Deep Learning CS69000 Homework 2

Tunazzina Islam, islam32@purdue.edu

February 2020

Q0

1. Student interaction with other students / individuals:

   (c) No, I did not discuss the homework with anyone.

2. On using online resources:

   (b) I have used online resources to help me answer this question, but I came up with my own answers.

   Here is a list of the websites I have used in this homework:

   - pytorch forum discussion
   - https://towardsdatascience.com/understanding-dimensions-in-pytorch-6edf9972d3be
   - https://www.youtube.com/watch?v=q0pm3BrIUFo
   - https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/

Q1 (2.5 pts)

1. All layers of the neural network collapse into one with linear activation functions, no matter how many layers in the neural network, the last layer will be a linear function of the first layer because a linear combination of linear functions is still a linear function. So a linear activation function turns the neural network into just one layer. A neural network with a linear activation function is simply a linear regression model. Also it is not possible to use backpropagation to train the model because the derivative of the function is a constant, and has no relation to the input, X. So it's not possible to go back and understand which weights in the input neurons can provide a better prediction.

2. Yes. We add nonlinear activation function (ReLU) after the inner product layers to model non-linearity.

3. Yes. It's called Dying ReLU problem. A ReLU neuron is 'dead' if it's stuck in the negative side and always outputs 0. Because the slope of ReLU in the negative range is also 0, once a

neuron gets negative, it's unlikely for it to recover. Such neurons are not playing any role in discriminating the input and is essentially useless. Over the time it might happen that a large part of the network is doing nothing. The dying problem is likely to occur when learning rate is too high or there is a large negative bias.

4. Yes. Because unsupervised learning must learn $P(y, x)$, that is, both $P(x)$ and $P(y|x)$. Supervised learning needs to learn only $P(y|x)$. For example: Unsupervised learning can be used to pre-process the data. Usually, that means compressing it in some meaning preserving way like with PCA or SVD before feeding it to a deep neural net or another supervised learning algorithm.

5.
   - In Transfer learning, we will leverage the knowledge learned by a source task to help learning another target task. For example, a well-trained, rich image classification network could be leveraged for another image target related task. Basically, there are two basic scenarios for neural networks transfer learning: Feature Extraction and Fine Tuning. Transfer learning refers to the situation where what has been learned in one setting (i.e., distribution $P1$) is exploited to improve generalization in another setting (say distribution $P2$)
   - Multi-task learning, in which different supervised tasks like predicting $y^{(i)}$ given x) share the same input x, as well as some intermediate-level representation $h^{(shared)}$ capturing a common pool of factors. The main goal of multitask learning is to improve performance of a number of tasks simultaneously by optimizing all network parameters using samples from these tasks. For example, we would like to have one network that can classify an input face image as male or female, and at the same time can predict its age. The model can generally be divided into two kinds of parts and associated parameters:
     - Task-specific parameters (which only benefit from the examples of their task to achieve good generalization).
     - Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks).

   For the task of facial recognition to distinguish males from females. We will use transfer learning. As we have a very large dataset with pictures of dogs and cats, we may learn about one set of visual categories (sampled from $P1$), then learn about a different set of visual categories, such as males and females, in the second setting. If there is significantly more data in the first setting, then that may help to learn representations that are useful to quickly generalize from only very few examples drawn from $P2$. In our case, target dataset is small and target task is different from source task: Here, we will fine tune bottom, generic layers and remove higher, specific layers. In other words, we use feature extraction from early stages.

Warm-up: Implement Activations

We write activations functions: relu, softmax and stable_softmax. Softmax contains $exp()$ and cross-entropy contains $log()$, so this can happen: very large number $\rightarrow exp() \rightarrow inf \rightarrow log() \rightarrow NaN$. We use stable_softmax to avoid the overflow.

Warm-up: Understand Example Network

To run the Example Networks following is the correct command:

python hw2_training.py data -i torch.nn

Q2 (2 pts): Implement Feedforward Neural Network with Autograd

1. command line arguments for running this experiment with hw2_training.py:

