HOMEWORK #4

Issued: 11/4/2016 **Due: 12/4/2016**

By: Hariprasad Ravishankar **USC ID: 8991379333**

Problem 1: LeNet-5 Training and Its Application to the MNIST Dataset

1.1 Abstract and Motivation

Convolutional Neural Networks can be described as an extension of Artificial Neural Networks and are used for classification of images into multiple classes. In the case of Neural Networks, an input Image is passed through a set of hidden layers, with each layer having a number of neurons. In each layer, the number of neurons in the layer must be equal to the total number of pixels across all channels in the image. The value of each pixel will be multiplied by a weight and added to a bias, and further a non-linear function might be applied. Thus the total number of weights will be equal to the number of pixels in the image.

The problem with ANN for image classification is that, as the dimensions of the image increases, the number of weights to be learned also increases proportionally. This added to the fact that multiple hidden layers will be added makes ANNs scale poorly.

Thus, CNNs were introduced. CNNs have different named layers such as Conv Layers, Max Pooling layers and fully- connected layers. The Conv layers take into advantage the spatial similarity in an image, and stack up neurons in a 3D volume. Thus the weights form a 3D volume, and each 2D slice of weights act like a learnable filter which extracts features from the input images. The depth refers to the number of filters. Thus a 3D volume of weights in a Conv Layer learns the weights for a number of filters that describe the content of the image. This is similar to the application of Laws filters in that instead of predefined weights, we let the network learn the most activation weights.

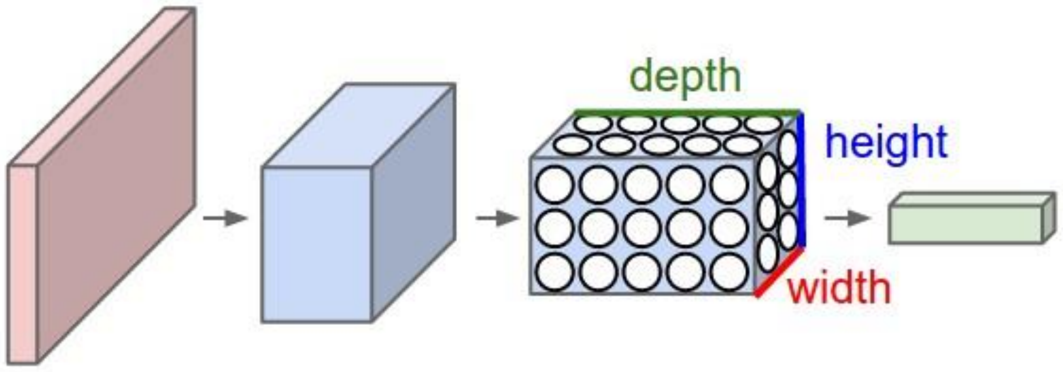

Figure 1: Architecture of a Convolutional Neural Network. Image Courtesy ~ http://cs231n.github.io/assets/cnn/cnn.jpeg

In this assignment, the LeNet-5 architecture described by Yan LeCunn [1] has been implemented to recognize hand written digits from the MNIST[2] dataset. Further, the performance of the architecture with negative MNIST has been studied, and modifications have been recommended. The Torch Deep Learning Framework based on Lua has been used for implementation.

1.2. Approach and Procedure

Components of LeNet-5
Convolutional Layer

The convolutional layer consists of a set of learnable filters. Each filter is a set of weights arranged in 2D and extracts one feature from the image. For example, in the first convolutional layer of the LeNet-5, there are 6, 5 x 5 filters. Each of these 5x5 filters are applied across the complete depth of the input volume. For example, the input volume for the Conv1 is 1x 32 x 32 whereas the input volume for the conv2 is 6 x 14 x 14. The filter application is achieved by sliding and convolving the filter weights with the local region (or volume) of the input image, by computing the dot product of the filter weights and the pixel values with that region. The filter is then slid by a stride, typically of 1 pixel and the dot products are computed again.

After the application of each filter, we obtain an activation map that represent the features that give the highest response for that particular filter. Thus, an activation map is obtained for each filter, and this forms as the input to the next hidden layer.

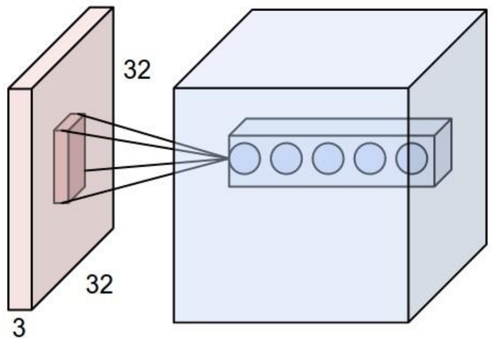

Figure 2: Convolution Layer Image Courtesy ~ http://cs231n.github.io/assets/cnn/depthcol.jpeg

The size of the output after each Convolutional layer can be computed using the mathematical formula.

$$NewHeight = \frac{Inputheight - Filterpadding}{Stride} + 1$$

$$NewWidth = \frac{Inputwidth - Filterpadding}{Stride} + 1$$

Thus the output size after the first convolutional layer in LeNet-5 is

New Dimension = $\frac{32-5+0}{1} + 1 = 28$

Rectified Linear Unit (ReLU)

The ReLU layer is a non-linearly function that is applied to the output of a Convolution or Fully Connected layer. It is mathematically defined as *max(0,x)* and thus the activation is thresholded at zero.

