

CS574: Assignment 1

Group 13

Members:

- 160101004 Abhinav Hinger
- 160101009 Abhishek Suryavanshi
- 160101013 Apurva N Saraogi
- 160101024 Daman Tekchandani
- 160101025 Divyam Agarwal

Summary

Major Libraries used: Pytorch(Architecture and training), Matplotlib(Plotting), Scikit-learn

The first process is understanding the data i.e MNIST. This is done by loading the dataset and create an iterable data loader that loads a Mini Batch of images. Then we start with a set of hyperparameters chosen by best practices. Instantiate the model, loss and optimizer and start the training. We monitor the training loss and test the trained model on the test set to detect underfitting/overfitting. We also use Confusion matrix to gain more insights into the model's behaviour.


Understanding the Data

There are 60000 Training images and 10000 Test images in the MNIST dataset each of 28x28.

Train data loader is a tensor of dimension: **torch.Size([100, 1, 28, 28])**

Here 100 is the Batch Size, 1 is Channels and 28x28 is the Image dimensions. Each pixel will be an input i.e 784. Each image is in grayscale.

Here is a sample of the dataset:



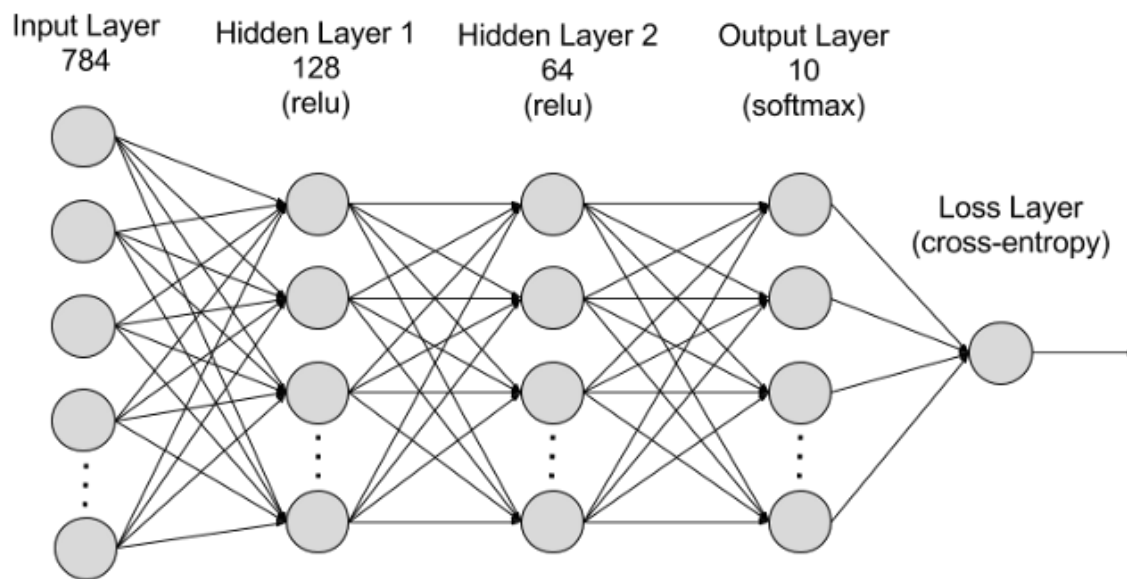
A 6x10 grid of handwritten digits from the MNIST dataset. The digits are as follows:

2	4	2	7	7	4	6	5	1	7
8	2	5	7	1	2	6	0	4	4
3	6	9	7	6	2	5	1	0	4
7	7	3	8	0	4	9	1	3	7
2	6	3	3	4	8	0	2	6	1
4	6	6	5	6	2	5	4	4	2

Building the Network: using torch.nn

We will build the network with an input layer (the first layer), an output layer of ten neurons (or units, the circles) and some hidden layers in between.

Eg.



We will experiment with the following Hyperparameters and observe the change in the learning process and test accuracy

1. Learning Rate
2. Number of Epochs
3. Number of Hidden Layers
4. Number of Hidden Units
5. Activation Functions (ReLU, Leaky ReLU, Softmax etc.)
6. Number of Convolution Kernels (CNN)
7. Size of Convolution Kernels (CNN)

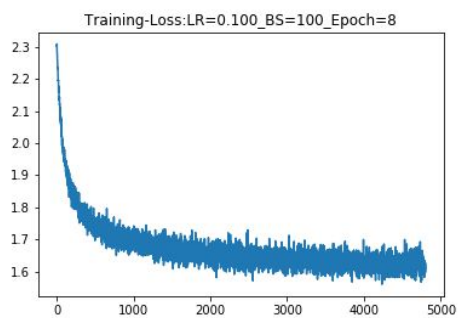
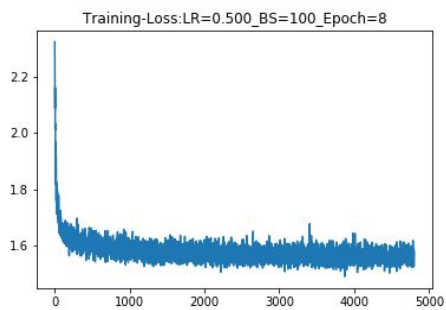
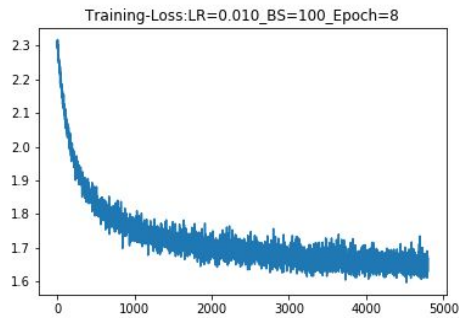
Logistic Regression

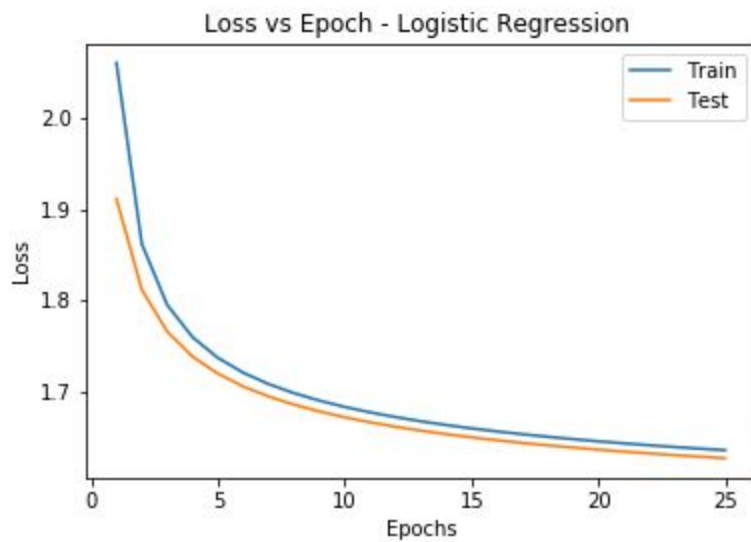
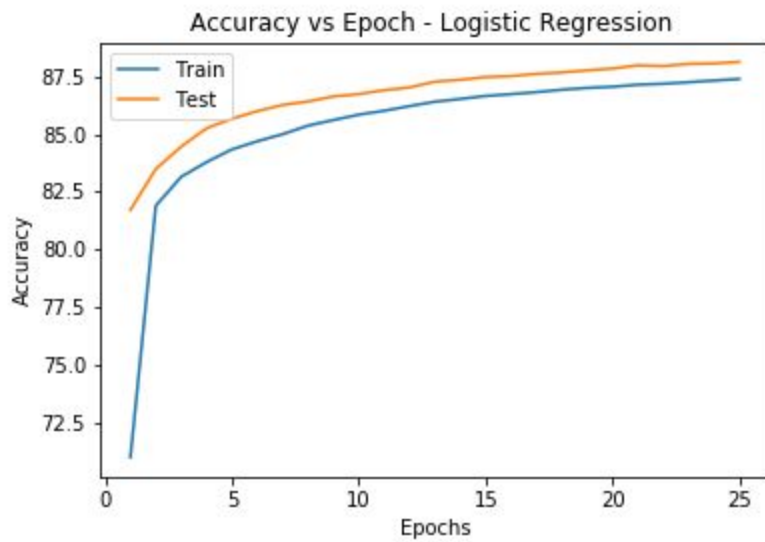
$y = \text{Sigmoid}(\mathbf{w}^T \mathbf{x} + \mathbf{b})$ where
x: Input vector (784 length)
w: Weights applied to the inputs
b: Bias

y: Output vector of size Number of classes to be classified into = 10

LR = 0.01, 0.05, 0.1 and 0.5

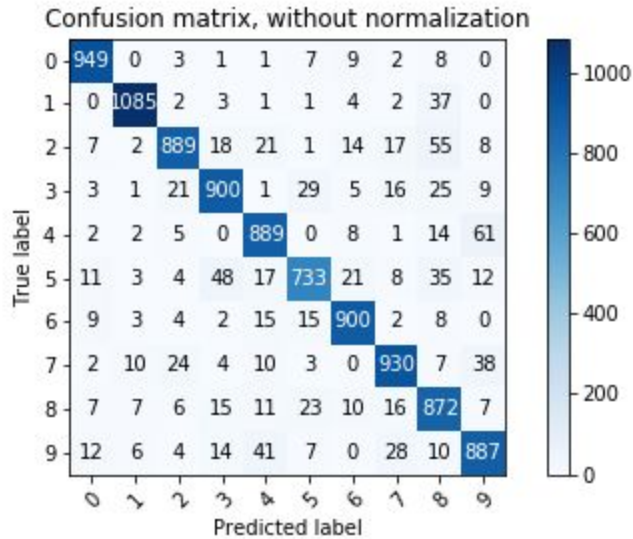
On increasing the learning rate the model converges faster. Initially accuracy on the test dataset increases but doesn't increase beyond 90 percent.





Epochs = 3 , 5 , 8 Keeping LR same:

3 Epochs give test accuracy 85 percent accuracy. On increasing the number of epochs, test accuracy increases indicating that model is **Underfitting**. On increasing epochs further models doesn't learn anything new (accuracy = 90%) and model also doesn't overfit because of simplicity of the model.