   python hw2_training.py data -i torch.autograd

2. Analyze and compare each plot generated in the last step.

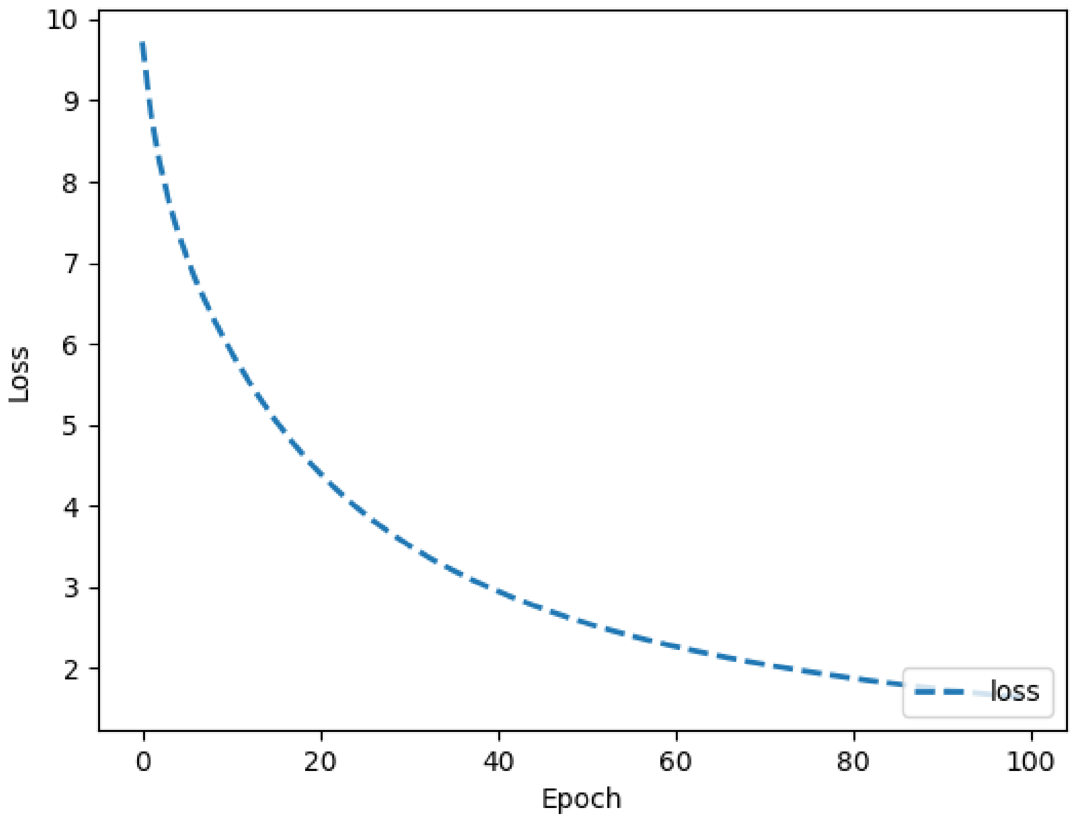

Figure 1: Loss vs. Epochs for AutogradNeuralNetwork.

In Fig. 1, loss is decreasing when the number of epochs increases. Until 20 epochs loss drops sharply, after that it drops gradually. Because training over the time helps to fit the data.

In Fig. 2, accuracy (both train and test) is increasing when the number of epochs increases. Here, both accuracy are almost similar though test accuracy started little bit higher than train accuracy after 50 epoch. For 100 epochs, we achieved approximately 59% test accuracy.

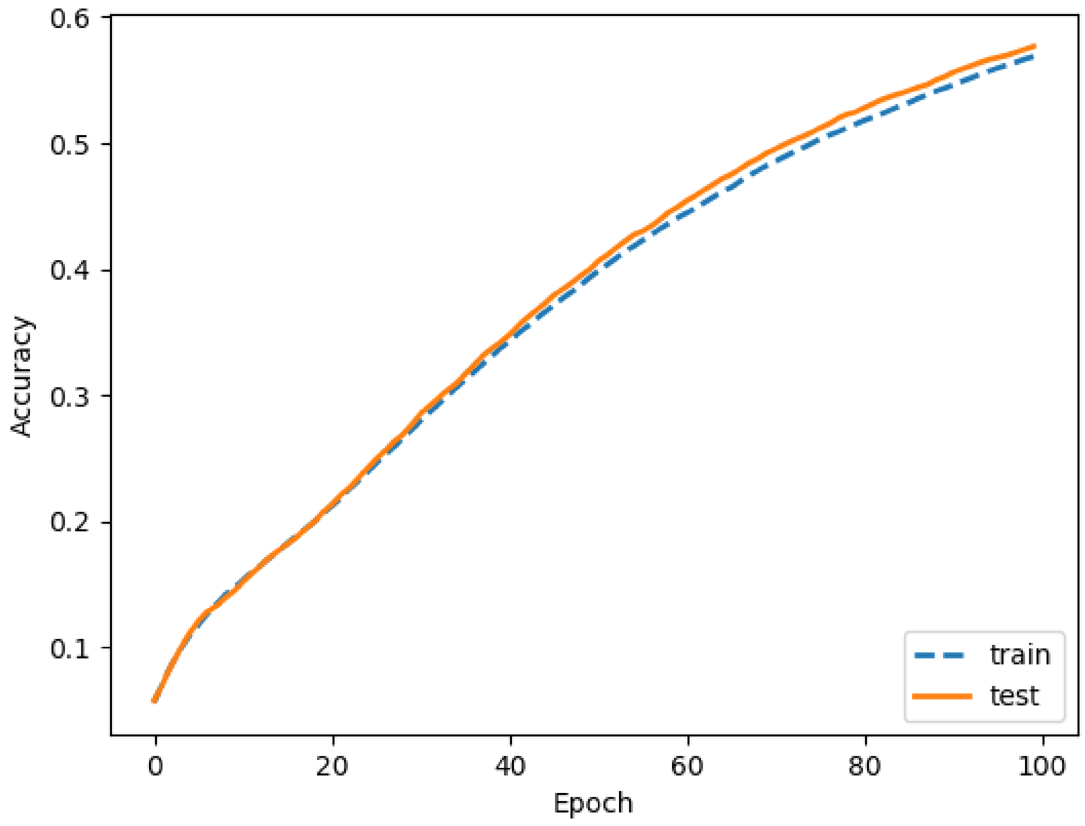

Figure 2: Accuracy vs. Epochs for AutogradNeuralNetwork.

