EE569 Digital Image Processing: Homework #5

Name: Nazim N Shaikh
USC Id: 8711456229
Email Id: nshaikh@usc.edu
Submission Date: 04/07/2019

CNN Training and Its Application to the MNIST Dataset:

(a) CNN Architecture and Training:

(Note: This section has no experimental results. Discussion have been included as part of Approach section)

Abstract and Motivation

- Convolutional Neural Networks (CNN) are a type of Neural Networks that have proven effective in applications such as image and video recognition, classification, etc. They are also used in object detection, face recognition and have become increasingly powerful in helping self-driving cars figure out surrounding environment.
- CNNs are somewhat similar to traditional neural networks in a way that they also contain a set of hidden layers with neurons which have learnable parameters – weights and biases. Inputs are passed to the neurons computing a weighted sum and then applying an activation function to produce output. Output is continuously optimized through use of loss function.
- However, CNNs differ from neural networks in a way that they take advantage of the fact that inputs are images. The input is a 3D volume with parameters: height, width and depth. Here, depth refers to number of channels in the input. For example, in case of MNIST, dimension of input volume is 28*28*1 (here, depth is 1, since images are grayscale/black and white).

Approach and Discussion

- In general, convolutional networks consists of sequence of layers (with or without parameters) which transforms 3D input volume layer by layer through some differentiable function at each layer to get final output classification scores.
- Most basic CNN architecture follows below pattern:

  INPUT -> [[CONV -> RELU] *N -> POOL?] *M -> [FC -> RELU]*K -> FC->SOFTMAX

  Here * indicates repetition of that particular layer for either N, M or K times

- Now, Let's discuss each of the components of convolutional network in detail

  1. CONVOLUTION LAYER – It is the fundamental building block of convolutional network. It consists of set of small learnable independent filters (also known as kernels) which are convolved with the input image. Each filter helps in learning/detecting different features (edge, curve, etc.) in the input image. Here, the filter is moved across width and height of the image and dot product is computed between input pixel and filter elements at respective positions. This results in a 2D output which represents response of filter at each position indicating whether some feature is present or not. If the feature is present (for example, edge), the output contains high value. In the absence of features, output will have low value at that particular position. Once the image is convolved will all the filters, 2D output is stacked along depth dimension resulting in an 3D output. The output is then passed through an activation function - either tanh or ReLU (will discuss more on this later) in order to introduce non-linearity in the network

     Furthermore, the size of output volume depends on 3 parameters: padding, stride and depth which are usually referred to as Hyperparameters

     a. Padding: Represents amount of padding to be done for input image. A padding of 1 means, one pixel is added all around the edges. This allows to control the size of output volume. The padding is used to overcome 2 issues:
        1. Image shrinking caused by convolution operation
        2. Information lost from corner pixels as they are only used few times during convolution.

        In general, there are 2 options for padding
        i. Same: results in an output which is of the same size as input.

j. Valid: means no padding is applied. Shrinked output

b. Stride: Represents number of steps to be taken while moving the filter across the image. For example, a stride of 2 means, we jump 2 pixels at a time when sliding the filter. Larger strides result in small output volumes spatially.

c. Depth: Represents number of filters to be used. Each filter will help in detecting various features in the input image.

d. Filter Size**:** Represents size of individual filters. Usually this is same for all filters being used in this layer across all channels.

. Above results in one layer of convolutional network. The general formula for output volume of this layer is given as below:

$$\left(\frac{n + 2p - f}{s} + 1\right) x \left(\frac{n + 2p - f}{s} + 1\right) x\, n_f$$

Where,

n is from input of shape $n\ x\ n\ x\ n_c$,

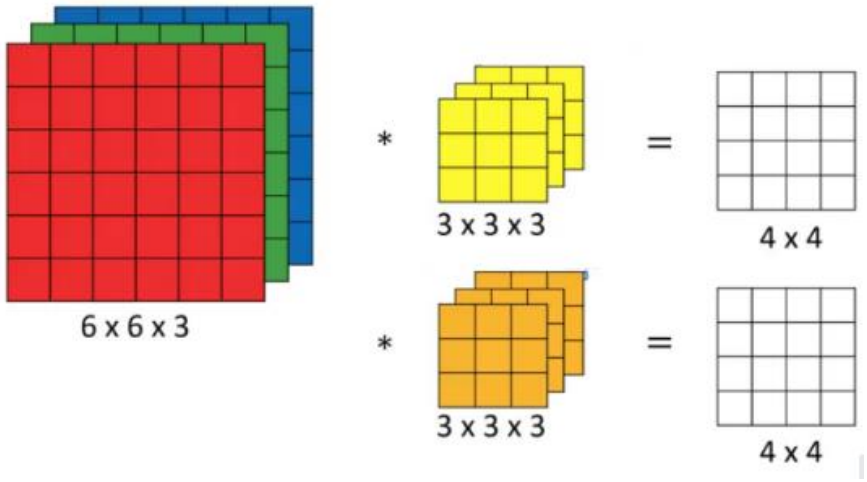
n_c is number of channels in the input and n_f represents number of filters,

f is from filter of size $f\ x\ f\ x\ n_c$,

p represents amount of padding,

s represents striding size.

Below is a small illustration of output of convolutional layer:

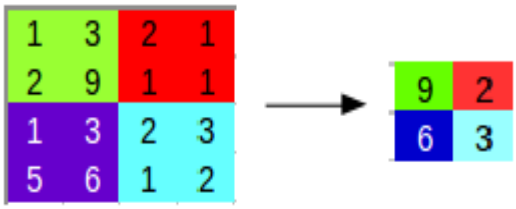


As per above example, input volume has size 6*6*1 with 2 filters, each of size 3*3, then the output volume will have size 4*4*2 as two 4*4 filters are stacked along depth dimension.