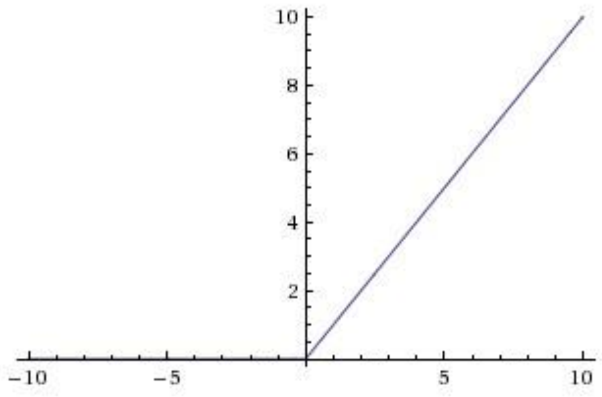

Figure 3: ReLU Activation Function. Image Courtesy ~ http://cs231n.github.io/assets/nn1/relu.jpeg

The ReLU unit has been found to accelerate the convergence of the stochastic gradient descent as compared to the other non-linearity functions such as sigmoid or tanh. This is primarily because of its simple thresholding operation and thus expensive computation of exponentials are avoided.

Max-Pooling Layer

A pooling layer is introduced to take advantage of the spatial similarity of the input image. Further, the pooling layer minimizes the size of the activation output. This considerably reduces the number of parameters to be learned in the next stage of the network.

In a Max-pooling layer, a window, typically of size 2 x 2 is slid with stride 2 across the activation map. The maximum activation value inside the 2 x 2 map is preserved and the rest are discarded.

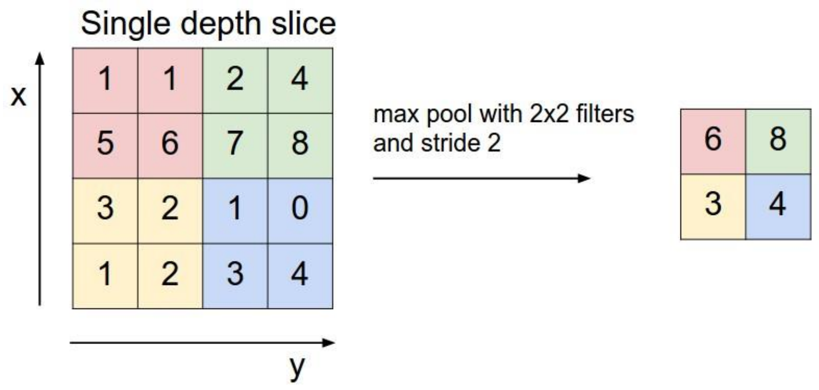

Figure 4: 2x2 Max-Pooling operation. Image Courtesy ~ http://cs231n.github.io/assets/cnn/maxpool.jpeg

Fully Connected Layer

Fully Connected layers are densely connected to all the activation output of the previous layer. These dense connections are identical to the ordinary Neural Network’s hidden layers and they help learn pixel wise weights.

Classification Layer

The last layer represents a classification layer which is connected to a LogSoftMax function to compute the class probability of the outputs. The classification layer has the number of neuron units equal to the number of classes in the dataset. The SoftMax function is mathematically

$$\text{SoftMax } P(y = j|x) = \frac{e^{x^T w_j}}{\sum_{k=1}^{K} e^{x^T w_k}}$$

Architecture of LeNet-5

The LeNet-5 architecture is designed to recognize hand written digits efficiently. The architecture can be described as follows

- Input images – These are 32 x 32 x 1 grayscale images
- Convolution 1 layer – This layer includes 6 learnable filters, each of size 5 x 5. There is no padding and the filter is applied with stride of 1 pixel.
- ReLU unit – The Rectified Linear Unit is an activation function that is applied to the output of the Conv1 layer and can be written as $max(0, x)$.
- Max-Pooling Layer – The Max-pooling layer is a subsampling layers, that shrinks the output of the Convolution layer. A 2x2 Max-pooling layer is used with stride of 2 pixels. Thus the output of the Conv1 layer will be shrunk in half across the height and width directions.
- Convolution 2 layer – This layer includes 16 learnable filters with a filter size of 5 x 5, with no padding and stride of 1 pixel.
- ReLU and Max-pooling layer follow the Convolution 2 layer
- Fully Connected layer 1 – A fully connected dense layer with 120 neurons.
- Fully Connected Layer 2 – A fully connected dense layer with 84 neurons.
- Fully Connected Layer 3 – A fully connected dense layer with 10 neurons to categorize 10 classes.
- The final layer is a LogSoftMax layer that gives the probability estimation for 10 classes.

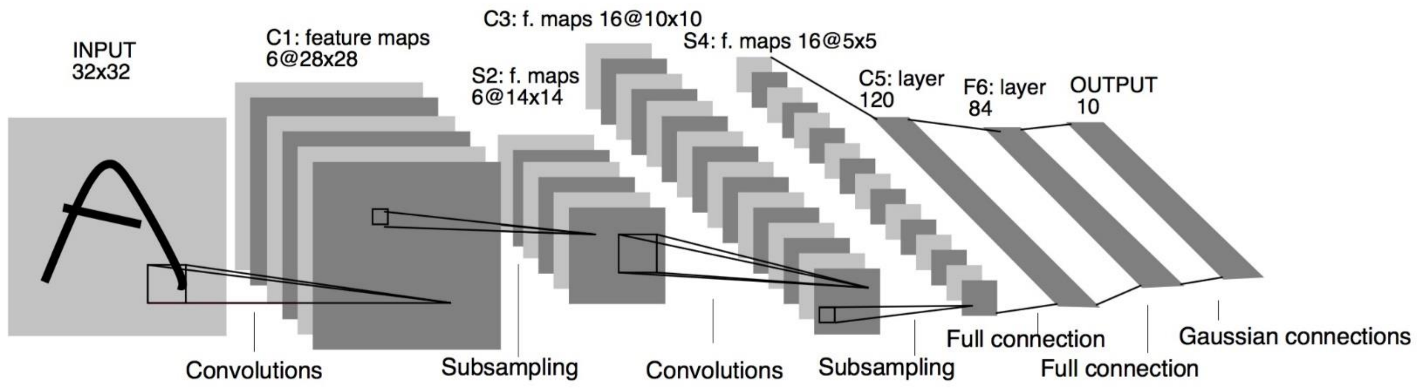

Figure 5: LeNet-5 Architecture. Image Courtesy ~ [1]

The MNIST dataset has 60000 samples for training and 10000 samples for testing. The training parameters and procedure followed can be explained in more detail below.