Q3 (1 pts): Learning Curves: Deep vs Shallow

For plotting the learning curves, we use trainer file called **hw2_learning curves.py**. We have created a different python file for the network named **networks_lc.py**.

It is inside **my_neural_networks/networks_lc.py**. We imported this one (networks_lc) inside the executable file **hw2_learning_curves.py**. We added argument $-c$ in command line to choose network whether it is deep or shallow. $-c$ 1 for deep and $-c$ 0: shallow network.

We varied training data size ranged from 250 to 10000 with $step = 1000$.

1. command line arguments for running this experiment with **hw2_learning_curves.py**:

   command for deep network:

   python hw2_learning_curves.py data -i torch.autograd -c 1

   command for shallow network:

   python hw2_learning_curves.py data -i torch.autograd -c 0

2. The early stop strategy we have used is the absolute difference between previous loss and current loss is < 0.005 for each epoch. We used the number of maximum epoch, $max\_epochs = 500$.

3. Analyze and compare each plot generated in the last step.

   In Fig. 3 for Deep AutogradNeuralNetwork, when the train dataset size is small (250) out network can easily memorize the data so accuracy of train is the highest (around 98%) but network can't perform well on test data (accuracy 51%). May be network could learn little bit better over the time but it will create overfitting. So our early stop strategy (threshold = abs(loss from previous epoch - loss from current epoch) < 0.005) helps to avoid overfitting. We gradually increases our train dataset size upto 10000 (where step = 1000). When dataset size is 1250 train accuracy drops sharply to 76% and train accuracy rises sharply to 66%. We can see, train accuracy started to decrease slowly with the increase of dataset size and test accuracy became almost stable. Finally, we get train accuracy around 69% and test accuracy approx. 67%. There is a gap between blue line (train) and orange line (test) because of variance.

   For shallow network (Fig. 4), accuracy for both train and test increases sharply for first 3 train dataset size. After that both are gradually increasing (most of the cases) with training set size (until size = 8250). This shallow network can memorize output for corresponding input for smaller size train data and creates overfitting. Finally, we get train accuracy around 79% and test accuracy approx. 76% and both of them are higher than the accuracy we achieved in deep network.

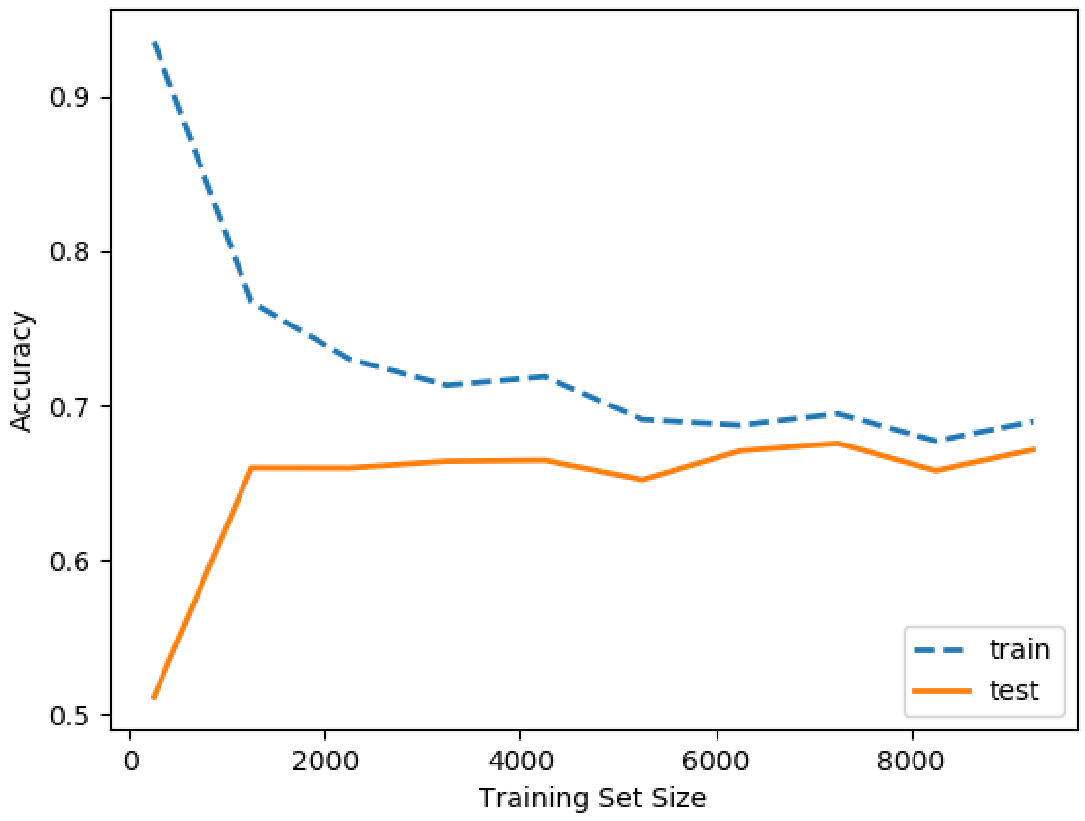

Figure 3: Learning curve for Deep AutogradNeuralNetwork.