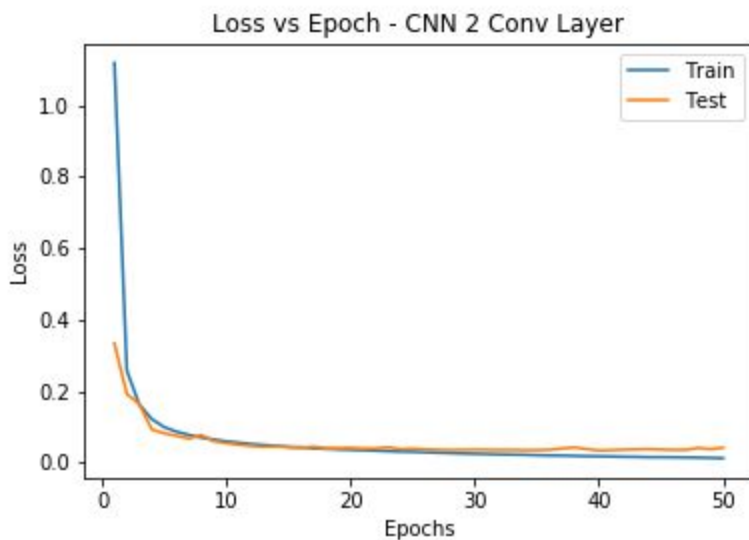
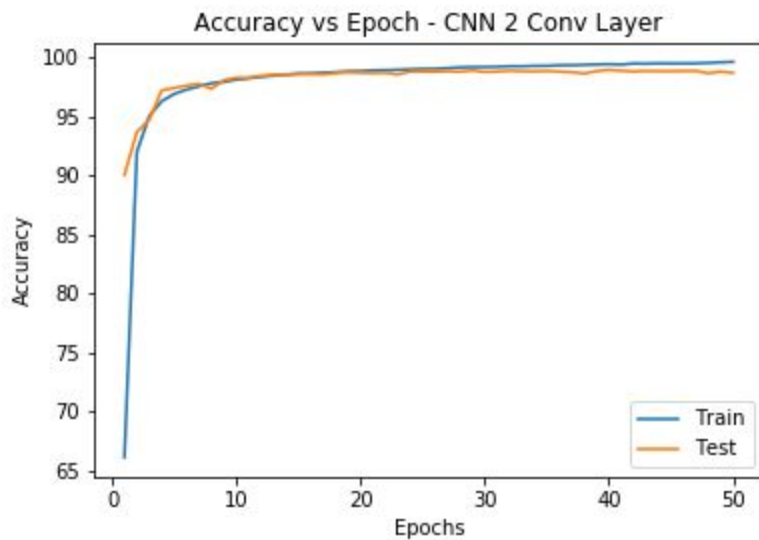


Confusion Matrix can be used to understand true/false positive/negative. This gives us an insight into where the model is going wrong. Eg Since 5 and 9 look really alike. Most 5 samples which are wrongly predicted are as 9.

CNN

CNN with MNIST

We used a Basic CNN with 2 Convolutional layers followed by Max Pooling and Activation function as ReLu. This is followed by 2 FC layers.



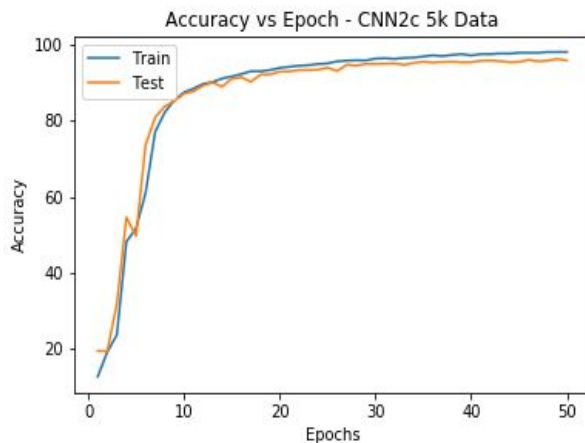
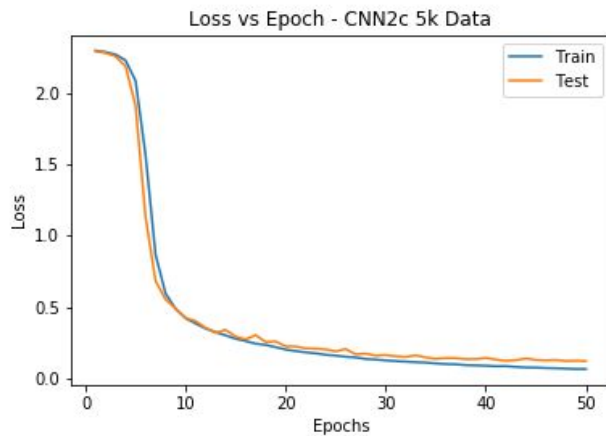
We can see that training loss always decreases because we are showing the same images to it with every epoch. However testing loss decreases for some time but either remains stagnant to increases with further epochs because of **Overfitting**. If we train for too long, the performance on the training dataset may continue to decrease because the model is overfitting and learning the irrelevant detail and noise in the training dataset. At the same time the error for the test set starts to rise again as the model's ability to generalize decreases.

This is due to the network learning specific images in the training set therefore performing poorly on the new images from the test set.

We can clearly see that the above model is overfitting because the test loss is more than training loss.

Not only by increasing **Epochs**, there are also other ways by which Overfitting occurs in a model. When the Training set is not representative of the Test set, or when there is **less data**. Overfitting occurs more easily when the complexity of the model increases.

Here is the result of training the model on **5000** images of the Train dataset out of 60000.



We can see that model overfits much more **easily and quickly** because less images are seen by our model.

Solving the problem of Overfitting

Learning and Generalization to new cases is hard.

1. Regularisation techniques like Dropout
2. Constrain the complexity of the model.
3. Generate more data i.e Data Augmentation techniques.

DROPOUT

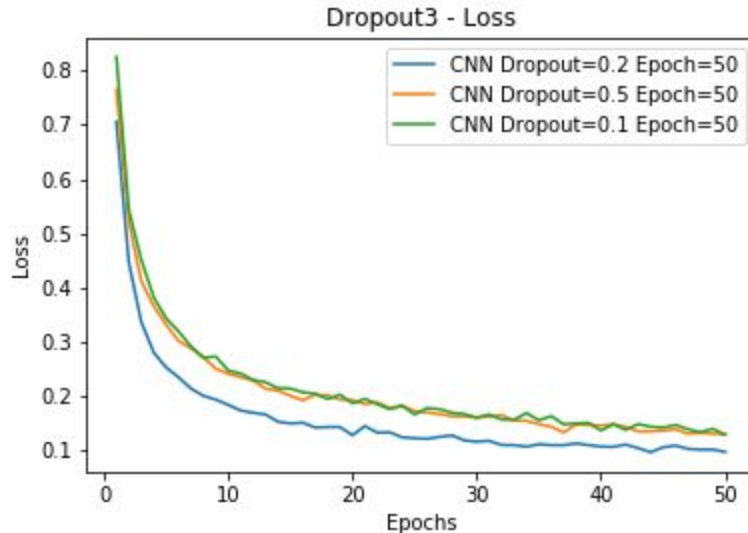
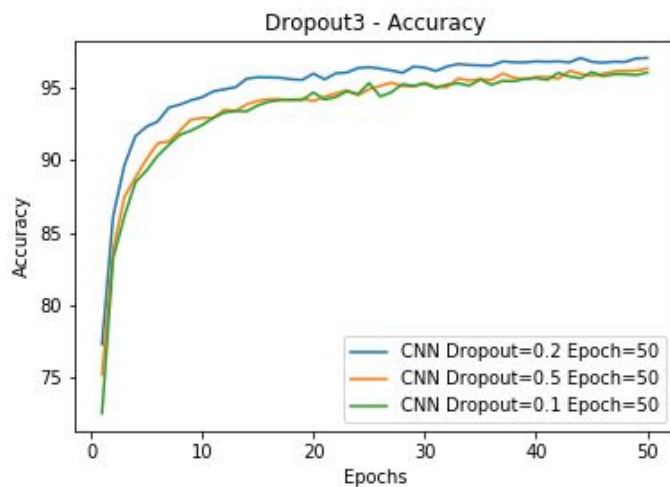
In a dropout layer, we randomly throw away activations/inputs with a probability of p . It results in making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.

The retained weights are rescaled to be used further. If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time. Also

Pytorch implements this rescaling of weights at training time, after each weight update at the end of the mini-batch.

We tested our default CNN with different values of dropout $p = 0.1$ 0.2 and 0.5 . In practice this is usually done by Grid Search parameters.

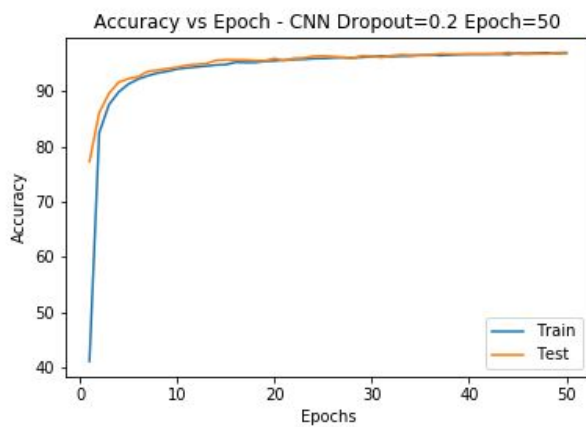
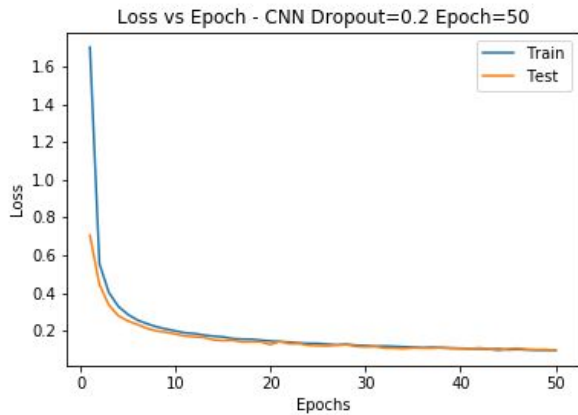
This is the loss and accuracy comparison for different p values.