Initialization Scheme

All the weights need to be initialized prior to training. This initializing step is important as it would determine the parameters learnt by the network and the rate of convergence. All weights have to be randomly initialized in order the break the symmetry as otherwise all the neurons will learn the same feature Two initialization schemes were used here

Random Initializing

All the parameters in Torch are by default initialized randomly from a uniform distribution from [-stdv,stdv)

The default value of stdv that is chosen is equal to

$$stdv = \frac{1}{\sqrt{Height*width*numberofinputplanes}}$$

Where Height = Height of Filter

Width = Width of Filter

Thus, for the first Convolution layer, height, width = 5, number of input planes is 1

Thus weights are randomly initialized from ***uniform (-0.2,0.2)***

Heuristic Initializing

The Heuristic Initializing is the initialization scheme using in [3] "Efficient Backprop" by Yann LeCun

The initializing scheme choses random parameters from uniform distribution from[-stdv,stdv)

Here stdv is given as

$$stdv = \frac{1}{\sqrt{3*Height*width*numberofinputplanes}}$$

Thus for the first Convolutional layers, the weights are randomly initialized from **uniform(-0.11,0.11)**

It was noticed that Heuristic Initialization gave a better convergence than the default initialization scheme.

The Heuristic initializing scheme was implemented using the Open Source Lua function available here

https://github.com/e-lab/torch-toolbox/blob/master/Weight-init/weight-init.lua

```lua
-- design model
require('nn')
local model = nn.Sequential()
model:add(nn.SpatialConvolutionMM(3,4,5,5))

-- reset weights
local method = 'xavier'
local model_new = require('weight-init')(model, method)
```

Different initiation schemes can be tried using the above sample code. For heuristic, method = 'heuristic'

Cost Function

The cost function used here is the **Negative Log Likelihood** function. This function can be defined as

$NLL(\theta, D) = -\sum_{i=0}^{\{D|} logP\left(Y = y^{(i)} \middle| x^{(i)}, \theta\right)$

Where $\theta = Parameter$

x = input

D = Dataset

This cost function is minimized in by computing the gradient of loss function with respect to the parameter θ . The log function helps in computation as it minimizes the scale of the cost function returned.

The cost function coupled with LogSoftMax function can be called the cross-entropy.

The 'nn' package includes the Negative Log Likelihood cost function and can be instantiated using the following command

Criterion = nn.ClassNLLCriterion()

The *nn.ClassNLLCriterion()* has inbuilt functions to compute the loss of the output of a network with respect to a target, and methods for back propagating the gradient with respect to the parameters and updating the weights.

Mini-Batch Stochastic Gradient Descent

Mini-Batch gradient descent implements Stochastic Gradient Descent in small batches of 10, 64 100, etc.

Gradient Descent is used to minimize the cost function with respect to the parameters. In this methods, we change the parameters along the negative gradient direction .

$w = w - \gamma dw$

$\gamma = Learningrate$

Typically, the learning rate is chosen to be a small value so as the converge to the minima of the Cost function.

Momentum Update

The momentum update is a modification of the Gradient Descent method. Here, we accumulate a vector v which points to the direction of the negative gradient, damped by a factor. The accumulated value is then added to the weight. The momentum update helps in faster convergence. And has an additional hyperparameter denoted by the damping factor

$$v = \mu v - \gamma dw$$

$$w = w + v$$

Where μ refers to the damping factor and is typically set to 0.99

In addition, we also have a parameter called Learning Rate Decay and Weight Decay

The **Learning Rate Decay** refers to the Decay in the learning rate over a number of epochs. This is similar to annealing where the Learning Rate is gradually decreased as the number of epochs is reduced. This is implemented as follows

$$\gamma = \frac{\gamma}{1 + nEvals * LearningRateDecay}$$

Weight Decay is a regularization parameter that is added to the weights to prevent overfitting

The ‘optim’ package was used to implement the Stochastic Gradient Descent to minimize the cost.

optim.sgd() takes in 3 arguments, the first one is a function that returns two arguments, one being a loss vector and the other being a linear vector of GradParameters. The second argument for sgd is a linear vector of all parameters of the network.

The configuration of the sgd trained in this assignment is a table with the following parameters

```
config = {
learningRate = 0.015
learningRateDecay = 1e-1
momentum = 0.99
weightDecay = 1e-2
}
```

Other minimization methods that were carried out were AdaGrad and Adam

Implementation

Creating LeNet-5 architecture

The LeNet-5 architecture was implemented in Torch using the ‘nn’ library.