The number of parameters for this layer depend upon filter size and is independent of input image size. For above mentioned example, the total number of parameters will be 2*(3*3*3+1) = 56. Here, one is bias term for each filter.
Generally, total number of parameters for this layer will be:

$$n_f * (filter\_size + 1)$$

2. POOLING LAYER – It is another building block of convolutional network. It is used to reduce spatial size of input image. This helps in reducing number of parameters down the network (thereby controlling overfitting) and also helps speed up the computation. The most common form of pooling is Max pooling. Here, small filter of size say 2x2 (as shown in the image below) is moved across the image with a stride of 2 and max value is retrieved from each window. This results in a down sampled output which is of half the original input size.
   The Hyperparameters for this layer are:
   a. Filter size
   b. Stride
   c. Type of Pooling (max or average)



Sometime avg pooling is also used, where instead of max, we take average of numbers within the filter window.

The general formula for output volume of this layer is given as below:

$$(\frac{n-f}{s} + 1)\ x\ (\frac{n-f}{s} + 1)\ x\ n_f$$

In above example, input volume of size 4*4*1 is pooled with filter size 2, stride 2 into output volume of size 2*2*1

This layer does not have any parameters since it does fixed computation of input. Also, the input is not padded with any zeros

3. FULLY CONNECTED LAYER - This layer takes in an output volume from previous layer and converts into a single dimensional output. In this layer, every neuron from previous layer is connected to every neuron in this layer. The purpose of this layer is to use features obtained from previous layers to classify the input image into various classes. A SoftMax activation function is applied at the output of the Fully Connected Layer.

4. SOFTMAX FUNCTION: It takes a vector of arbitrary real-valued numbers and squashes it to a vector of values in the range (0,1) thereby calculating probability distribution that sum to 1. This function is applied element wise (but not independently) where each element represents a class. The output gives us class probabilities. The SoftMax function is computed as:

$$f(y)_i = \frac{e^{y_i}}{\sum_j^C e^{y_j}}$$

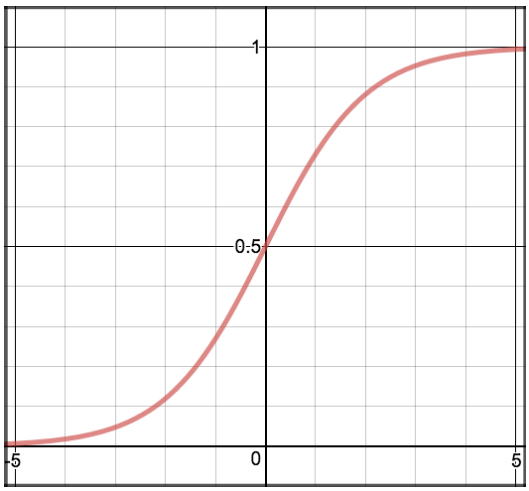
Where y_i represents scores for each class in C. The denominator of above equation shows that the SoftMax activation for a particular class depends on scores of all the classes. It is highly used in multi-classification task.

5. ACTIVATION FUNCTIONS: It is non-linear transformation that is applied to the input signal between the layers of neural network or at the output layer in order to learn the non-linearity which is present in the real-world data. Since it is computed as fixed function, it does not contain any learnable parameters.

Below is a list of different types of activation functions

i. Sigmoid:

➔ It takes a vector of real-valued numbers as input and outputs a vector of values in the range (0,1). The output does not sum up to 1 and is not zero centered.



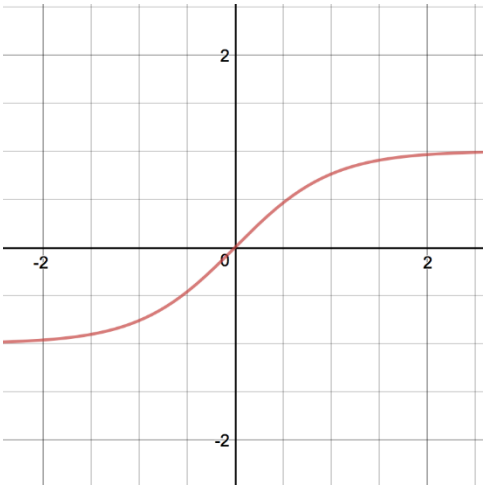
➔ Sigmoid functions are highly used in binary classification task.

➔ As seen from the graph, at either end, the y values change very less in response to changes in x. This makes the network to learn slowly leading in slower convergence.

➔ It is not used much since it gives rise to a problem of 'vanishing gradient'

ii. Tanh:

➔ It takes a vector of real-valued numbers as input and outputs a vector of values in the range (-1,1)

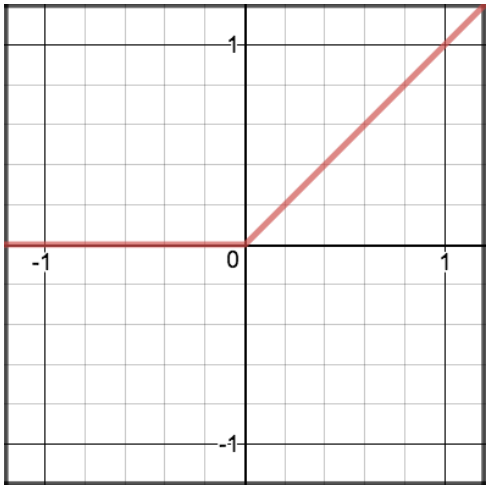


➔ Here, the output is zero centered and learning is faster compared to sigmoid.

➔ Tanh also faces the problem of 'Vanishing gradient'

iii. ReLU:

➔ Known as Rectified Linear Units, it is computed as max (0, x). It thresholds the input between 0 and x.