This shows that best accuracy is given by mid value $p = 0.2$. If we throw away too many activations, our model might Underfit and if we don't throw away enough values, our model may overfit.

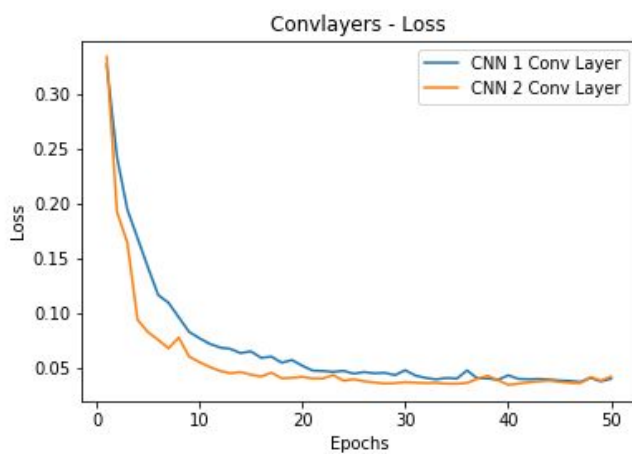
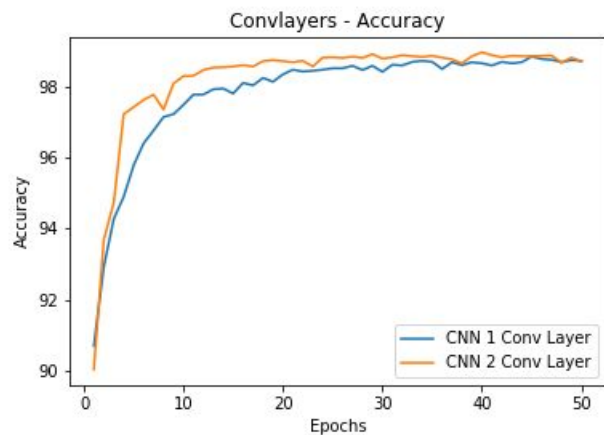
Also using dropout allows us to use Larger/complex architectures having less worry of overfitting.

We also know that $p=0.2$ solved the problem because of test/train loss. Below is the plot. Good Fit



Number of Conv2d Layers

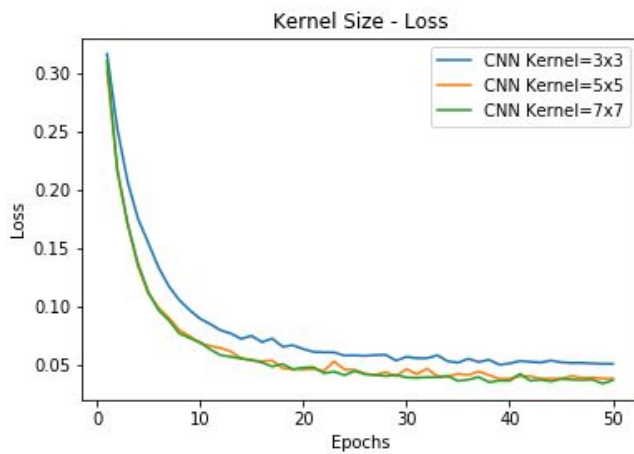
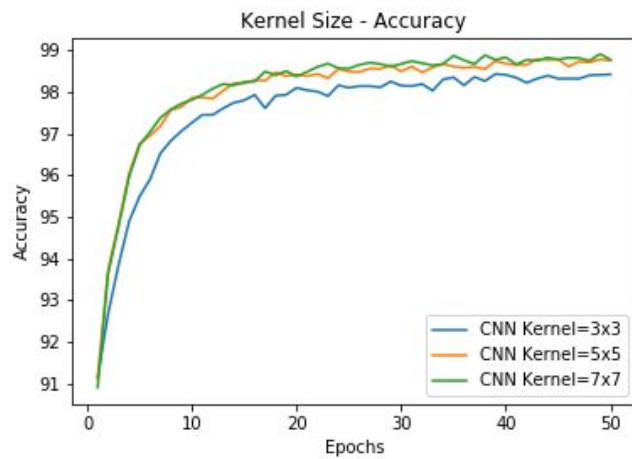
We studied the effect of adding more convolutional layers. Increasing the complexity increases the accuracy initially but model start overfitting in later epochs.



Increasing the size of Kernel

We experimented with different sizes of masks in a single conv2d layer network. We tried 3x3, 5x5 and 7x7.

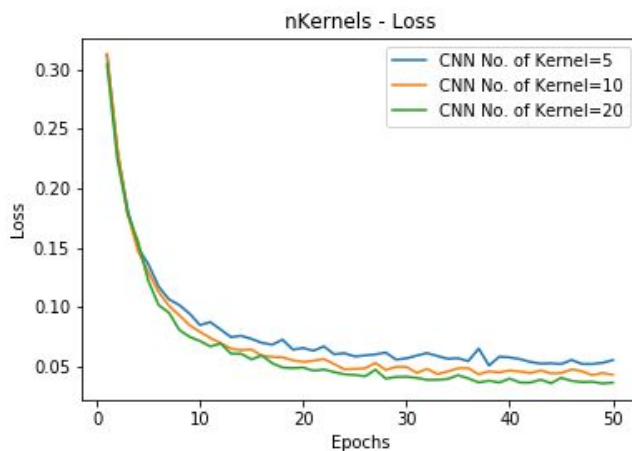
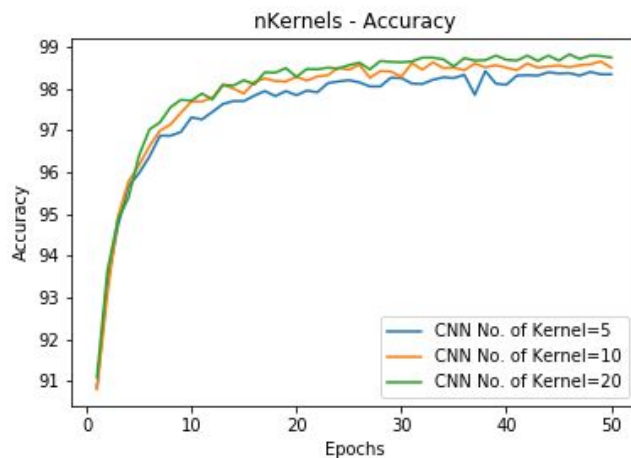
We observed that increasing sizes increases accuracy. This is because a pixel is able to look at more pixels in its neighbourhood and form dependencies with them. Counter argument can be given that minute and intrinsic features are left out with larger kernels. But due to the simplicity of MNIST dataset, there are less minute details that need to be captured.



Smaller Filter Sizes	Larger Filter Sizes
It has a smaller receptive field as it looks at very few pixels at once.	Larger receptive field per layer.
Highly local features extracted without much image overview.	Quite generic features extracted spread across the image.
Therefore captures smaller, complex features in the image.	Therefore captures the basic components in the image.
Amount of information extracted will be vast, maybe useful in later layers.	Amount of information extracted are considerably lesser.
Slow reduction in the image dimension can make the network deep	Fast reduction in the image dimension makes the network shallow
Better weight sharing	Poorer weight sharing
In an extreme scenario, using a 1x1 convolution is like treating each pixel as a useful feature.	Using a image sized filter is equivalent to a fully connected layer.

Increasing the number of Kernels in the Conv2d Layer

We experiment with increasing the number of kernels in a single conv2d layer CNN. Increasing the number of kernels means more parameters and more complex models. Complex models can fit more complex functions thus increasing accuracy. This is seen in the experiment. nKernels = 5 or 10 or 20



20 Kernels CNN gives the best performance.

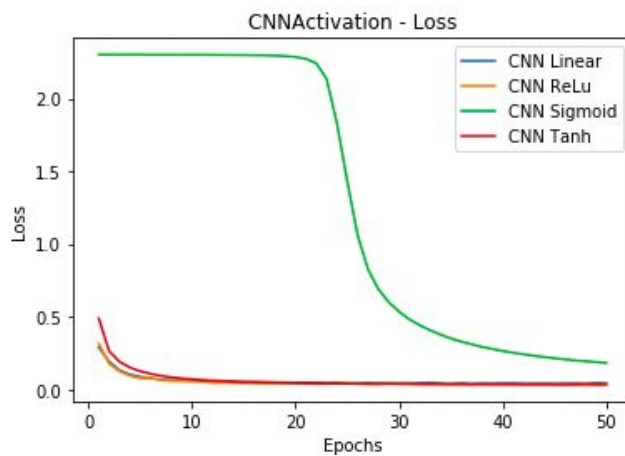
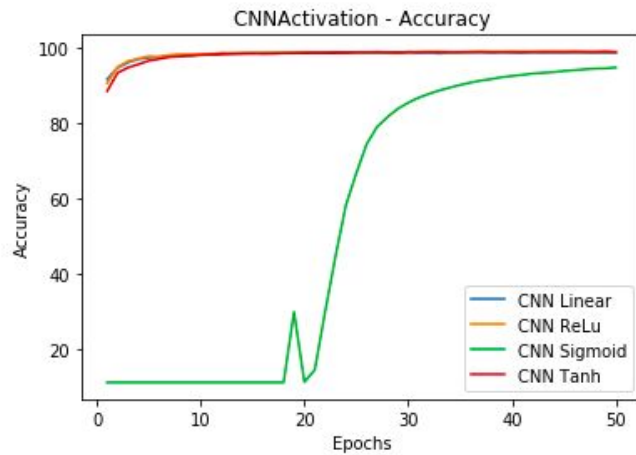
Activation Function:

We experiment with different activation functions. These activation functions are applied in 2Conv layer CNN and are applied to output of each hidden layer except the output layer which has Softmax layer only because we are doing Classification.

We observed that ReLu performed the best out all other functions. This is because it carries forward the inputs only which are positive (Sparsity) and also backprops the derivatives without altering them therefore reducing the likelihood of **Vanishing Gradients**.