A sequential model is first instantiated. This method lets one add layers sequentially to the network

network   = nn.Sequential()

A convolutional layer is added using the nn.SpatialConvolution() method with arguments (1,6,5,5) to represent 1 input channel, 6 output channels, and 5 x 5 filter. Here padding is zero.

network:add(nn.SpatialConvolution(1, 6, 5, 5))

Similarly, a ReLU layer can be added using the nn.ReLU() method as follows

network:add(nn.ReLU())

Max-Pooling layer is invoked using nn.SpatialMaxPooling() with arguments(2,2,2,2) to denote a 2 x 2 maxpooling with stride 2

net:add(nn.SpatialMaxPooling(2,2,2,2))

A Fully connected linear layer is added by calling the nn.Linear() method, which takes in parameters as number of inputs and number of output neurons respectively. In the first Fully Connected layer for example, the number of neurons is 120. Thus A fully connected layer can be added as follows

*network:add(nn.Linear(16*5*5,120)* to represent 16*5*5 input weights to 120 output neurons

In this way, the various layers of the LeNet-5 architecture can be added to create a network graph.

Performance Evaluation

Performance Evaluation was done based on Accuracy and mean Average Precision. Both these metrics are explained below

Accuracy: Accuracy is defined at the number of correct predictions obtain across all the samples.

$$\text{Accuracy} = \frac{correctPredictions}{totalnumberofsamples}$$

The “optim” package is used for obtaining the confusion matrix and calculating accuracy

optim.ConfusionMatrix() accepts a table of class labels . i.e 10 class labels for MNIST

Thus a confusion matrix tensor is initiated using the following line of command

```
confusion = optim.ConfusionMatrix(classes)
```

For every batch of the minibatch gradient descent, we add the predicts and targets to the confusion matrix and update the confusion matrix. The following two commands execute those two operations.

```
confusion:batchAdd(output,target)

confusion:updateValids()
```

Accuracy is then obtained after every epoch using the *[self].totalValid* object of the confusion matrix.

```
Accuracy = 100*confusion.totalValid
```

Mean Average Prediction: Mean average precision is described as the sum of number of correct predictions upon the number of samples seen.

The value is divided by the total number of correct samples to obtain the mAP.

mAP has a range of [0,1] with 1 being the best performance and 0 to be poor performance.

Implementation:

The output predictions of the test batch are first sorted in descending order based on the confidence score.

```
confidences, indices = torch.sort(output, true)
```

The argument “True” denotes descending order. Thus the *indices* object is a tensor with the first element of every row containing the predicted label.

Thus the value of the first column of the indices object is compared with the target label using the *torch.eq()* method.

The *torch.eq()* returns a binary vector with 1 denoting a match and 0 denoting a mismatch.

```
prediction = torch.eq(indices[{ {},{1}}], target)
```

Thus for every batch, we have a binary prediction vector of number of true predictions. The mAP is then evaluated as follows

```
For every element in prediction{
Check if element is correctly predicted{
    If yes, then
    correct+=1
    AP+= (correct/index)
}
mAP+ = AP/correct
```

Training

For training, we perform the following operations

```
Repeat for 30 epochs{

    Repeat for entire dataset{

            • Load batch size of 128 training samples and labels
            • Zero all Gradient parameters
            • Forward propagate the input and save the output
            • Compute loss between the output and target label
            • Accumulate the loss over all batches
            • Add the output and target labels in confusion matrix
            • Compute gradient of loss w.r.t parameters over all layers on the network
            • Backpropagate the gradients and update the weights
            • Use SGD to minimize the loss function

        }

    Display the accuracy after the current epoch training

}
```

Testing

For testing, the following procedure is followed

```
Repeat for entire testing dataset{

        • Load batch size of 128 testing samples and labels
        • Forward propagate the input and save the output
        • Compute the loss between the output and target label
        • Accumulate loss over all batches
        • Add the output and target tables in the confusion matrix

}

Display the accuracy
```

1.3. Experimental Results

The first 12 samples of the MNIST dataset is shown below

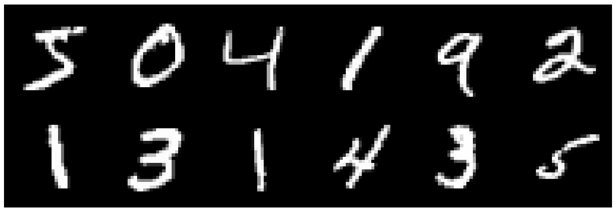

Figure 6: First 12 Samples of MNIST Dataset

The detailed list of parameters chosen to train the MNIST dataset using LeNet-5 architecture is as follows.

Parameter Setting 1

Initialization Scheme: **Random Initialization**

Number of Epochs: 30

Batch Size = 128

SGD parameters:	Learning Rate = 0.015
	Learning Rate Decay = 0.1
	Momentum = 0.99
	Weight Decay = 0.01

