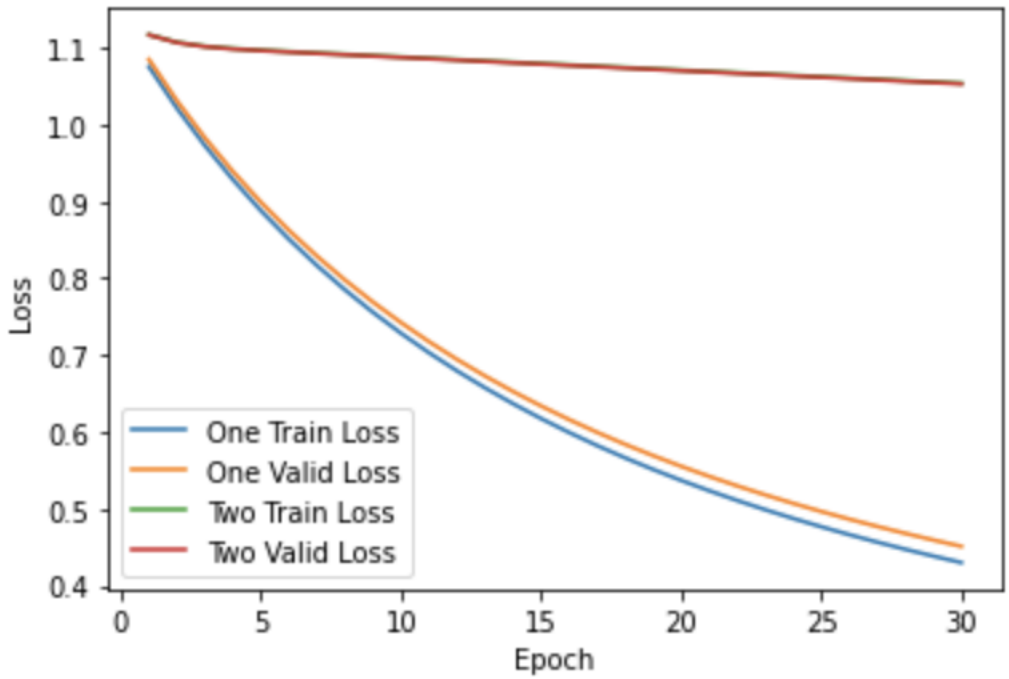
Solution:

Code:

```
epochs = [*range(1,31)]

plt.figure()
plt.plot(epochs, one_train_loss, label="One Train Loss")
plt.plot(epochs, one_valid_loss, label="One Valid Loss")
plt.plot(epochs, two_train_loss, label="Two Train Loss")
plt.plot(epochs, two_valid_loss, label="Two Valid Loss")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

Plot:



We can observe the loss of the two layered model for both the training and validation set is decreasing at a far lower rate than the one layered network. The slow learning is because in the 2 layered network the error gradient is back-propagated through the network with each adjustment. Because back-propagation uses the chain rule, we can multiply our adjustment by a very small derivative, barely changing our weights in the initial layers.

(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]

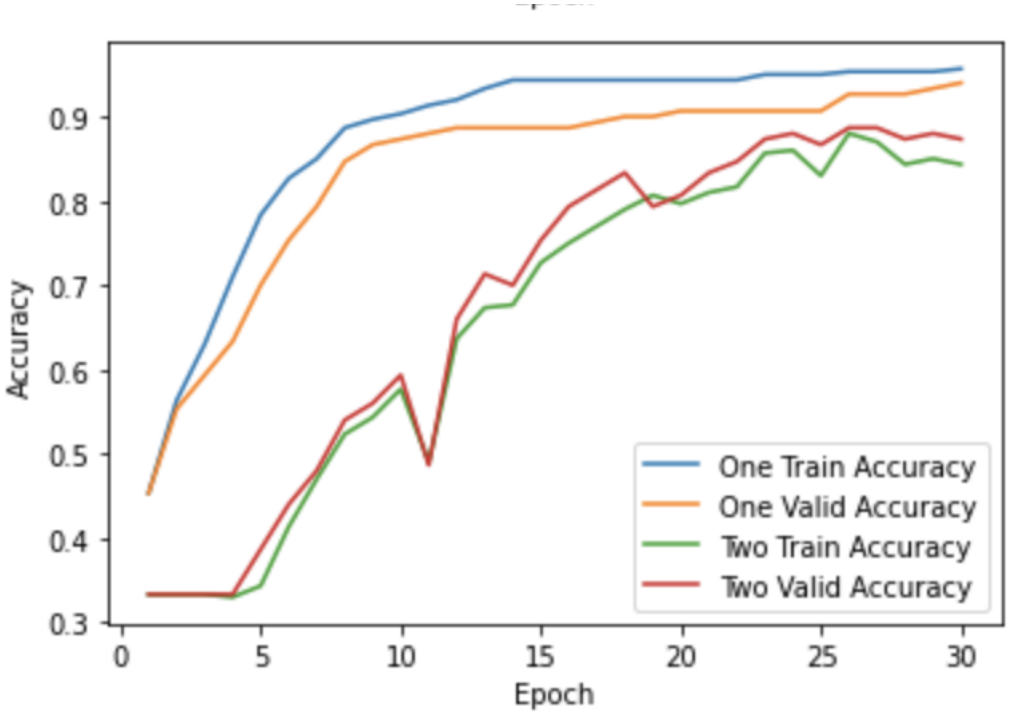
Solution:

Code:

```
epochs = [*range(1,31)]

    plt.figure()
    plt.plot(epochs, one_train_acc, label="One Train Accuracy")
    plt.plot(epochs, one_valid_acc, label="One Valid Accuracy")
    plt.plot(epochs, two_train_acc, label="Two Train Accuracy")
    plt.plot(epochs, two_valid_acc, label="Two Valid Accuracy")
    plt.legend()
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.show()
```

Plot:



We can observe the accuracy of the two layered neural network grows far slower than the accuracy of the single layered neural network. This is due to the same reason the loss function decreases slower, there are more weights to change and back-propagating through the weights can lead to small derivatives, which adjust the weights slowly. However by the later epochs the ac curacies do get close between the different models. It is also interesting that the accuracy on the validation data set is always less than the accuracy on the training dataset, which shows slight potential overfitting. But overall the data generalizes very well because the difference is not drastic