In contrast, the gradient of sigmoids becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning. Other benefit of ReLu

is that for negative values it returns 0 resulting in sparse representation in contrast to Sigmoid which gives values regardless of sign of input resulting in **Dense** representations. In practice, networks with Relu tend to show better convergence performance than sigmoid. Here is comparison of ReLu, Sigmoid, tanh, Linear



Multi Layer Perceptron

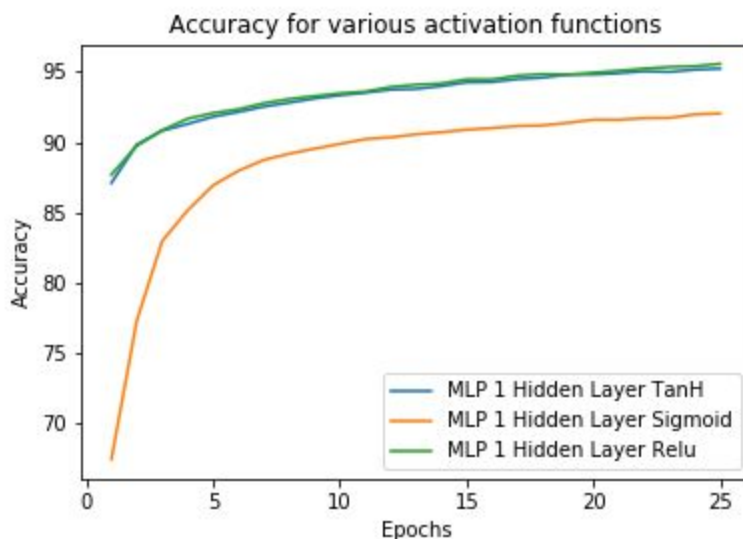
A multilayer perceptron (MLP) is a class of feedforward artificial neural network. A MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable.

A multilayer perceptron is a neural network connecting multiple layers in a directed graph, which means that the signal path through the nodes only goes one way. Each node, apart from the input nodes, has a nonlinear activation function. An MLP uses backpropagation as a supervised learning technique. Since there are multiple layers of neurons, MLP is a deep learning technique.

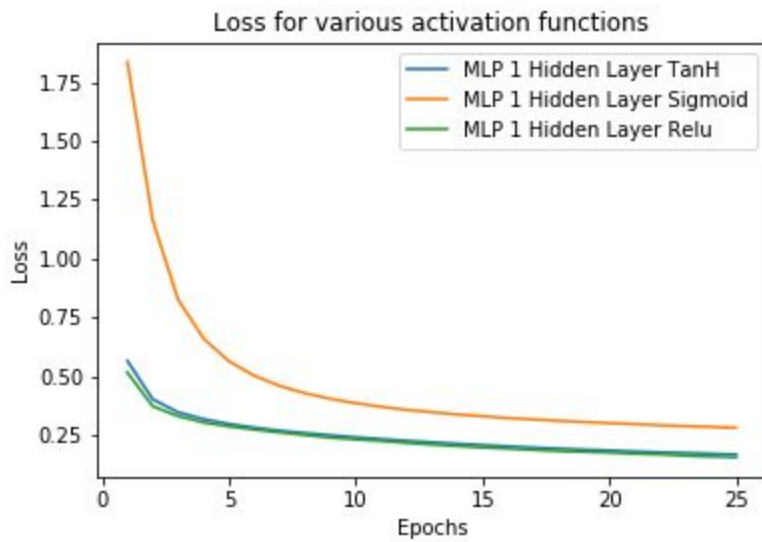
Effect of Different Activation Functions:

We are studying the use 3 activation functions across our model for 3 hidden layers and see the variation in accuracy and loss. The activation functions used are ReLU, Sigmoid and Tanh. From the below results we can see that ReLU (Rectified linear unit) gives the best accuracy among the ones tested on.

1. Accuracy



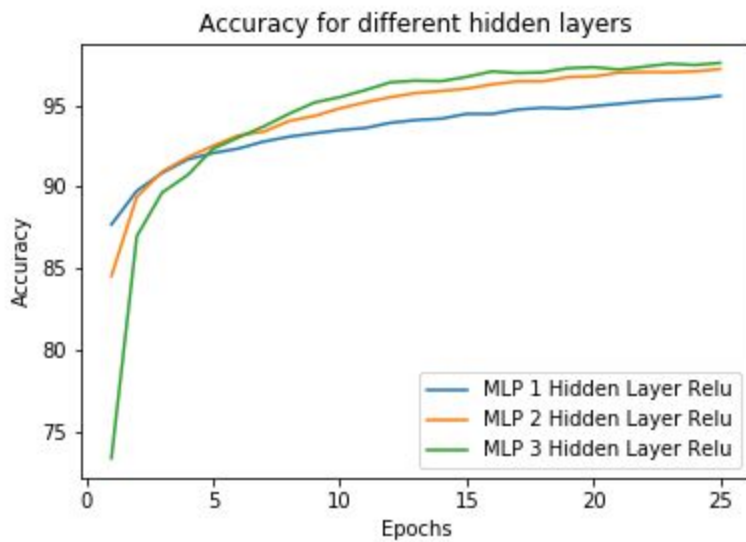
2. Loss



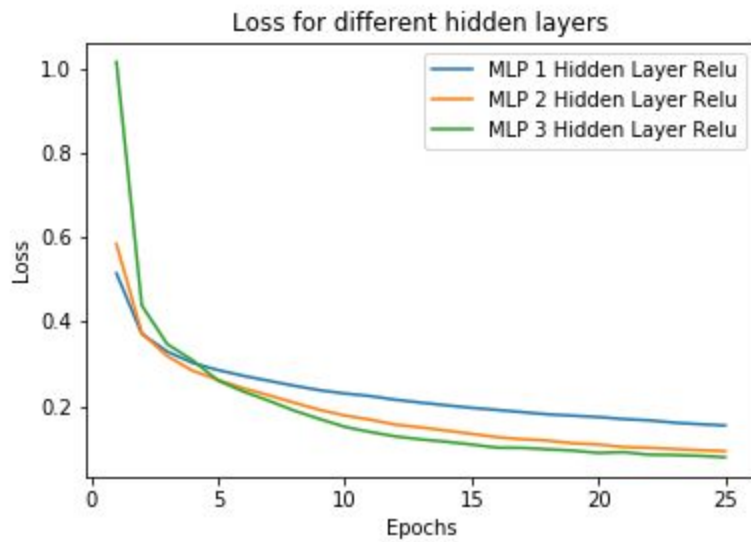
Effect of Hidden Layers:

We are experimenting with the number of hidden layers(1, 2, and 3 respectively) and see the variation in accuracy and loss. So increasing the number of hidden layers shows an increase in the accuracy.

1.Accuracy



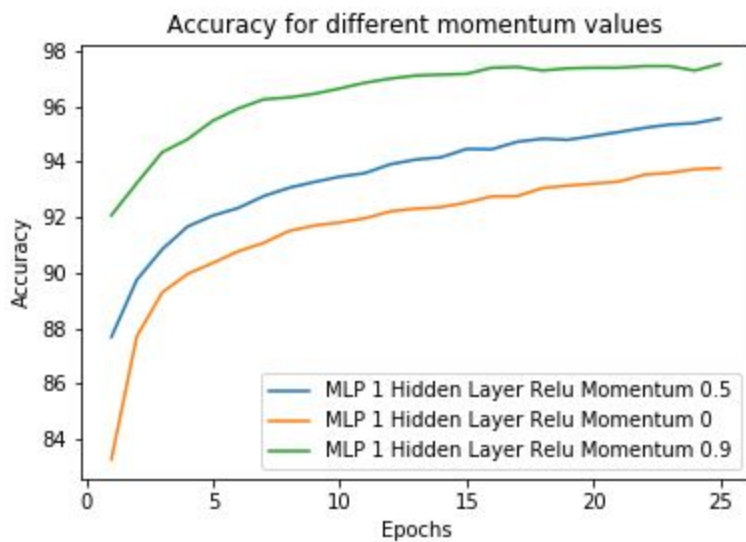
2.Loss



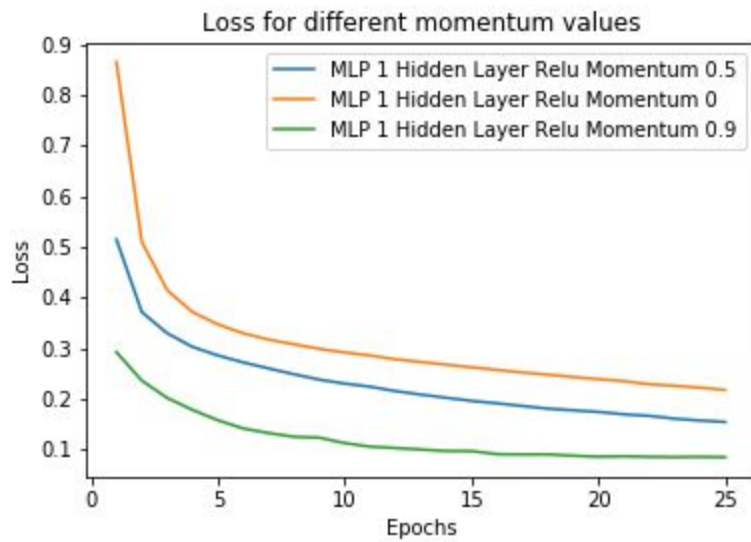
Effect of Changing Momentum:

We are experimenting with momentum (0, 0.5, and 0.9 respectively) and see the variation in accuracy and loss. momentum is a method which helps accelerate gradients vectors in the right directions, thus leading to faster converging.