Training Sample size = 60000

Testing Sample size = 10000

Testing Accuracy = 98.94%

Training Accuracy = 98.81%

mAP = 0.991

The Plot of Epoch – accuracy is for this parameter setting is shown as follows

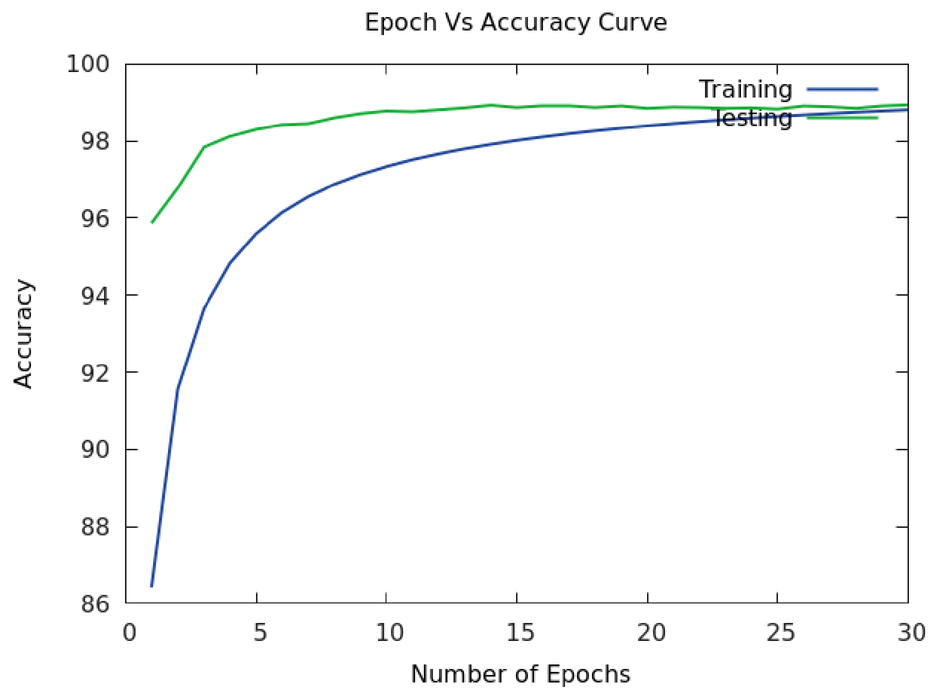

Figure 7: Epoch-Accuracy Plot for Parameter Setting 1

Parameter Setting 2

Initialization Scheme = **Heuristic Initialization**

Number of Epochs = 30

Batch Size = 128

SGD parameters:	Learning Rate = 0.015
	Learning Rate Decay = 0.1
	Momentum = 0.99
	Weight Decay = 0.01