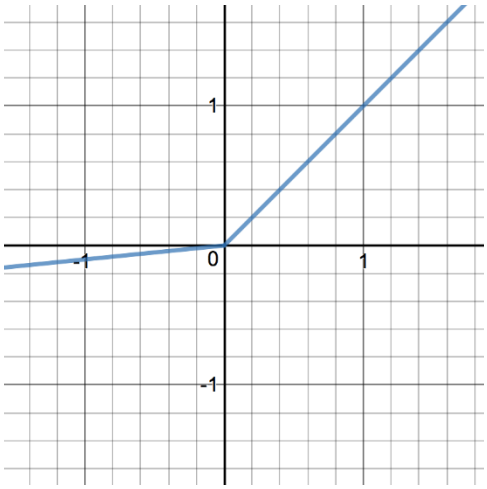
➔ The output is not zero centered. It converges faster and is less computationally expensive than sigmoid or tanh.

➔ ReLU solves the problem of vanishing gradient, however it suffers from something called 'Dying ReLUs'. This means, for activation in the region x<0, gradient becomes 0 as a result of which weight update won't happen, thereby stopping the neurons from responding to further variations in input.

➔ ReLU can also blow up the activate function, if the input is too large, since the range is [0, x)

iv. Leaky ReLU:

➔ It is a variant of ReLU which solves the problem of dying ReLU by allowing a small non-zero negative gradient in the region where x<0 as shown below



➔ The output is not zero-centered and converges in much faster than other activation functions.

- **What is the over-fitting issue in model learning? Explain any technique that has been used in CNN training to avoid the over-fitting.**
  - ➔ Over-fitting is a problem in which the model performs well on training set, however it fails to perform on validation/test set. This results in high variance, where model fails to generalize on new examples. Here, the training loss is much less compared to validation/test loss.
  - ➔ Overfitting can be reduced using many ways:
    1. Reduce architecture complexity
    2. Adding more data
    3. Performing data augmentation – randomly rotate, zoom, flip image, etc.
    4. Adding Regularization
    5. Adding dropout – randomly switching off few neurons
    6. Early Stopping the network
  - ➔ Here, we will discuss the most common one used in CNN which is called **Batch Normalization**.
    - ➢ We know that, normalizing input features with zero mean and unit standard deviation helps to speed up learning. However, as we go down the network, as and when the parameters are updated, each layer's input distribution (or distribution of hidden unit values) changes and we need to retrain the network to align the distribution. This increases training time. To avoid this, we apply normalization values in hidden layers for further improvement in training speed
    - ➢ Here, the input to hidden layers in the network is standardized by normalizing the output of previous activation layer by subtracting with batch mean and dividing by batch standard deviation. This makes the

hidden unit values more stable w.r.t changes in distribution. In this way, each layer of network also learns itself, little bit independent of other layers.

- Since each batch is scaled by mean/std dev, this adds noise within that batch. This is similar to dropout, as it adds noise to each hidden layer's activations. This results in regularization effect which reduces overfitting
- Advantage here is that, it prevents loss of information resulting from switching off the neurons as in case of dropout
- During test time, moving mean and variance is estimated using exponentially weighted average across mini-batches

- Why CNNs work better than traditional methods in many computer vision problems?
  - ➔ One of the challenges of computer vision problems is that the input data can get really big. For example, consider an image classification task with input image of the size 64 x 64 x 3 (here, 3 represents number of channels). The input feature dimension then becomes 12,288. If we have larger images (say, of size 1024 X 1024 X 3), then the dimension of input features will become ~ 3M. Now, if we pass such a big input to a traditional neural network, the number of parameters becomes too high (depending on the number of hidden layers and hidden units). With so many parameters, it's difficult to get enough data to prevent a neural network from overfitting. Additionally, the computational and memory requirements to train such a neural network becomes infeasible.
  - ➔ However, since many computer vision applications require us to work with large images, using convolution neural networks to becomes more efficient to work on such huge input data thereby greatly reducing the number of parameters in the network

- **Loss Function and Backpropagation:**

  - ➔ **Loss function** is an error metric which helps in evaluating how well the machine learning algorithm models input dataset. It is evaluated on model predictions. If the predicted output differs by large amount compared to real output, loss function will output a higher number and vice-versa.
  - ➔ Different types of loss functions are listed below:
    1. Mean Squared Error: It is average of squared difference between predicted output and real output. It is defined by below equation

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

It is widely used in linear regression problems to find optimum fitting line. The square terms provide large influences on MSE in cases where the error is large. So, it is affected by outliers

2. Mean Absolute Error: It is used to measure how close predictions are to the actual outcomes. It is computed by

$$MAE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n}$$

It is used in predictive modelling. Compared to MSE, MAE is robust to outliers as it does not contain square term

3. Cross Entropy Loss: It is used in binary classification. For multi-classification this loss is defined as below

$$\text{CE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where,

C = number of classes,

$\hat{y}_i$ and y_i are predicted and actual output

For binary classification, C = 2, then the cross-entropy loss becomes,

$$CE = -(y_i\, log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

It measures by how much 2 probability distributions differs. Large C.E loss, means difference between 2 distribution is large and vice versa.

4. Negative log likelihood: It is computed by

$$L = -\frac{\sum_{i=1}^{n} \log(\hat{y}_i)}{n}$$

It is used when model outputs probability for each class.