1.Accuracy



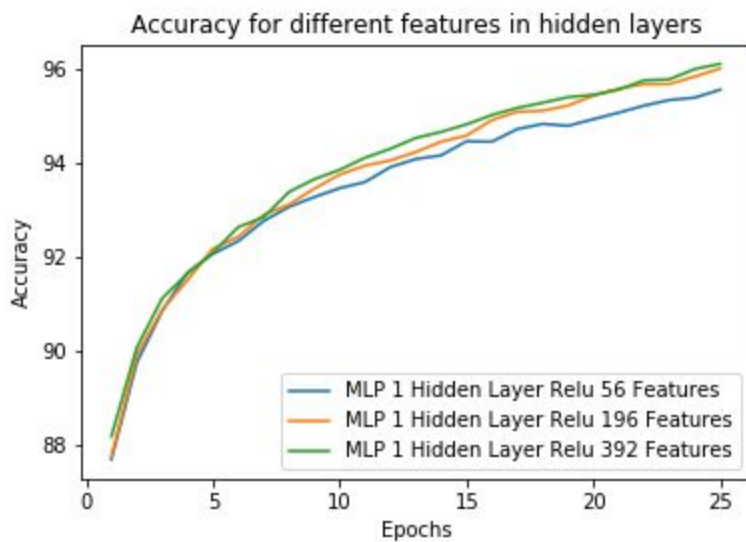
2.Loss



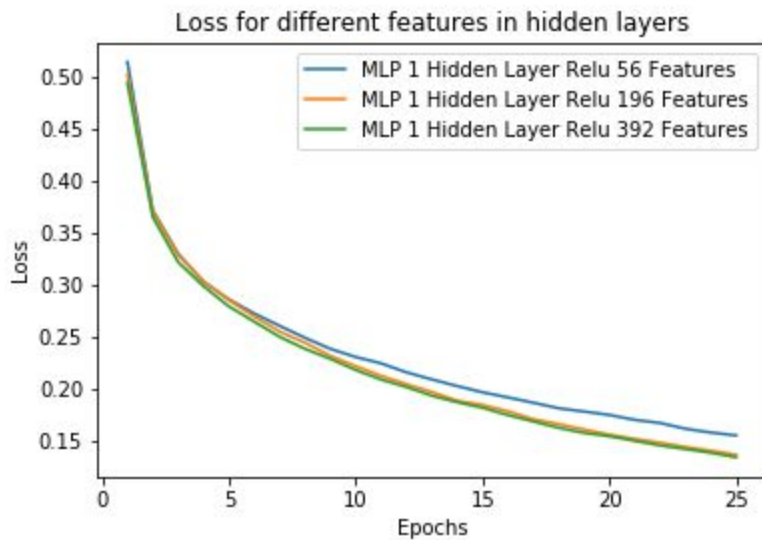
Effect of Increasing number of features in hidden layers:

Increasing the number of features in the hidden layers shows an increase in the accuracy as the complexity of data is increased, and it has more parameters to express the data.

1.Accuracy



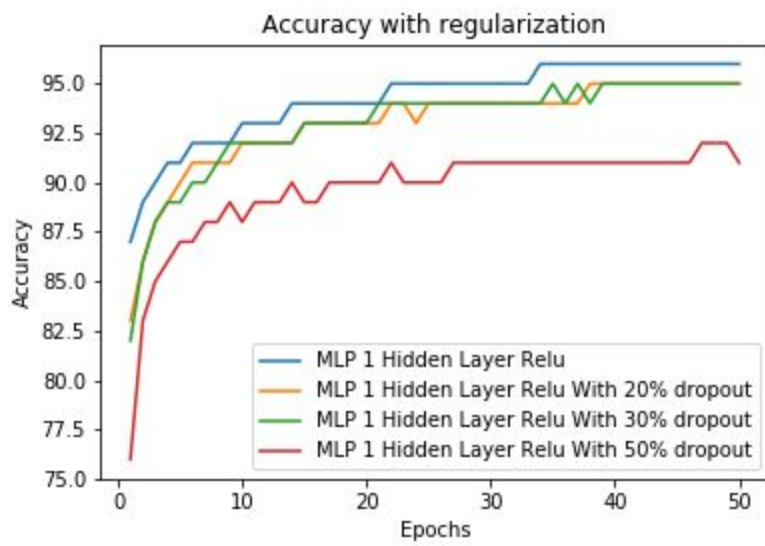
2.Loss



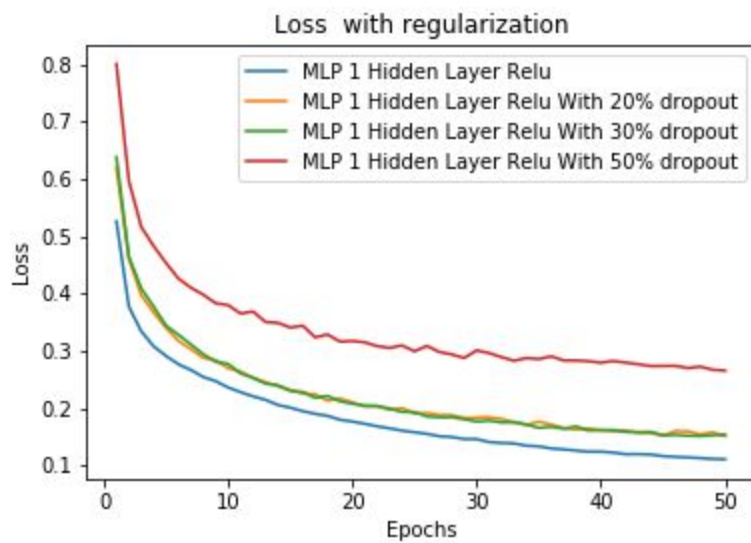
Effect of Dropout:

To overcome overfitting problem, a popular technique used is dropping out random values when transferring values between layers. A certain percentage of trained data is dropped while transferring data between layers. It randomly selects the data which helps avoiding getting overfitting. We are experimenting with different values of dropout(20%, 30% and 50% respectively) and see the variation in accuracy and loss. More dropout means throwing away more values at random for more generalisation leading to lower training accuracy and lesser underfitting.

1.Accuracy



2.Loss

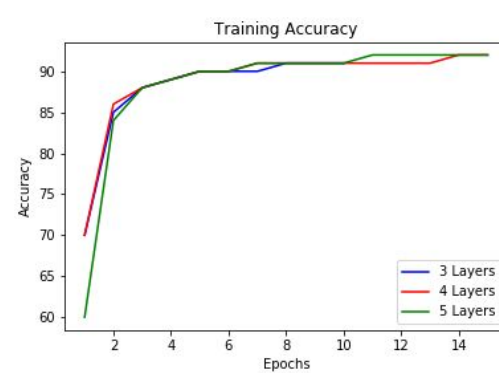
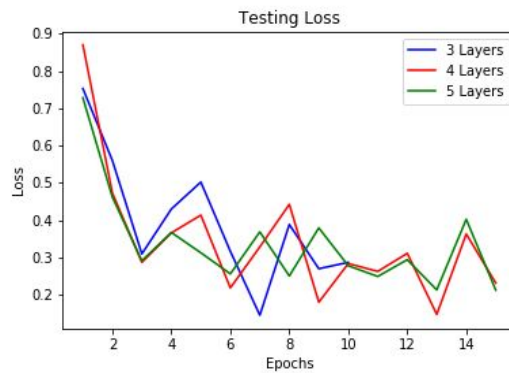
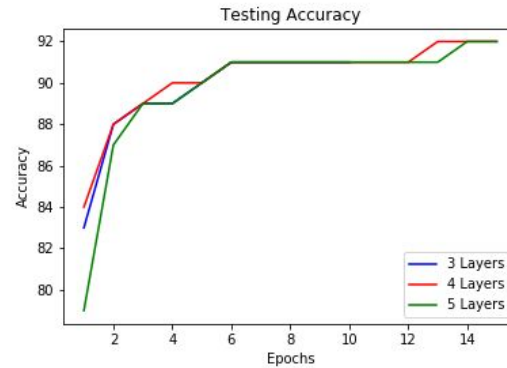
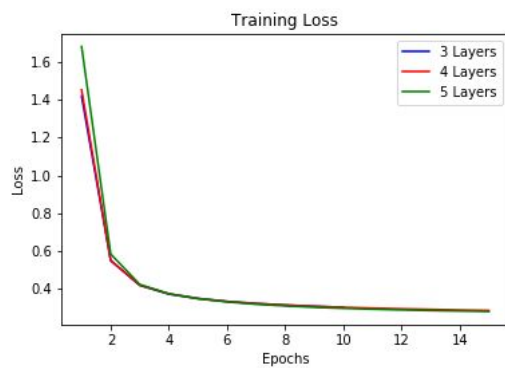


Deep Neural Network

Neural network is very complicated and powerful machine learning technique, it works by mimicking the human brain , there are various layers in the network where information is passed from one layer to another and as the information is passed on the model learns to recognise it.

Effect of Number of Layers:

We start with a 2 layers in the neural network, one by one we will add a hidden layers with 300 neurons and see the performance of the model on different parameters. The hidden layers are fully connected with each other and no activation function is used.

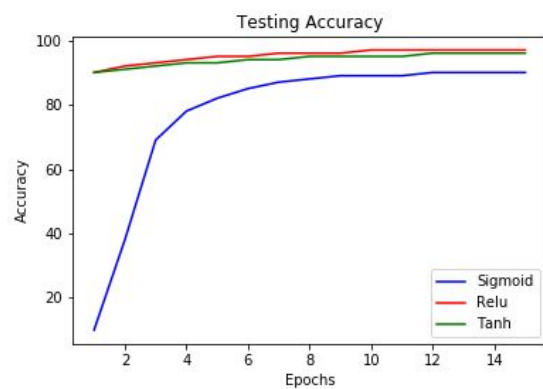
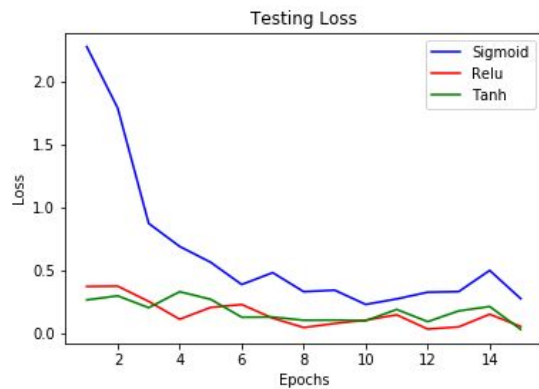
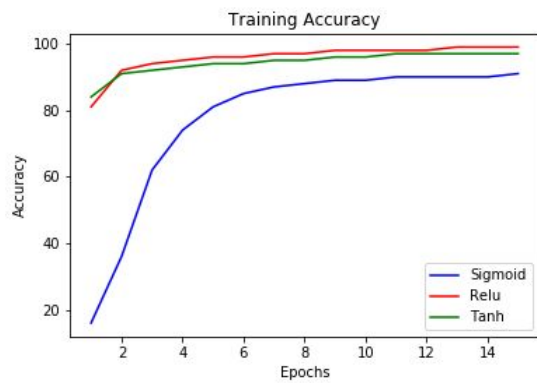
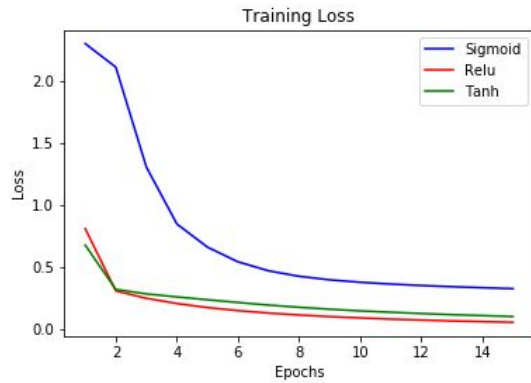


As seen from the above graphs there isn't much difference in the performance of models. The model with more layers are better but the model with 3 layers introduces enough complexity to fit the dataset properly.