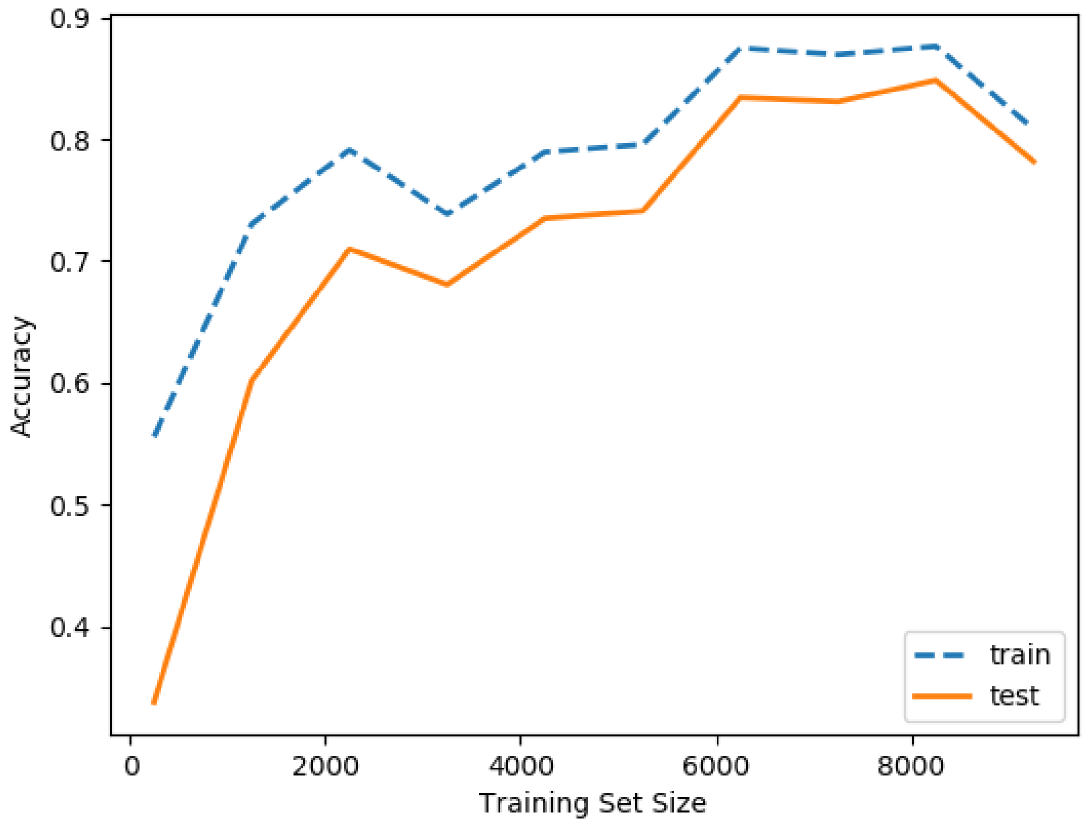

Figure 4: Learning curve for Shallow AutogradNeuralNetwork.

Q4 (4.5 pts): Implement Backpropagation from Scratch

1. command line arguments for running this experiment with hw2_training.py:

   python hw2_training.py data

2. Mathematical formulas used in the backpropagation implementation:

   Cross Entropy Loss,

   $$L = -\sum_{j=1}^{n} t_j \log a_j,$$

   where t_j is its true label; a_j is the propability predicted by the network which is a softmax result for class j.

   $$a_i = softmax(z_j) = \frac{\exp(z_i)}{\sum_{j=1}^{n}\exp(z_j)}.$$

   We combine computation of the partial derivative of the cross-entropy loss function w.r.t output activation (i.e. softmax) together with the partial derivative of the output activation w.r.t. z_j which results in a short and clear implementation. The partial derivative will be following:

   $$\begin{aligned}\sum_{k\in neurons}\frac{\partial L}{\partial a_k}.\frac{\partial a_k}{\partial z_j} = \sum_{k\in neurons;k\neq j} a_j t_k - t_j(1-a_j) = -t_j + t_j a_j + a_j \sum_{k\in neurons;k\neq j} t_k \\ = -t_k + a_j(t_j + \sum_{k\in neurons;k\neq j} t_k) = a_j - t_j\end{aligned}$$