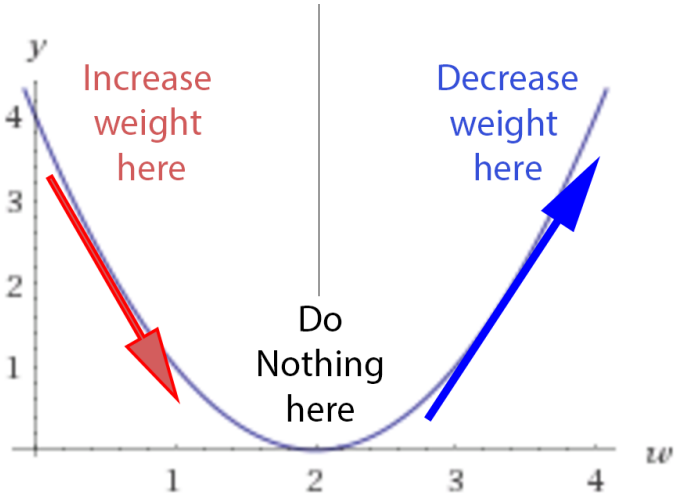
➔ **Backpropagation** is a way of optimizing loss function by computing its derivative with respect to weights or biases in the network and then updating the weights such that the neural network is able to perform any task be it classification or recognition with minimum error.

- We start by calculating the output error through loss function
- Find the gradient of loss function/error with respect to weights in the network

- Backpropagate the error from end to the start of neural network using chain rule
- Perform weight update at each layer using gradient descent as below

$$New\_weight = old\_weight - learning\_rate\ x\ derivative$$

Here, learning rate is a small constant introduced in order to force the weight update smoothly and slowly. This is usually used as hyperparameter of the model



  - If the derivative is positive, we need to decrease the weight else increase in weight will increase the error
  - If the derivative is negative, it means the new weight should be larger as increase in weight will increase the error
  - Once the derivative is 0, we reach the stable minimum and no further weight update is needed.

- In above, the gradient is computed for entire training dataset. However, there are different variants of gradient descents such as mini-batch gradient descent or stochastic gradient descent where gradient is computed gradient is computed for every small subset of training samples
- Weight updates can be done using various other methods such as, RMSProp, Adam, Adagrad, etc. These are also called as optimizers. Most commonly used is Adam Optimizer.

- This entire procedure is an iterative process where the update is performed again after going through whole training dataset once (referred to as an epoch). The updated weights are then used to minimize loss function.

The Importance of backpropagation is that as network is continuously trained, it learns to recognize different characteristics of input data. Once training is done, when an altogether new input is presented, the network is optimized in such way that it outputs a positive response if the new input contains features that resembles the once learned during training process.

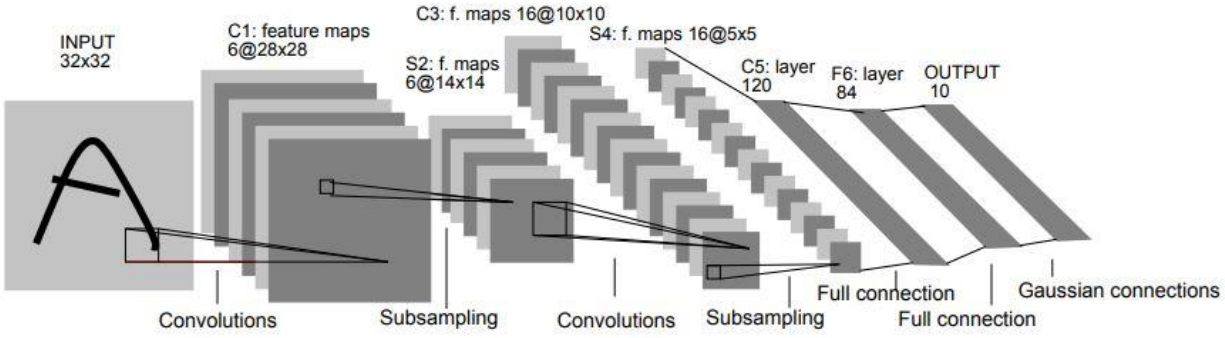
(b) Training LeNet-5 on MNIST Dataset:

Abstract and Motivation

- LeNet-5 is the first Convolutional based Neural Network proposed by Yann Lecun, Yoshua bengio, Leon Bottou and Patrick Haffner in 1990's for purpose of handwritten or machine printed character recognition.
- Now, it is being used as foundation for classifying or recognizing image of any categories – such as digits, cars, deer, airplane, ship, etc.
- Here, we will work on applying LeNet-5 network to classify handwritten digits (MNIST dataset)

Approach and Procedure

- Once we have understood the concept of convolutional neural networks, understanding the architecture of LeNet-5 is quite simple. It consists of 2 convolutional layers, each followed by pooling layers. The output of pooling layer is passed on to 3 fully connected network. The last FC layer is an output layer responsible for classification. The figure below shows the architecture of LeNet-5



- The input is 32x32 grayscale image. In our case, the input is MNIST grayscale images which are of size 28x28.
- The input is passed through first convolutional layer having 6 filters of size 5x5, stride as one and no padding.
- The LeNet then performs max pooling operation with a filter of size 2x2 and stride of 2 to down sample the image.
- The output of pooling layer is passed onto second convolutional layer with 16 filters having size 5x5, default of stride as one and no-padding.
- After this, second pooling operation is performed with filter of size 2x2 and stride 2 to further reduce the dimension
- The output of pooling layer is now passed to a set of fully connected networks having 120 and 80 neurons respectively.
- ReLU activation function is used at all conv and fc layers
- Finally, there is a fully connected SoftMax output layer with 10 neurons which computes probabilities for classifying digits from 0 - 9

- Entire training of LeNet5 on MNIST is done using Pytorch framework. The implementation is as follows:

1. Download the MNIST train and test data using torchvision library as below:

$$trainset = torchvision.datasets.MNIST(root = './data', train = True, download = True, transform = transform)$$

For testset, we set 'train' parameter=False

2. The dataset is then normalized with 0 mean and unit standard deviation using transforms function.

$$transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0,),(1,))])$$

3. Initialize parameters such as number of epochs, learning rate, batch size,etc.
4. Divide the dataset into above set batch size using DataLoader function as below:

$$trainloader = torch.utils.data.DataLoader(trainset, batch\_size = 16, shuffle = True, num\_workers = 1)$$