Effect of Different Activation Functions:

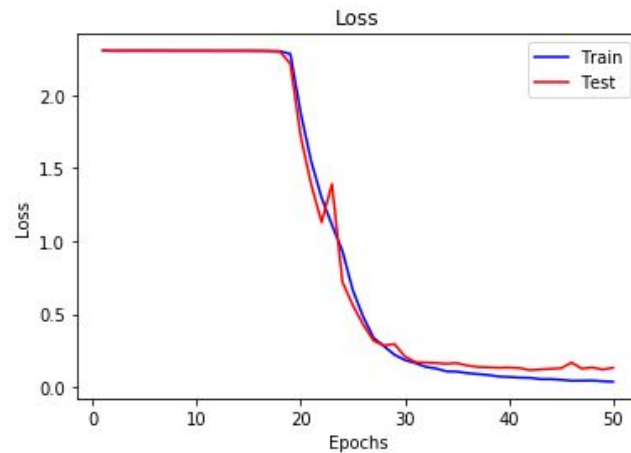
Activation functions introduce non linearity to the network so that our network can make sense of more complicated patterns.

Now we will use 3 layer network and see what effect does different activation have on performance of the model.



We see a general increase in accuracy of the model, and the testing loss becomes more stable. Performance of sigmoid activation function is poor as compared to the Relu and Tanh sigmoid function. This result was expected as ReLu and Tanh don't let the gradients saturate (Vanishing Gradients) and thus introduce more non linearity to the model, whereas the sigmoid curve saturates (becomes almost flat) for larger values.

Overfitting

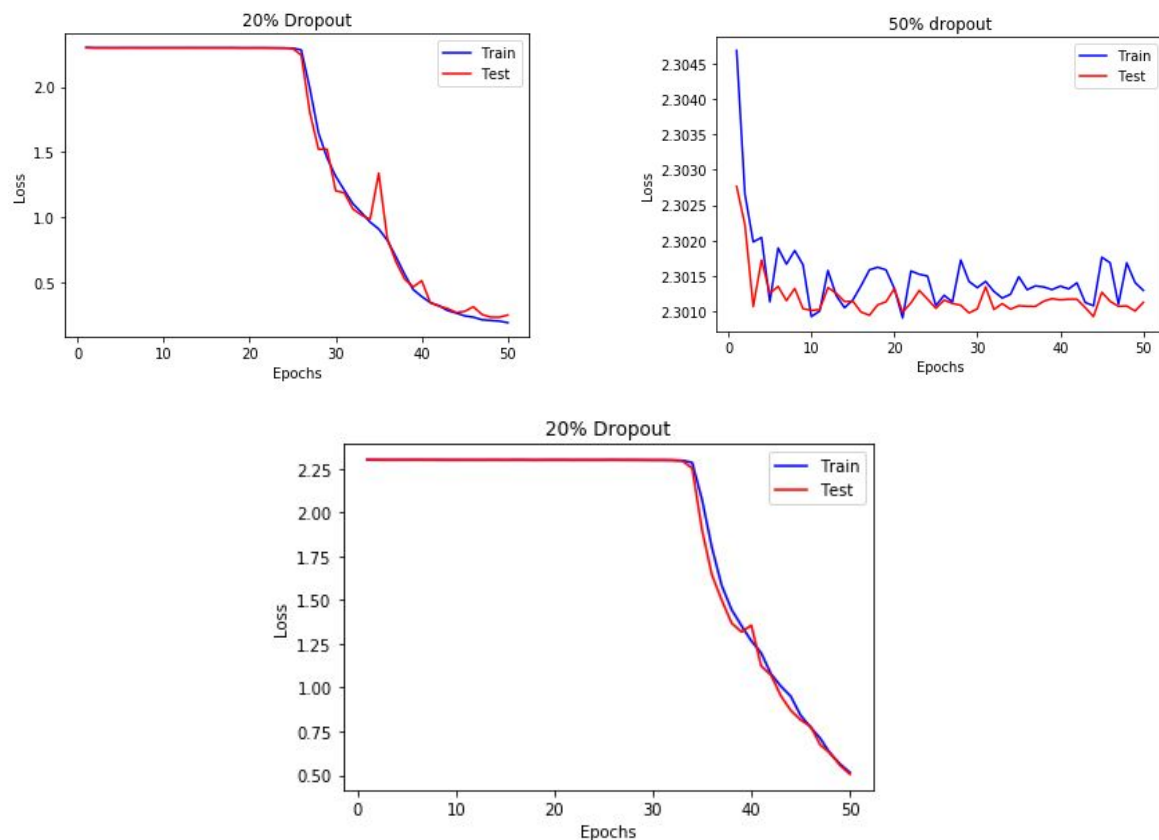


We trained a 7 layered network with ReLu activation on half of the train dataset to observe overfitting. Overfitting kicks in at 30 epochs where training loss continues to drop whereas test loss remains constant or increases, this is due to the fact that the complex model is learning unnecessary noise from the train set thus fitting the train set exactly but not generalizing enough.

Now to regularize the effect of overfitting we can

- a) Decrease the model complexity
- b) Get more training data
- c) Use dropout

Dropout



We observe that too little dropout doesn't work (10%) as we are not addressing the overfitting problem and if we use more dropout than required then the model starts to underfit (50%) as the loss isn't decreasing. Dropout with 20% probability works best as it fits and generalizes well.

CONCLUSION

There is a trade-off between time and accuracy. Higher the time, for which we train our Neural Networks, they are able to achieve higher accuracy, but with increasing Epochs to much higher value can lead to overfitting. This is why we use Regularisation in form of Dropouts to help the model learn generally.

Convolutional Neural Networks exploit the locality in case of images because we only need to map a pixel with its neighbouring pixels only. They are able to map **spatial** features like edges, corners etc. It shows the importance of validation data.

Different values of hyperparameters e.g. - number of hidden layers, activation function etc. result in varied accuracy even for same neural network.

Complexity of model allow it to learn more complex patterns and features but make it computationally expensive and also prone to overfitting.

Code

MLP.py

```
#!/usr/bin/env python
# coding: utf-8

# In[8]:

import torch
```

```
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
from torch.nn import functional as F
import matplotlib.pyplot as plt
from statistics import mean

# In[35]:

# Hyper Parameters
input_size = 784
inter = 56
num_classes = 10
num_epochs = 15
batch_size = 100
learning_rate = 0.01
momentum = 0.5

# In[36]:

# MNIST Dataset (Images and Labels)
train_dataset = dsets.MNIST(root='./files',
                             train=True,
                             transform=transforms.ToTensor(),
                             download=True)

test_dataset = dsets.MNIST(root='./files',
                            train=False,
                            transform=transforms.ToTensor(),
                            download=True)

# Dataset Loader (Input Pipeline)
```

```

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                             batch_size=batch_size,
                                             shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                           batch_size=batch_size,
                                           shuffle=False)

# In[37]:

print(len(train_dataset))
print(len(test_dataset))

# In[38]:

# GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# ## MODEL

# In[39]:

class MLP1linear(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(MLP1linear, self).__init__()
        self.fc1 = nn.Linear(input_size, inter)
        self.fc2 = nn.Linear(inter, num_classes)

    def forward(self, x):
        out = self.fc2(self.fc1(x))
        return out

```

```

class MLP1relu(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(MLP1relu, self).__init__()
        self.fc1 = nn.Linear(input_size, inter)
        self.fc2 = nn.Linear(inter, num_classes)

    def forward(self, x):
        out = self.fc2(F.relu(self.fc1(x)))
        return out

class MLP1relu2(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(MLP1relu2, self).__init__()
        self.fc1 = nn.Linear(input_size, inter)
        self.dropper = nn.Dropout(p=0.2)
        self.fc2 = nn.Linear(inter, num_classes)

    def forward(self, x):
        out = self.fc2(self.dropper(F.relu(self.fc1(x))))
        return out

class MLP1relu3(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(MLP1relu3, self).__init__()
        self.fc1 = nn.Linear(input_size, inter)
        self.dropper = nn.Dropout(p=0.3)
        self.fc2 = nn.Linear(inter, num_classes)

    def forward(self, x):
        out = self.fc2(self.dropper(F.relu(self.fc1(x))))
        return out

class MLP1relu5(nn.Module):
    def __init__(self, input_size, inter, num_classes):

```

```

        super(MLP1relu5, self).__init__()
        self.fc1 = nn.Linear(input_size, inter)
        self.dropper = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(inter, num_classes)

    def forward(self, x):
        out = self.fc2(self.dropper(F.relu(self.fc1(x))))
        return out

class MLP1tanh(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(MLP1tanh, self).__init__()
        self.fc1 = nn.Linear(input_size, inter)
        self.fc2 = nn.Linear(inter, num_classes)

    def forward(self, x):
        out = self.fc2(F.tanh(self.fc1(x)))
        return out

class MLP1sig(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(MLP1tanh, self).__init__()
        self.fc1 = nn.Linear(input_size, inter)
        self.fc2 = nn.Linear(inter, num_classes)

    def forward(self, x):
        out = self.fc2(F.sigmoid(self.fc1(x)))
        return out

class MLP2linear(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(MLP2linear, self).__init__()
        self.fc1 = nn.Linear(input_size, 300)
        self.fc2 = nn.Linear(300, inter)
        self.fc3 = nn.Linear(inter, num_classes)