   For our case, we will represent it as $(\hat{y}_i - y_i)$

   The derivative of the last layer parameters are

   $$\frac{\partial L(i)}{\partial W^{(3)}} = \frac{\partial z^{(3)}(i)}{\partial W^{(3)}}\frac{\partial L}{\partial z^{(3)}} = h^{(2)}(\hat{y}_i - y_i)$$

   because

   $$\frac{\partial z^{(3)}(i)}{\partial W^{(3)}} = \frac{\partial(h^{(2)}.W^{(3)} + b^{(3)})}{\partial W^{(3)}} = h^{(2)}$$

   and

   $$\frac{\partial L(i)}{\partial b^{(3)}} = \frac{\partial z^{(3)}(i)}{\partial b^{(3)}}\frac{\partial L(i)}{\partial z^{(3)}} = (\hat{y}_i - y_i),$$

   as

   $$\frac{\partial z^{(3)}(i)}{\partial b^{(3)}} = 1.$$

   The derivatives with respect to the hidden values $h^{(2)}$ of the previous layer are

   $$\frac{\partial L(i)}{\partial h^{(2)}} = \frac{\partial z^{(3)}(i)}{\partial h^{(2)}}\frac{\partial L(i)}{\partial z^{(3)}} = W^{(3)}(\hat{y}_i - y_i)$$

The derivatives with respect to the $W^{(2)}$ of the previous layer are

$$\frac{\partial L(i)}{\partial W^{(2)}} = \frac{\partial z^{(2)}(i)}{\partial W^{(2)}}\frac{\partial h^{(2)}}{\partial z^{(2)}}\frac{\partial L(i)}{\partial h^{(2)}} = \frac{\partial(h^{(1)}.W^{(2)}+b^{(2)})}{\partial W^{(2)}}.\frac{\partial h^{(2)}}{\partial z^{(2)}}*W^{(3)}(\hat{y}_i-y_i) = h^{(1)}.\frac{\partial h^{(2)}}{\partial z^{(2)}}*W^{(3)}(\hat{y}_i-y_i)$$

The derivatives with respect to the hidden values $h^{(1)}$ of the previous layer are

$$\frac{\partial L(i)}{\partial h^{(1)}} = \frac{\partial z^{(2)}(i)}{\partial h^{(1)}}\frac{\partial h^{(2)}}{\partial z^{(2)}}\frac{\partial L(i)}{\partial h^{(2)}} = \frac{\partial(h^{(1)}.W^{(2)}+b^{(2)})}{\partial h^{(1)}}.\frac{\partial h^{(2)}}{\partial z^{(2)}}*\frac{\partial L(i)}{\partial h^{(2)}} = W^{(2)}.\frac{\partial h^{(2)}}{\partial z^{(2)}}*W^{(3)}(\hat{y}_i-y_i)$$

and bias,

$$\frac{\partial z^{(2)}(i)}{\partial b^{(2)}} = 1,$$

and the derivative of the ReLU is

$$\frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} = \begin{cases} 1 & \text{, if } \mathbf{z}^{(2)}(i) > 0 \\ 0 & \text{otherwise.} \end{cases}.$$

We then backpropagate $\partial L(i)/\partial h^{(1)}$ to the previous layer. This will be needed in the final derivative the final derivatives are

$$\frac{\partial L(i)}{\partial W^{(1)}} = \frac{\partial z^{(1)}(i)}{\partial W^{(1)}}\frac{\partial h^{(1)}(i)}{\partial z^{(1)}}\frac{\partial L(i)}{\partial h^{(1)}},$$

and

$$\frac{\partial L(i)}{\partial b^{(1)}} = \frac{\partial z^{(1)}(i)}{\partial b^{(1)}}\frac{\partial h^{(1)}(i)}{\partial z^{(1)}}\frac{\partial L(i)}{\partial h^{(1)}},$$

where

$$\frac{\partial z^{(1)}(i)}{\partial W^{(1)}} = x_i,$$

and

$$\frac{\partial z^{(1)}(i)}{\partial b^{(1)}} = 1,$$

and the derivative of the ReLU is

$$\frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{z}^{(1)}} = \begin{cases} 1 & \text{, if } \mathbf{z}^{(1)}(i) > 0 \\ 0 & \text{otherwise.} \end{cases}.$$

In total,

$$\frac{\partial L(i)}{\partial W^{(1)}} = \frac{\partial z^{(1)}(i)}{\partial W^{(1)}}\frac{\partial h^{(1)}(i)}{\partial z^{(1)}}\frac{\partial L(i)}{\partial h^{(1)}} = x_i.\frac{\partial h^{(1)}(i)}{\partial z^{(1)}}*W^{(2)}.\frac{\partial h^{(1)}(i)}{\partial z^{(1)}}*W^{(3)}(\hat{y}_i-y_i)$$

The final output are the derivatives of the parameters. This function outputs

$$\mathrm{d}W^{(1)} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L(i)}{\partial W^{(1)}},$$

$$\mathrm{d}b^{(1)} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L(i)}{\partial b^{(1)}},$$

$$\mathrm{d}W^{(2)} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L(i)}{\partial W^{(2)}},$$

$$\mathrm{d}b^{(2)} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L(i)}{\partial b^{(2)}},$$

$$\mathrm{d}W^{(3)} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L(i)}{\partial W^{(3)}},$$

and

$$\mathrm{d}b^{(3)} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L(i)}{\partial b^{(3)}}.$$