5. The lenet5 network is built as per mentioned in the question

$$\begin{aligned}&Net(\\&(conv1): Conv2d(1, 6, kernel\_size = (5, 5), stride = (1, 1))\\&(pool): MaxPool2d(kernel\_size = 2, stride = 2, padding = 0, dilation = 1, ceil\_mode = False)\\&(conv2): Conv2d(6, 16, kernel\_size = (5, 5), stride = (1, 1))\\&(fc1): Linear(in\_features = 256, out\_features = 120, bias = True)\\&(fc2): Linear(in\_features = 120, out\_features = 80, bias = True)\\&(fc3): Linear(in\_features = 80, out\_features = 10, bias = True)\\&)\end{aligned}$$

6. Initialize Loss and optimizer function

$$criterion = nn.CrossEntropyLoss()$$
$$optimizer = optim.Adam(net.parameters(), lr)$$

We use cross entropy loss function, since this is multi-class classification. Adam optimizer is used as it proven to give good convergence.
Here, lr is learning rate initialized at the start

7. We train the network as follows; for each epoch:
    - Take each batch of train data:
        - Perform feed forward operation
        - Calculate Loss
        - Perform backpropagation
        - Perform Weight update
    - After each epoch, perform predictions on train data and calculate train accuracy
    - After each epoch, perform predictions on test data and calculate test accuracy

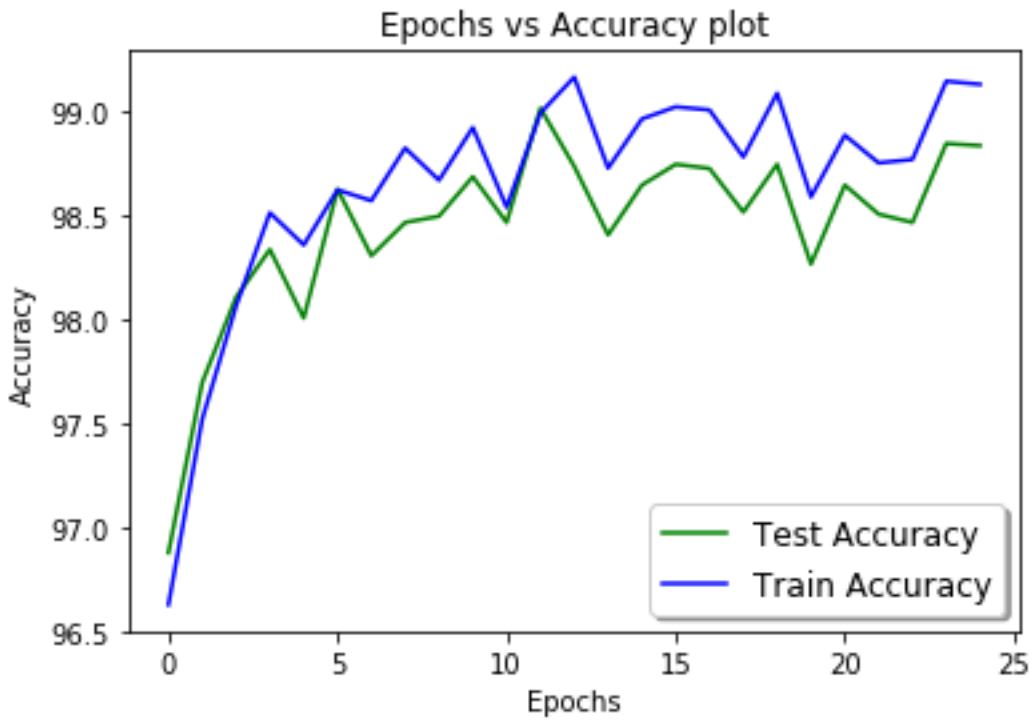
Note: 1 epoch is when we perform feed-forward, loss calculation, backpropagation and weight update on whole training data once.

8. To observe model performance over each epoch we plot train accuracy vs epochs and test accuracy vs epochs
9. Finally, Evaluate the model by performing predictions on test data using trained model and calculate final test accuracy

Experimental Results

1) **Set 1:**

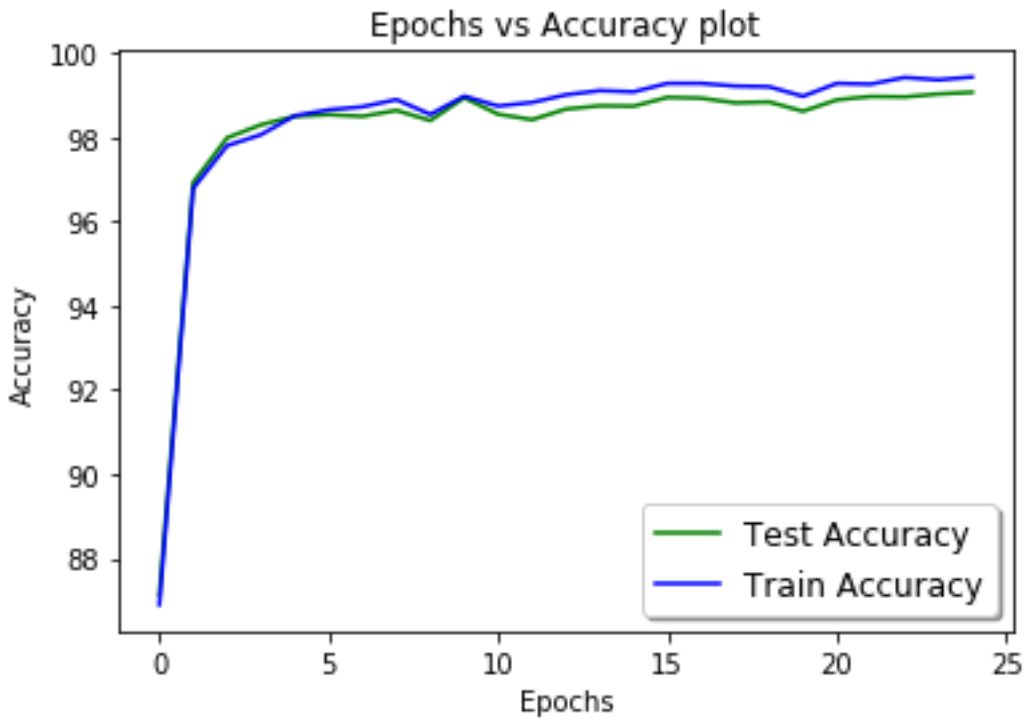
Epochs: 25
Learning Rate:0.001
Batch Size: 64
Xavier Initialization, Adam Optimizer




```
Final test accuracy of the network is: 98.84%
Final train accuracy of the network is: 99.13%
```