Training Sample size = 60000

Testing Sample size = 10000

Testing Accuracy = 98.72%

Training Accuracy = 99.03%

mAP = 0.988

The Plot of Epoch – accuracy is for this parameter setting is shown as follows

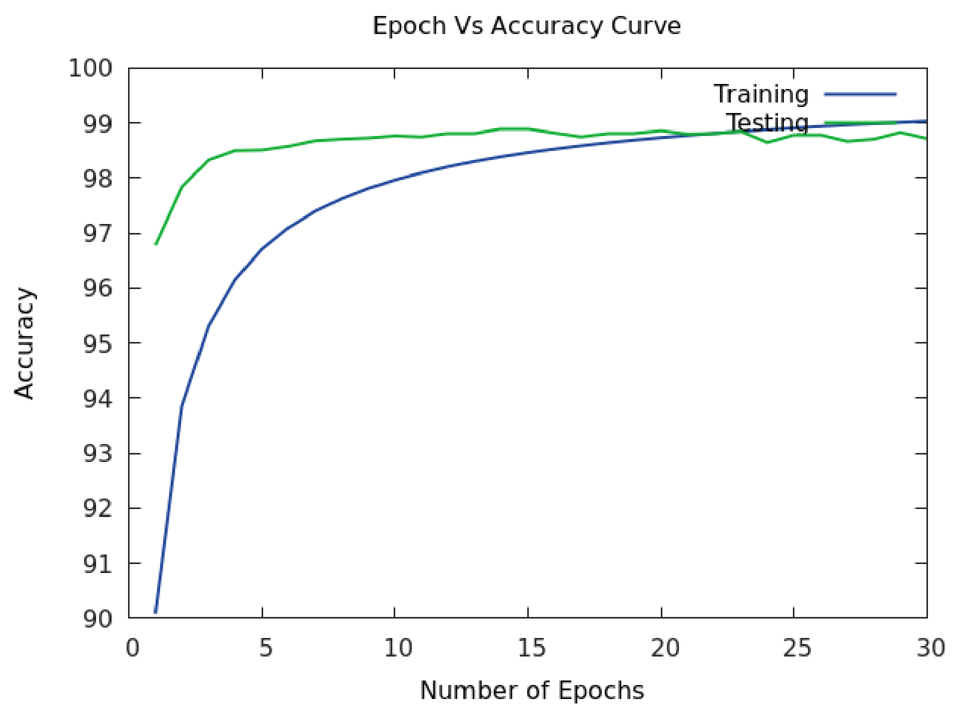

Figure 8: Epoch-Accuracy Plot for Parameter Setting 2

Parameter Setting 3

Initialization Scheme = Heuristic Initialization

Number of Epochs = 60

Batch Size = 64

SGD parameters:	Learning Rate = 0.01
	Learning Rate Decay = 0.1
	Momentum = 0.9

Training Sample size = 60000

Testing Sample size = 10000

Testing Accuracy = 99.05%

Training Accuracy = 99.72%

mAP = 0.993

The Plot of Epoch – accuracy is for this parameter setting is shown as follows

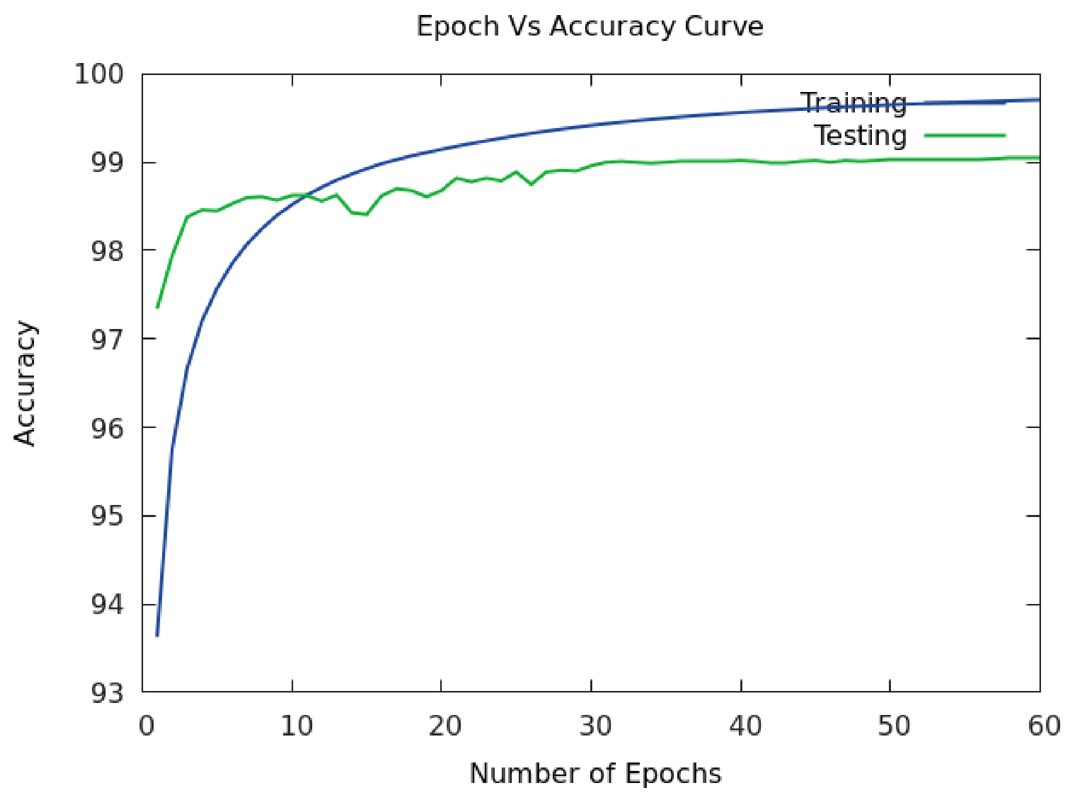

Figure 9: Epoch-Accuracy Plot for Parameter Setting 3

1.4. Discussion

It can be seen that the LeNet-5 architecture performs well in recognizing the Hand-written digit dataset by MNIST with the average testing accuracy using the three parameter settings mentioned above to be **98.79%**

By the design of the CNNs, the first Convolutional layer obtains simple features from the dataset. These can be edges, contours, colour values etc. The higher convolutional layers learn more complex geometry and textures. This is similar to the Law's filter architecture for texture classification except the filters learn any feature that gives improved classification.

The first Convolutional layers sees only a 5x5 window at a time within the larger 32 x 32 MNIST image. This gives localized features and are mostly constrained with thresholds and edges. However, the second Convolutional layers observes a larger 10 x 10 window of the original 32 x 32 image. This is because of the Max Pooling layer that helps to reduce variance and throws out back information that is not important to the network.

The weights and output of the first convolutional layer are shown below

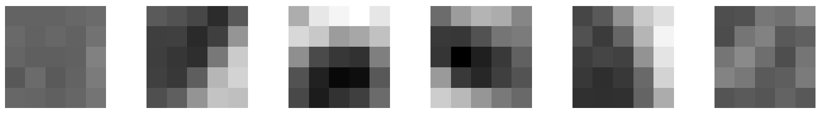
Figure 9: Learned Weights of the 1st Convolutional Layer

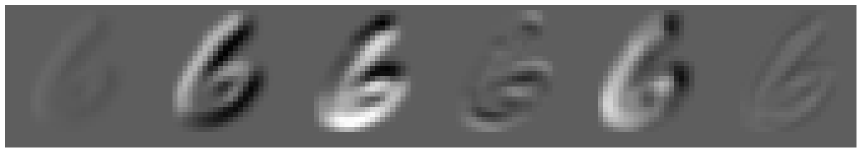
Figure 10: Output of the 1st Convolutional Layer

Thus, the second convolutional layers detect and learns more complex curves, tangents, shapes and contours which make the essential part of the hand written digit classification.

The output of the 16 filters of the second convolutional layer is shown below

Figure 11: Output of the 2nd Convolutional Layer

Further, three parameter settings were tested in this project and all three settings performed well on the MNIST dataset but with some variations.

The first parameter setting uses **Random Initialization** which is explained in section 1.2 and gives a good testing accuracy of **98.94%.** This is similar to the results obtained in [4] and has an **mAP** of **0.991** which is the highest among all three parameter settings.

The higher **momentum of 0.99** helps in good convergence of the test accuracy within 30 epochs itself. This was observed to give accelerated performance when compared to using a smaller momentum as in the case of the 3rd parameter setting.

The weight decay prevents overfitting and thus the performance of testing and training accuracies are very similar with no large variance. In a direct comparison with the parameter setting 3, we see that removing weight decay causes the network to over fit on the training set and thus the accuracy difference between the training and testing sets is higher.

The second parameter setting differs from the first with only respect to the initialization scheme. The second parameter setting uses the **Heuristic Initialization** scheme mentioned in [3]. It can be observed that this initialization scheme for the weights gives a better starting scheme for the weights as the training accuracy starts from **91%** as compared to the 1st parameter setting at **87%**. Further, the crossover point between the training and testing accuracy plots occur much earlier with this initialization scheme as compared to the first parameter setting. However, the Heuristic Initialization scheme gives a poorer Testing accuracy of **98.72%** and an **mAP of 0.988** which is lower to the first parameter setting. This can be attributed to overfitting as 30 epochs training might have over fitted the network to the training set as shown by the higher accuracy on the training set at **99.03%**

The third parameter setting removes the weight decay, uses a slightly lower momentum of **0.9** and uses a smaller batch size.

This parameter setting performs the best on the testing set with an accuracy of **99.05%.** The mAP is highest among all three settings at **0.993** denoting the precision is higher as compared to the first parameter setting. Further the absence of weight decay and lower momentum requires the network to be trained for more epochs to reach good accuracy.

b) Difference between LeNet-5 and classic 3 layer ANN

In a classic ANN network, the spatial similarity of regions within the image is ignored. This is because each pixel of the image is connected to a neuron unit which learns a weight parameter for each pixel across all channels in the image. This leads to a burst in the number of parameters and furthers leads to a lot of over-fitting as the spatial similarity of pixels is lost.