```

```

def forward(self, x):
    out = self.fc3(self.fc2(self.fc1(x)))
    return out

class MLP3linear(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(MLP2linear, self).__init__()
        self.fc1 = nn.Linear(input_size, 300)
        self.fc2 = nn.Linear(300, inter)
        self.fc3 = nn.Linear(inter, num_classes)

    def forward(self, x):
        out = self.fc4(self.fc3(self.fc2(self.fc1(x))))
        return out

class CNN1c(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN1c, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.fc1 = nn.Linear(1440, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = x.view(-1, 1440)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN2c(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN2c, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)

```

```

        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN1ck3(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN1ck3, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=3)
        self.fc1 = nn.Linear(1690, 320)
        self.fc2 = nn.Linear(320, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = x.view(-1, 1690)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN1ck5(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN1ck5, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.fc1 = nn.Linear(1440, 320)
        self.fc2 = nn.Linear(320, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = x.view(-1, 1440)
        x = F.relu(self.fc1(x))

```

```

        x = self.fc2(x)
        return F.log_softmax(x)

class CNN1ck7(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN1ck7, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=7)
        self.fc1 = nn.Linear(1210, 320)
        self.fc2 = nn.Linear(320, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = x.view(-1, 1210)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN2cLinear(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN2cLinear, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = (F.max_pool2d(self.conv1(x), 2))
        x = (F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)
        x = (self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN2cTanh(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN2cTanh, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)

```



```

        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.tanh(F.max_pool2d(self.conv1(x), 2))
        x = F.tanh(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)
        x = F.tanh(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN2cSigmoid(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN2cSigmoid, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.sigmoid(F.max_pool2d(self.conv1(x), 2))
        x = F.sigmoid(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)
        x = F.sigmoid(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN1cNK5(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN1cNK5, self).__init__()
        self.conv1 = nn.Conv2d(1, 5, kernel_size=5)
        self.fc1 = nn.Linear(720, 320)
        self.fc2 = nn.Linear(320, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))

```

```

        x = x.view(-1, 720)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN1cNK10(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN1cNK10, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.fc1 = nn.Linear(1440, 320)
        self.fc2 = nn.Linear(320, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = x.view(-1, 1440)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

class CNN1cNK20(nn.Module):
    def __init__(self, input_size, inter, num_classes):
        super(CNN1cNK20, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, kernel_size=5)
        self.fc1 = nn.Linear(2880, 320)
        self.fc2 = nn.Linear(320, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = x.view(-1, 2880)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)

# In[40]:

```

```

def test(test_loader, device, model, criterion, reshape):
    correct = 0
    total = 0
    test_loss = []
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        if reshape==1:
            images = Variable(images.view(-1, 28*28))
        outputs = model(images)
        loss = criterion(outputs, labels)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()
        test_loss.append(loss.item())
    return (int( (100 * correct / total))),mean(test_loss)

```

```

def runner(input_size, inter, num_classes, filename, train_loader,
test_loader, learning_rate, device, num_epochs, momentum, model_class, criterion
, reshape):
    model = model_class(input_size, inter, num_classes)
    model.to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,
momentum=momentum)
    training_losses = []
    accuracy = []
    test_loss = []
    accuracy_test = []
    for epoch in range(num_epochs):
        print(epoch, num_epochs)
        correct = 0
        train_loss = []
        total = 0
        for i, (images, labels) in enumerate(train_loader):

```

```

        images = images.to(device)
        labels = labels.to(device)
        if reshape==1:
            images = Variable(images.view(-1, 28*28))
        labels = Variable(labels)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        total += labels.size(0)
        _, predicted = torch.max(outputs.data, 1)
        correct += (predicted==labels).sum()
        loss.backward()
        optimizer.step()
        train_loss.append(loss.item())
    acc, lo = test(test_loader,device,model,criterion,reshape)
    test_loss.append(lo)
    training_losses.append(mean(train_loss))
    accuracy_test.append(acc)
    accuracy.append(int( (100 * correct / total)))

#    accuracy = int( (100 * correct / total))
    fig = plt.figure()
    plt.plot(range(len(training_losses)+1)[1:],
training_losses, label='Train')
    plt.plot(range(len(test_loss)+1)[1:], test_loss, label='Test')
    plt.title('Loss vs Epoch - ' + str(filename))
    plt.xlabel('Epochs')
    plt.legend()
    plt.ylabel('Loss')
    plt.savefig(str(filename) + ' loss vs epoch.png')
    plt.clf()
    fig = plt.figure()
    plt.plot(range(len(accuracy)+1)[1:], accuracy, label='Train')
    plt.plot(range(len(accuracy_test)+1)[1:], accuracy_test, label='Test')
    plt.title('Accuracy vs Epoch - ' + str(filename))
    plt.xlabel('Epochs')

```

```

plt.ylabel('Accuracy')
plt.legend()
plt.savefig(str(filename) + ' accuracy vs epoch.png')
return [accuracy_test, test_loss]

def plot_losses(losses, title, loa=1):
    # loa == loss or accuracy
    ylabel = 'Accuracy'
    if loa == 2:
        ylabel = 'Loss'
    fig = plt.figure()
    for _loss in losses:
        loss = _loss[loa]
        label = _loss[0]
        plt.plot(range(len(loss)+1)[1:], loss, label=label)
    plt.title(title)
    plt.xlabel('Epochs')
    plt.ylabel(ylabel)
    plt.legend()
    plt.savefig(title+'.png')

# In[42]:

conv_layers = []
conv_activation = []
conv_nkernel = []
conv_kernelsize = []

filename = 'CNN 1 Conv Layer'
z = [filename] +
runner(784, 56, 10, filename, train_loader, test_loader, learning_rate, device, num_epochs, momentum, CNN1c, F.nll_loss, 0)
conv_layers.append(z)

```

```

filename = 'CNN 2 Conv Layer'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN2c,F.nll_loss,0)
conv_layers.append(z)

filename = 'CNN Kernel=3x3'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN1ck3,F.nll_loss,0)
conv_kernelsize.append(z)

filename = 'CNN Kernel=5x5'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN1ck5,F.nll_loss,0)
conv_kernelsize.append(z)

filename = 'CNN Kernel=7x7'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN1ck7,F.nll_loss,0)
conv_kernelsize.append(z)

filename = 'CNN No. of Kernel=5'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN1cNK5,F.nll_loss,0)
conv_nkernel.append(z)

filename = 'CNN No. of Kernel=10'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN1cNK10,F.nll_loss,0)
conv_nkernel.append(z)

filename = 'CNN No. of Kernel=20'

```

```

z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN1cNK20,F.nll_loss,0)
conv_nkernel.append(z)

filename = 'CNN Linear'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN2cLinear,F.nll_loss,0)
conv_activation.append(z)

filename = 'CNN ReLu'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN2c,F.nll_loss,0)
conv_activation.append(z)

filename = 'CNN Sigmoid'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN2cSigmoid,F.nll_loss,0)
conv_activation.append(z)

filename = 'CNN Tanh'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN2cTanh,F.nll_loss,0)
conv_activation.append(z)

# filename = 'MLP 1 Hidden Layer Relu With 50% dropout'
# z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,MLP1relu5,nn.CrossEntropyLoss())
# dropouts.append(z)

```

```
# In[50]:
```

```
conv_dropout = []
```

```
class CNN2c2dr(nn.Module):
```

```
    def __init__(self, input_size, inter, num_classes):
```

```
        super(CNN2c2dr, self).__init__()
```

```
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
```

```
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
```

```
        self.conv2_drop = nn.Dropout2d(p=0.2)
```

```
        self.fc1 = nn.Linear(320, 50)
```

```
        self.fc2 = nn.Linear(50, 10)
```

```
    def forward(self, x):
```

```
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
```

```
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
```

```
        x = x.view(-1, 320)
```

```
        x = F.relu(self.fc1(x))
```

```
        x = F.dropout(x, training=self.training)
```

```
        x = self.fc2(x)
```

```
        return F.log_softmax(x)
```

```
class CNN2c5dr(nn.Module):
```

```
    def __init__(self, input_size, inter, num_classes):
```

```
        super(CNN2c5dr, self).__init__()
```

```
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
```

```
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
```

```
        self.conv2_drop = nn.Dropout2d(p=0.5)
```

```
        self.fc1 = nn.Linear(320, 50)
```

```
        self.fc2 = nn.Linear(50, 10)
```

```
    def forward(self, x):
```

```
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
```

```
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
```

```
        x = x.view(-1, 320)
```

```
        x = F.relu(self.fc1(x))
```



```
x = F.dropout(x, training=self.training)
x = self.fc2(x)
return F.log_softmax(x)
```

```
# In[48]:
```

```
filename = 'CNN Dropout=0.2'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN2c2dr,F.nll_loss,0)
conv_activation.append(z)
```

```
# In[51]:
```

```
filename = 'CNN Dropout=0.5'
z = [filename] +
runner(784,56,10,filename,train_loader,test_loader,learning_rate,device,num_epochs,momentum,CNN2c5dr,F.nll_loss,0)
conv_activation.append(z)
```

```
# In[58]:
```

```
conv_dropout = []
```

```
# In[59]:
```

```
conv_dropout.append(conv_activation[-1])
```

```
# In[60]:
```

```
conv_dropout.append(conv_activation[-2])
```

```
# In[ ]:
```

```
# In[62]:
```

```
plot_losses(conv_dropout, 'Dropout - Accuracy')
```

```
plot_losses(conv_dropout, 'Dropout - Loss', loa=2)
```

```
# In[44]:
```

```
# plot_losses(activation, 'Multi Layer Perceptron - Activation Functions')
```

```
# plot_losses(layers, 'Multi Layer Perceptron - Different Layers')
```

```
# print(activation)
```

```
plot_losses(conv_layers, 'Convlayers - Accuracy')
```

```
plot_losses(conv_layers, 'Convlayers - Loss', loa=2)
```

```
plot_losses(conv_kernelsize, 'Kernel Size - Accuracy')
```

```
plot_losses(conv_kernelsize, 'Kernel Size - Loss', loa=2)
```

```
plot_losses(conv_nkernel, 'nKernels - Accuracy')
```

```
plot_losses(conv_nkernel, 'nKernels - Loss', loa=2)
```

```
plot_losses(conv_activation, 'CNNActivation - Accuracy')
```

```
plot_losses(conv_activation, 'CNNActivation - Loss', loa=2)
```

```
# In[89]:

try:
    from google.colab import files
    get_ipython().system('zip output.zip *.png')
    while 1:
        try:
            files.download('output.zip')
            break
        except:
            continue
except:
    pass

# In[77]:
```