2) Set 2:

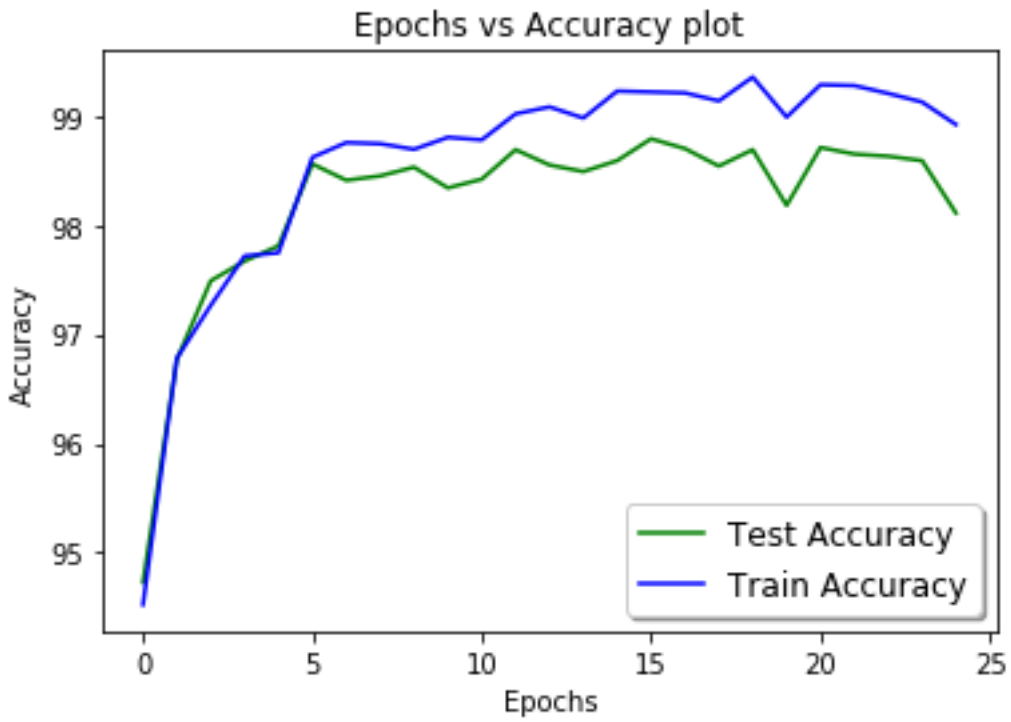
Epochs: 25
Learning Rate:0.001
Batch Size: 128
Xavier Initialization, Adam Optimizer




```
Final test accuracy of the network is: 98.11%
Final train accuracy of the network is: 98.92%
```

3) Set 3:

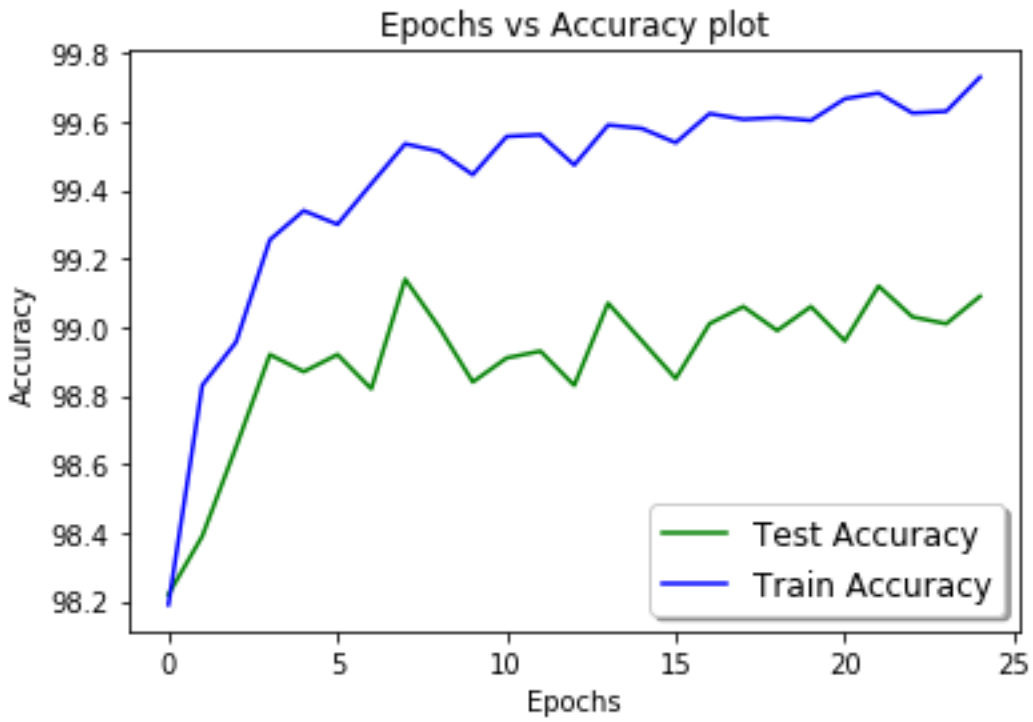
Epochs: 25
Learning Rate:0.0005
Batch Size: 128
Xavier Initialization, Adam Optimizer




```
Final test accuracy of the network is: 98.76%
Final train accuracy of the network is: 99.21%
```