3. Analyze and compare each plot generated in the last step.

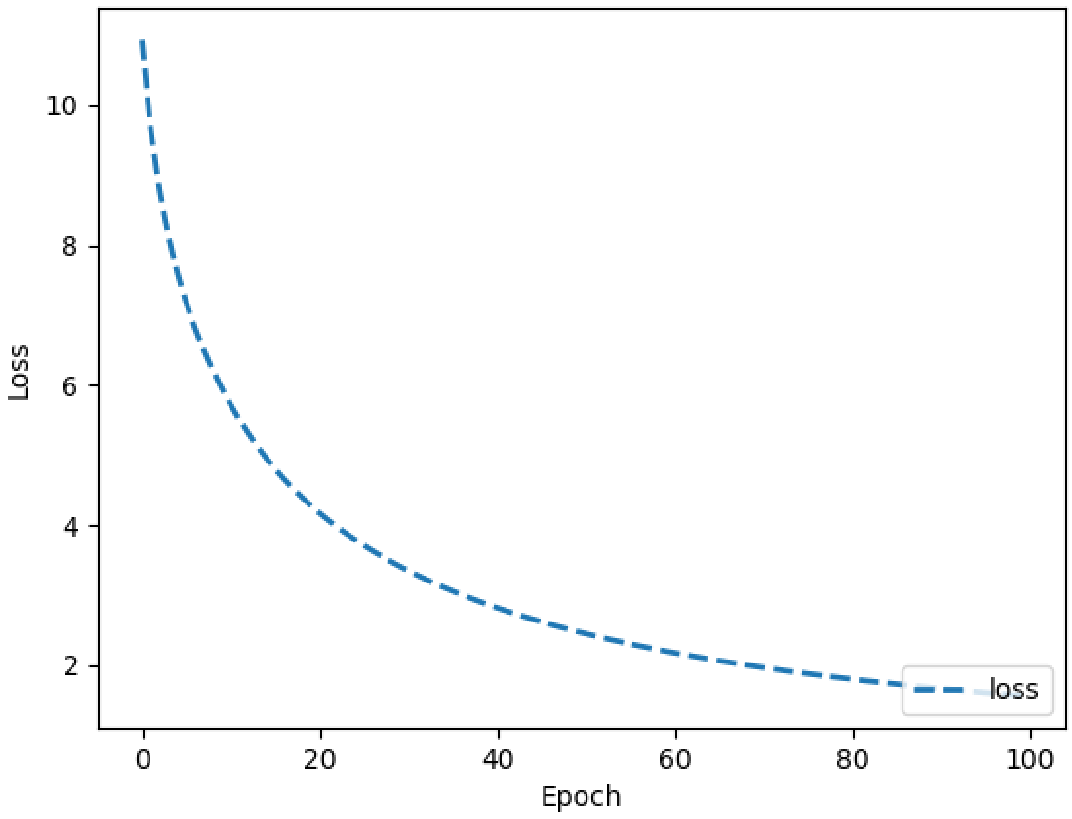

Figure 5: Loss vs. Epochs for BasicNeuralNetwork.

In Fig. 5 for BasicNeuralNetwork, loss is decreasing when the number of epochs increases. Until 20 epochs loss drops sharply, after that it drops smoothly. Because training over the time helps to fit the data.