Logistic Regression.py

```
#!/usr/bin/env python
# coding: utf-8

# In[15]:

import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
```



```
# In[13]:
```

```
print(len(train_dataset))
```

```
print(len(test_dataset))
```

```
# ## MODEL
```

```
# In[16]:
```

```
class LogisticRegression(nn.Module):  
    def __init__(self, input_size, num_classes):  
        super(LogisticRegression, self).__init__()  
        self.linear = nn.Linear(input_size, num_classes)  
  
    def forward(self, x):  
        out = F.sigmoid(self.linear(x))  
        return out
```

```
# In[17]:
```

```
#Instantiate the model class
```

```
model = LogisticRegression(input_size, num_classes)
```

```
# Loss: Computes Softmax and then CrossEntropy loss
```

```
criterion = nn.CrossEntropyLoss()
```

```
#Optimizer: Stochastic Gradient Descent
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```

# In[24]:

training_losses = []

# Training the Model: If you Rerun this cell, model start training from
where you left it (Weights)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images.view(-1, 28*28))
        labels = Variable(labels)

        # Forward + Backward + Optimize
        #Reset the Gradients before Training
        optimizer.zero_grad()
        outputs = model(images)
        #Find loss using predicted output and true labels
        loss = criterion(outputs, labels)
        #BackProp the grads
        loss.backward()
        #Update the parameteres
        optimizer.step()

        training_losses.append(loss.data)
        if (i+1) % 100 == 0:
            print ('Epoch: [%d/%d], Step: [%d/%d], Loss: %.4f'
                  % (epoch+1, num_epochs, i+1,
len(train_dataset)//batch_size, loss.data))

# In[33]:

training_losses

# In[27]:

```

```

# Test the Model
correct = 0
total = 0
for images, labels in test_loader:
    images = Variable(images.view(-1, 28*28))
    outputs = model(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum()

print('Accuracy of the model on the 10000 test images: %d %%' % (100 *
correct / total))

# In[28]:

torch.save(model.state_dict(), './results/model_linear_regression.pth')
torch.save(optimizer.state_dict(),
'./results/optimizer_linear_regression.pth')

# In[ ]:

```

DeepNN.py

```

# Report link
#
https://docs.google.com/document/d/1rElNDpHfDFQjACL\_KcCWW5UUFW2EArOx8pxwyxsec7c/edit?usp=sharing

```

```
import torch
import torchvision
import matplotlib.pyplot as plt
from torch.autograd import Variable
import torchvision.datasets as dsets
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
from torch.nn import functional as F
from statistics import mean

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

n_epochs = 1
batch_size_train = 64
batch_size_test = 1000
learning_rate = 0.01
momentum = 0.5
log_interval = 10

random_seed = 1
torch.backends.cudnn.enabled = False
torch.manual_seed(random_seed)

# MNIST Dataset (Images and Labels)
train_dataset = dsets.MNIST(root='files',
                             train=True,
                             transform=transforms.ToTensor(),
                             download=True)

test_dataset = dsets.MNIST(root='files',
                            train=False,
```



```

        transform=transforms.ToTensor(),
        download=True)

# Dataset Loader (Input Pipeline)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                           batch_size=batch_size,
                                           shuffle=False)

examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)

# will have arrays of losses and accuracies for different setups
trainlosses = []
testlosses = []
trainacc = []
testacc = []

class Net5(nn.Module):

    def __init__(self, input_size, hidden_size, out_size):
        super(Net5, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, hidden_size)
        self.fc4 = nn.Linear(hidden_size, hidden_size)
        self.fc5 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        x = self.fc4(x)
        x = self.fc5(x)

```

```
        return F.log_softmax(x)
```

```
class Net3(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, out_size):
```

```
        super(Net3, self).__init__()
```

```
        self.fc1 = nn.Linear(input_size, hidden_size)
```

```
        self.fc2 = nn.Linear(hidden_size, hidden_size)
```

```
        self.fc3 = nn.Linear(hidden_size, out_size)
```

```
    def forward(self, x):
```

```
        x = self.fc1(x)
```

```
        x = self.fc2(x)
```

```
        x = self.fc3(x)
```

```
        return F.log_softmax(x)
```

```
class Net4(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, out_size):
```

```
        super(Net4, self).__init__()
```

```
        self.fc1 = nn.Linear(input_size, hidden_size)
```

```
        self.fc2 = nn.Linear(hidden_size, hidden_size)
```

```
        self.fc3 = nn.Linear(hidden_size, hidden_size)
```

```
        self.fc4 = nn.Linear(hidden_size, out_size)
```

```
    def forward(self, x):
```

```
        x = self.fc1(x)
```

```
        x = self.fc2(x)
```

```
        x = self.fc3(x)
```

```
        x = self.fc4(x)
```

```
        return F.log_softmax(x)
```

```
class Net3relu(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, out_size):
```

```
        super(Net3relu, self).__init__()
```

```
        self.fc1 = nn.Linear(input_size, hidden_size)
```

```

        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, out_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x)

class Net3tanh(nn.Module):

    def __init__(self, input_size, hidden_size, out_size):
        super(Net3tanh, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, out_size)

    def forward(self, x):
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x)

class Net3sigmoid(nn.Module):

    def __init__(self, input_size, hidden_size, out_size):
        super(Net3sigmoid, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, out_size)

    def forward(self, x):
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x)

```

```

class Net3reludrp(nn.Module):

    def __init__(self, input_size, hidden_size, out_size, prb):
        super(Net3reludrp, self).__init__()
        self.dropout = nn.Dropout(p=prb)
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, out_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return F.log_softmax(x)

input_size = 784
hidden_size = 300
output_size = 10
num_epochs = 15

learning_rate = 0.05
criterion = nn.CrossEntropyLoss()
model_list = []
#Optimizer: Stochastic Gradient Descent

# network = Net3(input_size,hidden_size, output_size)
# model_list.append(network)
# network = Net4(input_size,hidden_size, output_size)
# model_list.append(network)
# network = Net5(input_size,hidden_size, output_size)
# model_list.append(network)

# network = Net3sigmoid(input_size,hidden_size, output_size)
# model_list.append(network)

```

```

# network = Net3relu(input_size,hidden_size, output_size)
# model_list.append(network)
# network = Net3tanh(input_size,hidden_size, output_size)
# model_list.append(network)

network = Net3reludrp(input_size,hidden_size, output_size,0)
model_list.append(network)
network = Net3reludrp(input_size,hidden_size, output_size,0.2)
model_list.append(network)
network = Net3reludrp(input_size,hidden_size, output_size,0.5)
model_list.append(network)

for m in model_list:
    #Plotting loss
    optimizer = torch.optim.SGD(m.parameters(), lr=learning_rate)
    train_losses = []
    train_counter = []
    test_losses = []
    test_counter = [i*len(train_loader.dataset) for i in range(n_epochs +
1)]

    training_losses = []
    train_counter = []

    training_acc = []
    testing_acc = []

    testing_loss = []
    training_loss = []
    c=0

    # Training the Model: If you Rerun this cell, model start training from
where you left it (Weights)
    for epoch in range(num_epochs):
        l = []
        correct = 0
        total = 0

```

```

for i, (images, labels) in enumerate(train_loader):
    images = Variable(images.view(-1, 28*28))
    labels = Variable(labels)

    # Forward + Backward + Optimize
    #Reset the Gradients before Training
    optimizer.zero_grad()
    outputs = m(images)
    #Find loss using predicted output and true labels
    loss = criterion(outputs, labels)
    #BackProp the grads
    loss.backward()
    #Update the parameteres
    optimizer.step()
    l.append(loss.item())
#         training_losses.append(loss.item())
#         train_counter.append((i*64 +
((epoch-1)*len(train_loader.dataset)))

    output = m(images)
    _, predicted = torch.max(output.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum()

    if (i+1) % 100 == 0:
        print ('Epoch: [%d/%d], Step: [%d/%d], Loss: %.4f'
              % (epoch+1, num_epochs, i+1,
len(train_dataset)//batch_size, loss.data))
    # Train Loss
    c = c+1

train_counter.append(c)
training_loss.append(mean(l))
#         for image, label in train_loader:
#             image = Variable(image.view(-1, 28*28))
#             output = m(image)
#             _, predicted = torch.max(output.data, 1)

```

```

#         total += label.size(0)
#         correct += (predicted == label).sum()
#         print('Accuracy of the model on the 10000 test images: %d %%' %
(100 * correct / total))

#test acc
training_acc.append((100 * correct / total))

correct = 0
total = 0
l = []
for image, label in test_loader:
    image = Variable(image.view(-1, 28*28))
    output = m(image)
    loss = criterion(outputs, labels)
    l.append(loss.item())
    _, predicted = torch.max(output.data, 1)
    total += label.size(0)
    correct += (predicted == label).sum()
#         print('Accuracy of the model on the 10000 test images: %d %%' %
(100 * correct / total))

#test acc
testing_acc.append((100 * correct / total))
testing_loss.append(mean(l))

#training_losses
# fig = plt.figure()
# plt.plot(train_counter, training_losses, color='blue')
# plt.xlabel('number of training examples seen')
# plt.ylabel('negative log likelihood loss')
# plt.savefig('4.png')

# print(network)

```

```
print(len(training_loss))
trainlosses.append(training_loss)
testlosses.append(testing_loss)
trainacc.append(training_acc)
testacc.append(testing_acc)

plt.title('Training Loss')
plt.plot(train_counter, trainlosses[0], color='blue')
plt.plot(train_counter, trainlosses[1], color='red')
plt.plot(train_counter, trainlosses[2], color='green')
plt.legend(['Sigmoid', 'Relu', 'Tanh'], loc='upper right')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.savefig('activation-tl.png')
```