4) Set 4:

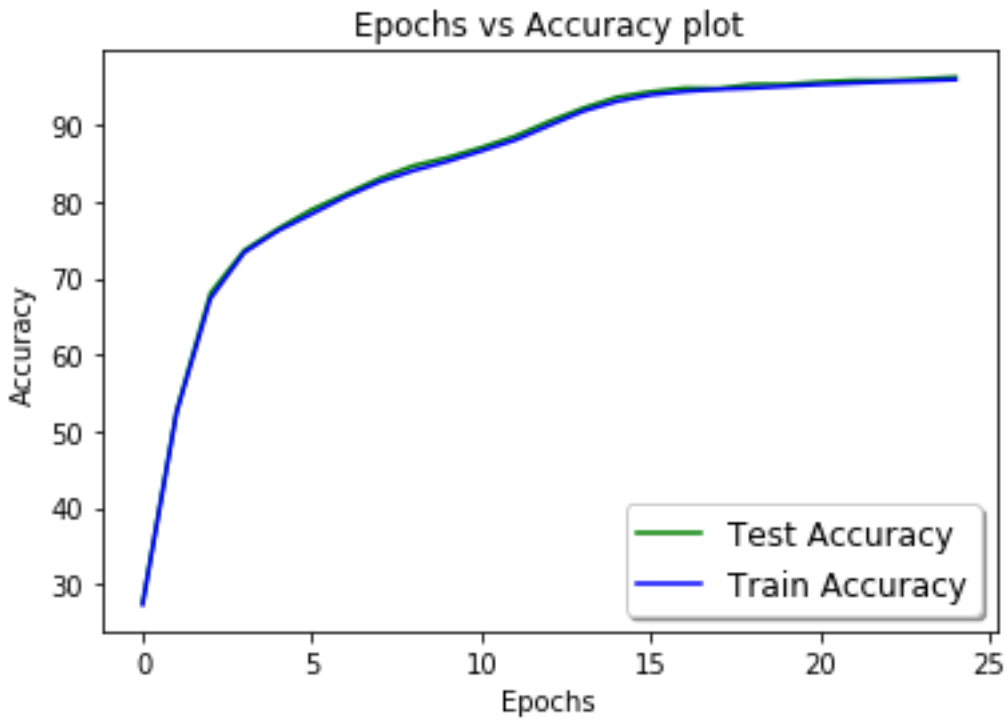
Epochs: 25
Learning Rate:0.001
Batch Size: 128
Xavier-Initialization, Batch Normalization, Adam Optimizer




```
Final test accuracy of the network is: 99.07%
Final train accuracy of the network is: 99.67%
```

5) Set 5:

Epochs: 25
Learning Rate:0.001
Batch Size: 64
Xavier Initialization, Batch Normalization, SGD Optimizer




```
Final test accuracy of the network is: 96.08%
Final train accuracy of the network is: 95.89%
```

Setting	Test accuracy (%)	Train accuracy (%)	Mean of train accuracy (%	Mean of test accuracy (%)	Variance of train accuracy	Variance of test accuracy
Set – 1	98.84	99.13	98.61	98.13	1.67	1.36
Set – 2	98.11	98.92				
Set – 3	98.76	99.21				
Set – 4	99.07	99.67				
Set – 5	95.89	96.08				

As seen from above table, Best accuracy is obtained in set 4 with,

```
Final test accuracy of the network as: 99.07%
Final train accuracy of the network as: 99.67%
```

Discussion

- The LeNet5 Network was trained on MNIST dataset using various settings. In almost all settings, learning rate was kept fixed at 0.001 and batch size as 128, since as per many research papers those are the optimal values. Additionally, the weights of the network were initialized using Xavier initialization since it is proven to give good results.
  - Xavier initialization is done to make sure the variance in input data weights remain same as the data moves from one layer to another so that the weights does not explode or vanish to zero.
  - From the table above, we see that, variance is less for both train and test. This indicates that over different settings, the accuracies change hardly by ~1.6%. Mean values represent the overall average accuracies over different settings. This means that for any of the above settings we can get at least test-train accuracy of ~98% and above
  - For sake of comparison among different settings, all the networks are trained for fixed 25 epochs.

1) In **Set-1**, the network was trained with learning rate of 0.001, batch size of 64 and Adam optimizer for weight update. From the plot, we see that train and test curves does not align so well, however there is steady increase at start with less difference in accuracies. Post 10^{th} epoch, the network seems to have learned well and oscillates between 98 - 99%. There doesn't seem to be any further improvement in the accuracy.

2) In **Set-2**, the network was trained with learning rate of 0.001, increased batch size of 128 and Adam optimizer for weight update. Batch size was increased to see how the network behaves as opposed to in set-1 with small size

   As seen from the plot, there aren't any erratic changes in the curve. Train and test accuracies aligns well and steadily. Initially, both train and test accuracies increase well reaching ~98%. Post that, the accuracy increase is slow, possibly because network as already learned most features at initial stages. Later it settles down to ~98.5% at the end of 25 epochs

3) In **Set-3**, the learning rate was reduced to 0.0005. The network was trained for a batch size of 128 with Adam optimizer for weight update. From the plot, we observe that, train – test accuracies do not align perfectly, but there seems to be somewhat steady increase in the accuracy with oscillations at later epochs. However, this does not seem to give any improvement in accuracy to previous to settings. Also, since there seems to be an oscillatory increase, it is possible to increase accuracy by training for more epochs.