In contrast, CNNs have 3 D volume of weights, with each 2D layer acting as a filter and operates on a window region on the input image across all the channels. Further these weights are shared over the entire image and thus this reduces the number of parameters to be learned.

The LeNet-5 also includes max-pooling layers which helps to remove variance information, reduce the number of parameters learned further and also helps scan a larger sub region of the image for features.

The LeNet-5 performs much better than ordinary ANN network for hand-written digit classification.

c) Why LeNet-5 works better for the handwritten digit classification problem as compared with ANN?

The LeNet-5 works better than ANN because it learns features that are most relevant to the hand-written digit classification problem. The weights learnt in the first convolutional layer are edges which describe the particular number and the more complex curves in digits like 5, 8, 9 etc are learnt in the Conv2 layers. More information is learnt by observation a region than pixel-wise observation as done in ANN.

Further, LeNet-5 is robust to overfitting as more irrelevant information are removed using the pooling layers. The LeNet-5 then sends the most using features to a 3 layer ANN which contributes to high accuracy.

Problem 2: Capability and Limitation of Convolutional Neural Networks

2.1 Abstract and Motivation

The problem 1 showed that the LeNet-5 performed well for the MNIST dataset which consists of white digits on a black background. However, this is the most typical case, and the digit information is should be robust to change in the colour of the digit, translation of the number or presence of other backgounds. In this problem the performance of LeNet-5 is studied on all the three cases, and modified networks and solutions are tested to make LeNet-5 more robust to these variances. Particularly it was observed that the LeNet-5 performs poorly on inverted MNIST dataset when trained on original MNIST dataset. Further, the accuracy drops when a background information is added to the MNIST dataset.

The LeNet-5 is also translation invariant for small pixel translations but the accuracy drops as the translation is increased.

2.2. Approach and Procedure

a) Application of LeNet-5 to Negative MNIST Images

The model trained using the Parameter settings 1 and 2 on the MNIST dataset was tested on the negative MNIST testset and the accuracy and mAP have been reported. The negative MNIST was obtained as follows.

testset.data = 255 – testset.data

Designing architecture to improve accuracy

Two approaches were carried to improve the performance of LeNet-5 on Negative MNIST

Approach 1 – Modified LeNet-5

In the first approach, from [4] it was noted that multiplying the weights of the first convolutional layer by -1 makes the network to perform well with inverted MNIST. Thus, this was testing by using the following code.

```
net:get(1).weight:mul(-1)
```

The above command multiples the weights of the first convolutional layer by -1 while leaving the rest of the network the same. The performance of the network improved drastically inline with [4] and the results are shown in section 2.3.

Thus, given the information that multiplying the weights by -1 helps to detect the negative MNIST, the number of filters in the 1st Convolutional layer is expanded from 6 to 12 as per [4]. The first 6 2D filters have the same weights as before, whereas the next 6 layers have the negative of the first 6 layers weights.

For the bias, the learned bias terms of the first 6 layers were copied for the next 6 layers.

Thus the modified network now has 12, 2D filter weights and 12 bias terms. The output of the first convolutional layer is give 12 output images of dimensions 28 x 28.

The second Convolutional layer now requires 12 filter weights instead of the original 6 to operate on all 12 channels of the input image. The first 6 weights of the convolutional layer 2 were replicated and concatenated with the original 6 filter weights to obtain 12, 2D filter layers.

The number of parameters now in the Conv2 layer is 16 x 12 x 5 x 5 instead of 16 x 6 x 5 x 5.

The following code was used modify the architecture of the network.

```
C1_weights = net:get(1).weight:clone()
C1_bias = net:get(1).bias:clone()
C1_weights:mul(-1)
net:get(1).weight = torch.cat(net:get(1).weight,C1_weights,1)
net:get(1).bias = torch.cat(net:get(1).bias,C1_bias,1)
C2_weights = net:getParameters(4).weight:clone()
net:get(4).weight = torch.cat(net:get(4).weight,C2_weights,2)
```

The performance on both original and Negative MNIST dataset improved considerably when tested on these two new networks

Approach 2 – Training with 50% Negative MNIST samples

In this approach, the first 30000 samples of the training set were inverted and the network was training using the parameter setting 1. The following code was used to accomplish this.

```
trainset.data:sub(1,30000):mul(-1):add(255)
```

sub(start,end) references the first 30000 samples in trainset.data which are then multipled by -1 and then added to 255 to produce the inverted dataset.

The network was trained for 15 epochs when the parameter setting 1 with random initialization and the network was tested on both MNIST and negative MNIST and the accuracy and mAP have been reported. The network was then trained for 15 epochs using the parameter setting 1, with batch size of 128
SGD parameters

```
config = {
learningRate = 0.015
learningRateDecay = 1e-1
momentum = 0.99
weightDecay = 1e-2
}
```

b) Application of LeNET-5 to MNIST Images with Background

In this problem, the MNIST images overlaid with background scenes were used to train the LeNet-5, however this dataset has 3 input channels and 11 classes including a class that contains no digit. The network was hence modified as follows.