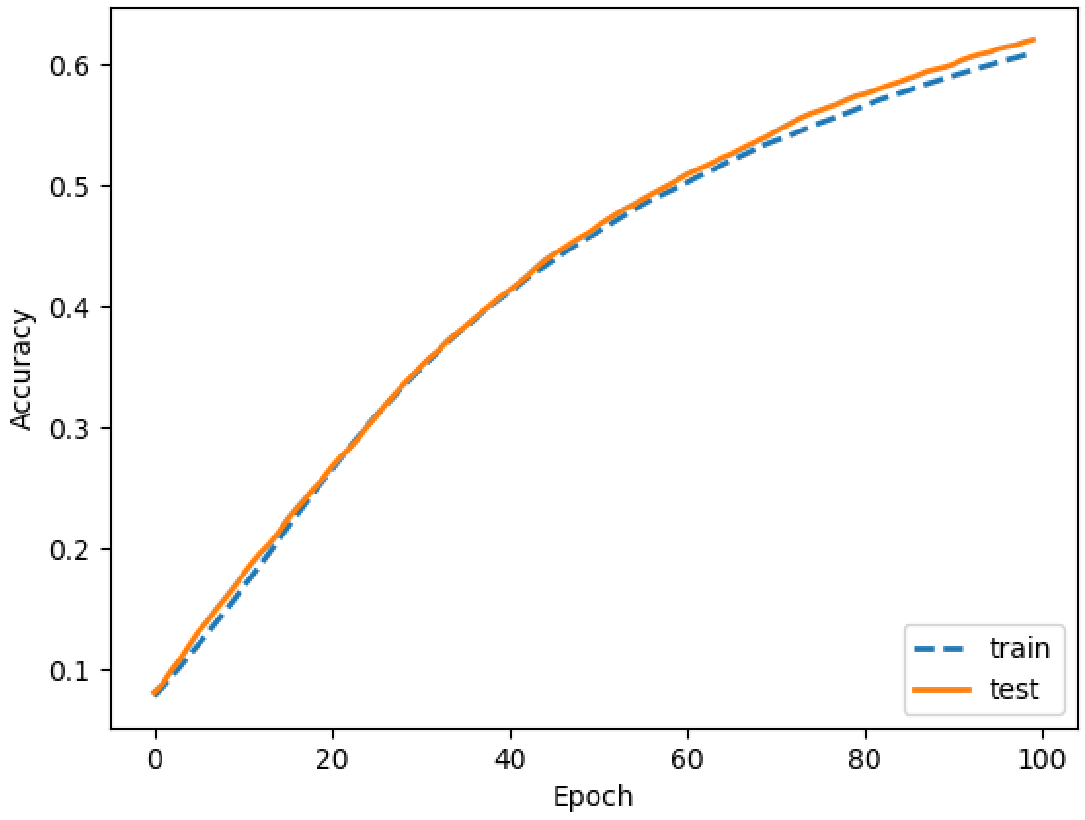

Figure 6: Accuracy vs. Epochs for BasicNeuralNetwork.

In Fig. 6, accuracy (both train and test) is increasing when the number of epochs increases. Here, both accuracy are almost similar though test accuracy started little bit higher than train accuracy after 70 epoch. For 100 epochs, we achieved approximately 61% test accuracy.

4. For plotting the learning curves, we use trainer file called **hw2_learning curves.py**. We have created a different python file for the network named **networks_lc.py**. It is inside **my_neural_networks/networks_lc.py**. We imported this one (networks_lc) inside the executable file **hw2_learning_curves.py**. We added argument $-c$ in command line to choose network whether it is deep or shallow. $-c$ 1 for deep and $-c$ 0: shallow network.

   We varied training data size ranged from 250 to 10000 with $step = 1000$.

   - command line arguments for running this experiment with hw2_learning_curves.py:

     command for deep network:

     python hw2_learning_curves.py data -c 1

     command for shallow network:

     python hw2_learning_curves.py data -c 0
   - The early stop strategy we have used is the absolute difference between previous loss and current loss is < 0.005 for each epoch. We used the number of maximum epoch, $max\_epochs = 500$.
   - Analyze and compare each plot generated in the last step.

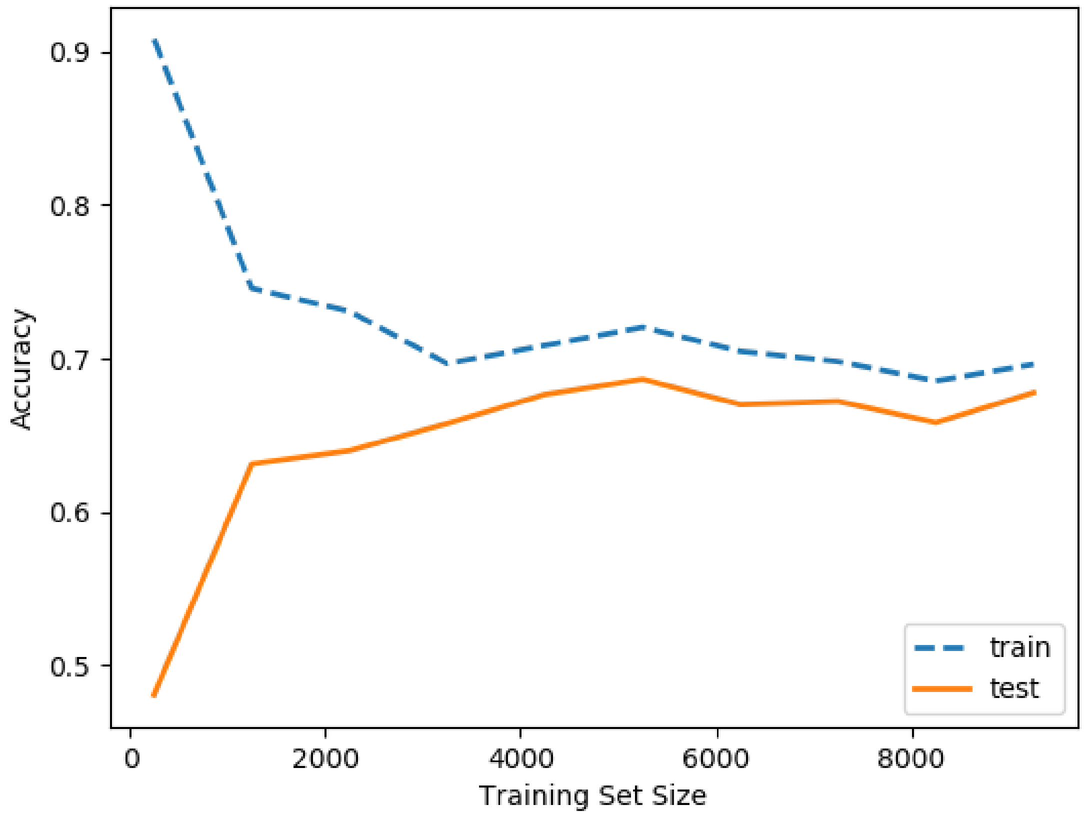

Figure 7: Learning curve for Deep BasicNeuralNetwork.