4) In **Set-4**, The network was trained with learning rate of 0.001 for 25 epochs, batch size of 128 and Adam optimizer for weight update. Additionally, batch normalization was applied after each conv layer and fully connected layer for reasons explained in part (a) discussion. From plot, we see that, again there is an oscillatory increase at later epochs. However, one significant observation is that training curve is little higher compared to testing curve, with some difference in accuracies. This may be because batch normalization may have back-fired little allowing network to learn specific features well and causing a slight overfitting. This can be balanced out by adding dropout along with batch normalization.

   This setting resulted in the highest test accuracy of 99.07%

5) In **Set-5** Network was tested by training with learning rate of 0.001. However, here, SGD optimizer was used for weight update. From plot we see that the training and testing curve aligns perfectly, however, the training seems to be slow. Also, batch size chosen was 64 as this seems to provide a little faster convergence in case of SGD. However, there seems to be any significant improvement in accuracy compared to other settings. In fact, this setting gave lowest accuracy of all.

   Additionally, since this network was trained only for 25 epochs, from the plot, we can infer that increasing number of epochs, it is possible to achieve better results but at the expense of training time.

(c) Applying Trained network to negative images:

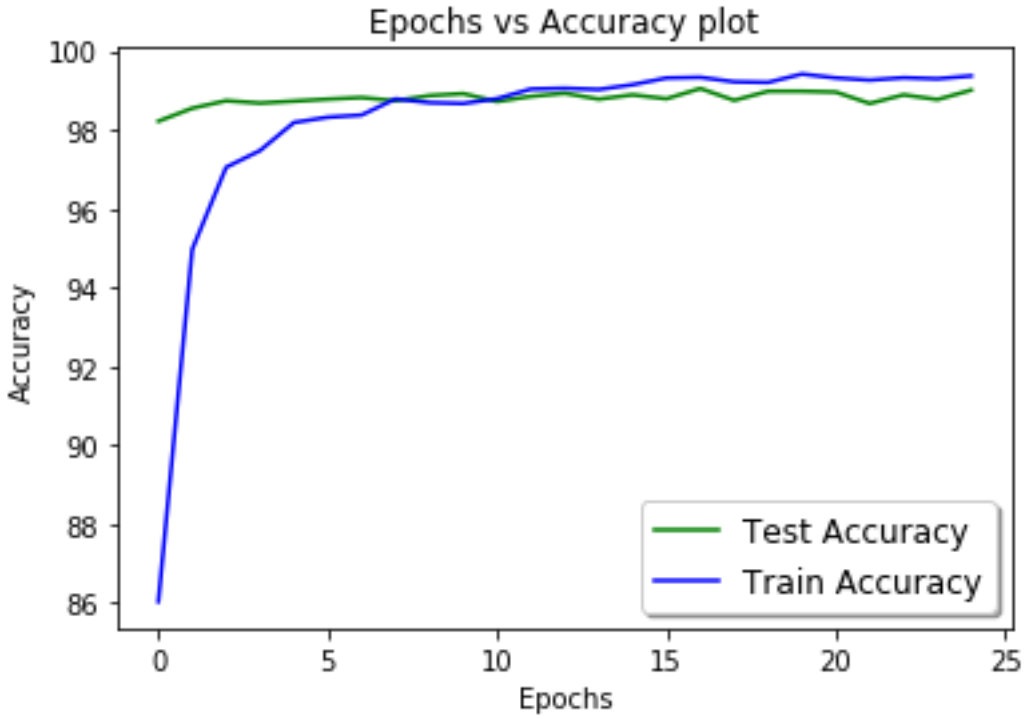
Abstract and Motivation

- As observed from part(b), LeNet5 achieves significantly good results on MNIST Dataset.
- However, to further evaluate how the network performs on various kinds of digit images, we try to test the network on a set of negative images.
- Additionally, we see how we can improve the network to handle such images.

Approach and procedure

- From part(b), we choose the network setting which gave best accuracy. In our case, it was set -4.
- Now, we negate all the test images and evaluate the network on these images.
- We get the final accuracy of the network on negative test images as: 46.06%-
- We see that, the network doesn't seem to generalize well. One way to improve the accuracy is to train the network on both original and negative images from MNIST dataset.
- We follow the same training procedure as mentioned in part (b). However, For every batch, we negate half of the train and test images and another half are kept as it is before passing it to network.
- The network is trained and evaluated in the same way as part (b)

Experimental Result



Final Test Accuracy of the network on positive test images: 98.79%

Final Test Accuracy of the network on negative test images is: 98.92%

Discussion

- From the final accuracy results, we observe that, there is drastic increase in the accuracy compared to results of part (b) on negative images and network is able to generalize well.

- From the accuracy vs epoch plot, there is gradual increase in the train accuracy at the start while the test accuracy seems to start at 98%.
- The network seems to have learn well and quickly at early, since we fed the network with shuffled set of negative and positive images for every batch. After 5th epoch, the accuracy becomes somewhat steady until it settles down to ~98.5% at the end of 25th epoch.

References

[1] http://cs231n.github.io/convolutional-networks/
[2] https://engmrk.com/lenet-5-a-classic-cnn-architecture/
[3] https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/
[4] https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c
[5] https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html
[6] https://blog.algorithmia.com/introduction-to-loss-functions/
[7] Paper on 'Batch-normalized Maxout Network in Network' by Jia-Ren Chang, Yong-Sheng-Chen
[8] https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e
[9] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py