(k) Calculate and report the test accuracy of both the one-layer network and the two-layer network. Explain why we get such results. [3 pts]

Solution: I found:

Model one has a test accuracy of: 0.9599999...

Model two has a test accuracy of: 0.8666666...

I believe the one-layered model ends up getting better results because it is far quicker to train. We gave the models a fixed number of epochs instead of stopping them at convergence upon a loss function. The deeper layered model needs more time to train compared to the single layered model.

Code:

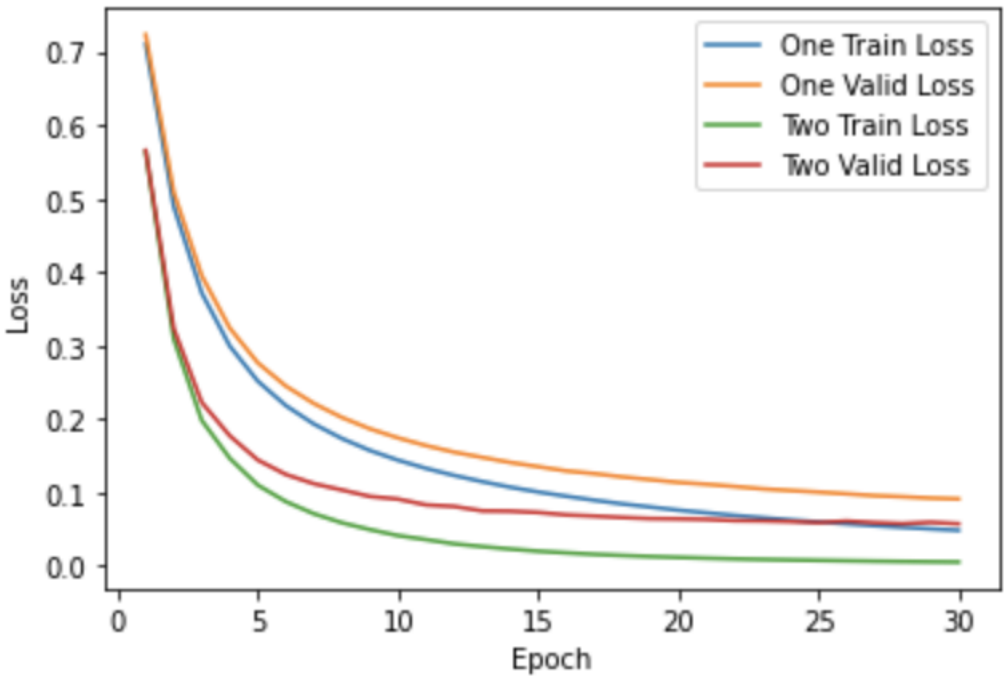
```
one_test_acc = evaluate_acc(model_one,test_loader)
two_test_acc = evaluate_acc(model_two,test_loader)
print("Model one has a test accuracy of: {}".format(one_test_acc))
print("Model two has a test accuracy of: {}".format(two_test_acc))
```

(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]

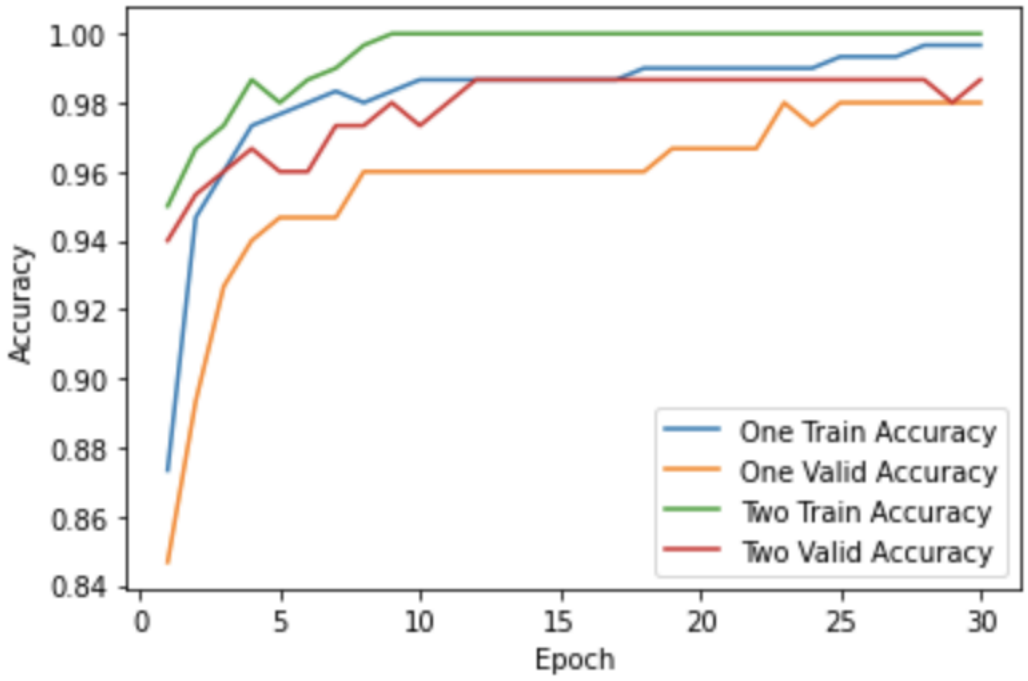
You can refer to https://pytorch.org/docs/stable/optim.html for more information about `torch.optim.Adam`.

Solution:

New loss plot:



We can observe from the new loss plot that the Adam optimizer was far more efficient in training both the single layered and double layered models. The single layered model "concaved" far quicker, showing that in the earlier epochs it updated far quicker with the SGD optimizer. These quicker drops in loss are due to Adam's dynamic learning rate, which allows the algorithm to adjust the weights earlier in the life cycle with greater magnitude than in the end. Amazingly the double layered neural network consistently showed a better loss rating at nearly every data point. The extra layer of abstraction provided by the hidden layer allows the model to better guess the targets. New accuracy plot:



The double layered neural network tested on the training data performed the best. This is not incredibly surprising, once you factor in Adam,'s dynamic learning rate to quickly speed up the growth of the model. The hidden layer of the second model allows further abstractions like edges, or curves to be learned from images, and provides better approximations. The second model performs better than the first model if you compare training tests, and validation tests together. However, there is an argument this improvement may be simply over fitting. This is supported by the accuracy on the test results:

Model one has a test accuracy of: 0.9666666388511658

Model two has a test accuracy of: 0.9666666388511658

We can see both models perform with identical accuracy on test data, which could mean that model two performing better in training is over fitting. It could also be a property of the test data, that some images are very similar to training data and easy to guess and some are not consistent with the rest, and are virtually impossible to guess.

Code:

```
optimizer = torch.optim.Adam(model_one.parameters(), 0.0005)
```

Submission instructions for programming problems

- Please export the notebook to a `.py` file by clicking the "File" → "Download.py" and upload to CCLE.

  Your code should be commented appropriately. The most important things:
  - Your name and the assignment number should be at the top of each file.

  - Each class and method should have an appropriate doctsring.
  - If anything is complicated, it should include some comments.

  There are many possible ways to approach the programming portion of this assignment, which makes code style and comments very important so that staff can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

- Please submit all the plots and the rest of the solutions (other than codes) to Gradescope