In Fig. 7 for Deep BasicNeuralNetwork, when the train dataset size is small (250) out network can easily memorize the data so accuracy of train is the highest (around 91%) but network can't perform well on test data (accuracy 41%). May be network could learn little bit better over the time but it will create overfitting. So our early stop strategy (threshold = abs(loss from previous epoch - loss from current epoch) < 0.005) helps to avoid overfitting. We gradually increases our train dataset size upto 10000 (where step = 1000). When dataset size is 1250 train accuracy drops sharply to 74% and train accuracy rises sharply to 63%. We can see, train accuracy started to decrease slowly with the increase of dataset size and test accuracy is also slowly increasing. Finally, we get train accuracy around 69% and test accuracy approx. 67%. There is a gap between blue line (train) and orange line (test) because of variance.

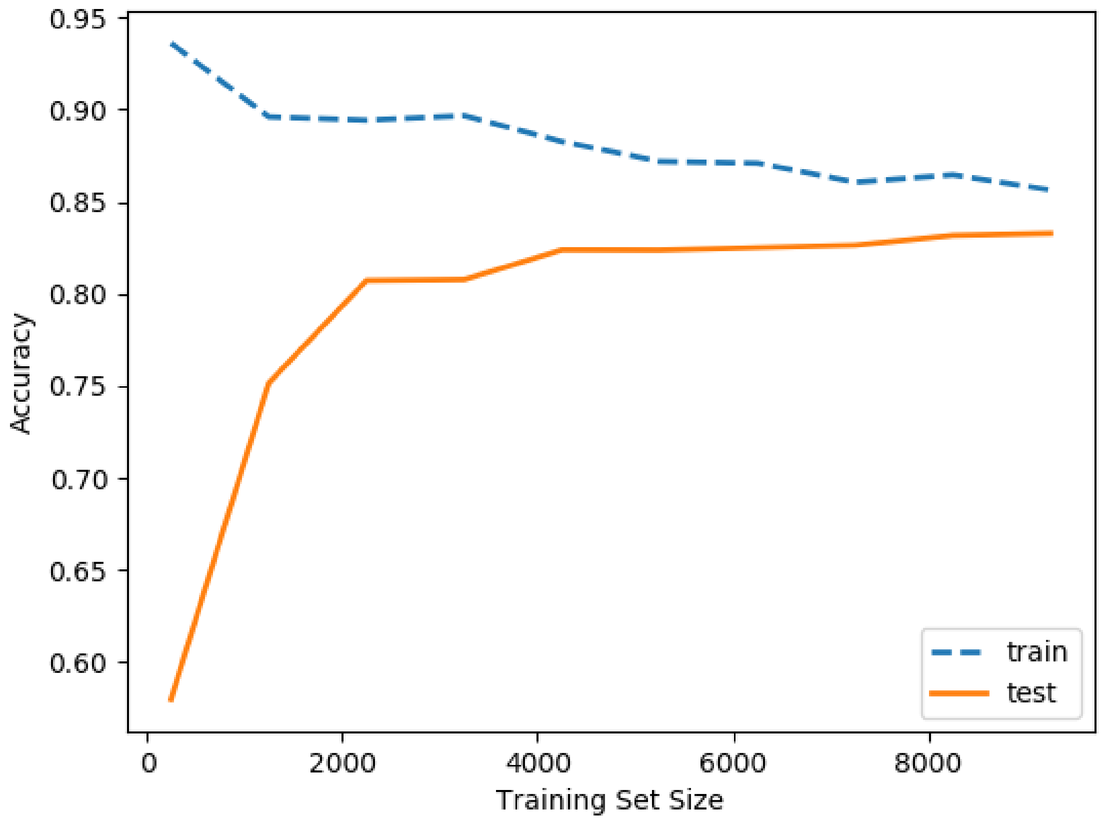

Figure 8: Learning curve for Shallow BasicNeuralNetwork.

For shallow network (Fig. 8), both train and test are overfitted. Test accuracy (80%) increases sharply for first 3 train dataset size. After that both are gradually increasing (most of the cases) with training set size (until size = 8250). This shallow network can memorize output for corresponding input for smaller size train data and creates overfitting. After train size 2000, we can motice that both train and test accuracy stay almost stable (train accuracy around 87% and test accuracy approx. 83%). Both of them are higher than the accuracy we achieved in deep network.