The first Convolutional layer now have 3 input channels, and thus the first argument of the Spatial Convolutional method must be 3 instead of 1.

```
net:add(nn.SpatialConvolution(3, 6, 5, 5))
```

The last classification later of LeNet-5 has 11 output neurons instead of 10.

```
net:add(nn.Linear(84, 10))
```

Further, the optim.ConfusionMatrix(classes) method requires a table of 11 classes. Thus the following table was passed as argument which has 11 labels.

```
classes = {'0','1','2','3','4','5','6','7','8','9',’10’}
```

The network was then trained for 15 epochs using the parameter setting 1, with batch size of 128
SGD parameters

```
config = {
learningRate = 0.015
learningRateDecay = 1e-1
momentum = 0.99
weightDecay = 1e-2
}
```

The performance of the network for this dataset has been reported in section 2.3.

c) Checking Translation Invariance of LeNet-5

In this problem the digits in the MNIST test dataset were translated by *x* and *y* pixels in the horizontal and vertical directions, and were used to test the trained network on non-translated MNIST dataset.

The function *TranslatedDataset(images)* takes in a dataset and returns a translated dataset of same size but shifted by random combination of pixels between range (-a,a) in both x and y directions, where a an integer.

```
function TranslatedDataset(images)
        for i = 1,images:size(1) do
                images[i] = image.translate(images[i],torch.random(-6,6),torch.random(-2,2))
        end
        return images
end
```

The *image.translate()* method accepts 3 parameters, the first being an image tensor, the second and third are the translations in horizontal and vertical directions.

torch.random(-a,a) returns a random integer between integers -a and a. Thus a random combination of translated images is obtained when are then used to test the network.

Methods to make LeNet-5 Translationally invariant

It was observed that the LeNet-5 is not translationally invariant when the pixels were shifted by a large amount. To counter this problem two approaches were tried.

Approach 1 – Increasing the Max-Pooling window size

Consider a white pixel in the MNIST image which represents a pixel in the digits. For every such pixel, there are 8 directions in which one can translate the input image by a single pixel. Thus when as 2 x 2 max pooling function is applied over it, there are 3 out of 8 possible locations where this pixel can be shifted to when 1-pixel translation is done. Thus 3/8 of configurations will give the same output at that convolutional layer.

However, when a 3 x 3 max pooling window is considered, there are 5 possible combinations that will produce the same result for a translation of 1 pixel. Thus increasing the max-pooling will help in making the network more robust to translation.

Here, a max-pooling layer of 3 x 3 window with a stride of 2 pixels were used. This was done against a stride of 3 pixels as the number of inputs to the 1st dense layer would be just 16 which degrades results.

```
net:add(nn.SpatialMaxPooling(3,3,2,2))
```

For this parameter setting, the number of inputs to the first fully connected layer is 16* 4 * 4 thus this was modified in the architecture.

```
net:add(nn.View(16*4*4))
```

```
net:add(nn.Linear(16*4*4, 120))      -- the first fully connected layer
```

This network was trained with parameter setting 1 for 30 epochs with random initialization. The parameters for SGD is mentioned below

```
config = {
learningRate = 0.015
learningRateDecay = 1e-1
momentum = 0.99
weightDecay = 1e-2
}
```

The network was then tested on testing data that was translated in x and y direction by a number of pixels.

Approach 2 – Training on translated and non-translated MNIST dataset

In this approach the original LeNet-5 was trained using both translated and non-translated MNIST dataset for 15 epochs. The parameter set 1 was used to train the network with random initialization

The trained network was then tested on both translated and non-translated test datasets, and the performance has been tabulated.

2.3. Experimental Results

The first 12 inverted images of the MNIST dataset is shown below

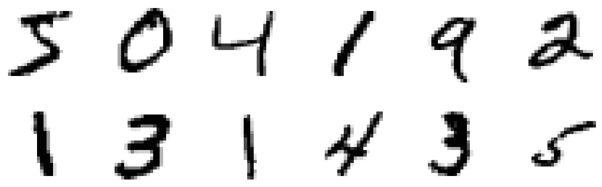

Figure 12: First 12 samples of Inverted MNIST

a) Application of LeNet-5 to Negative MNIST Images

The performance of the LeNet-5 trained using the parameter setting 1 and 2 on Negative MNIST is shown below

Parameter Setting	Test Accuracy	mAP
1 (Random Initialization)	45.72%	0.482
2 (Heuristic Initialization)	47.95%	0.499

Table 1: Performance of LeNet-5 on Negative MNIST

Approach 1 – Modified LeNet-5

Multiplying Conv1 weights by -1

The performance of the above LeNet-5 network when the first convolutional layer weights are multiplied by -1 is shown below.

Parameter Setting	Test Accuracy on Negative MNIST	mAP
1 (Random Initialization)	95%	0.95
2 (Heuristic Initialization)	97.86%	0.981

Table 2: Performance of LeNet-5 on Negative MNIST when Conv1 weights are multiplied by -1

Performance on expanding the number of filters in Conv1 to 12 as explained in Section 2.2.

Parameter Setting	Test Accuracy on MNIST	mAP - MNIST	Test Accuracy on Negative MNIST	mAP – Negative MNIST
1 (Random Initialization)	95.43%	0.958	58.7%	0.605
2 (Heuristic Initialization)	97.23%	0.973	73.25%	0.747

Table 3: Performance of modified LeNet-5 on Negative MNIST as proposed by [4]

Approach 2 – Training with 50% Negative MNIST samples

The performance of the LeNet-5 with the parameter setting 1 with 50% negative MNIST samples during training is shown below.

Parameter Setting	Train Accuracy on MNIST	Test Accuracy on MNIST	mAP - MNIST	Test Accuracy on Negative MNIST	mAP – Negative MNIST
1 (Random Initialization)	96.04%	98.46%	0.985	95.78%	0.958

Table 4: Performance of LeNet-5 when 50% of training samples are inverted

The plot of training and testing accuracy for the aforementioned approach using MNIST dataset is shown below.

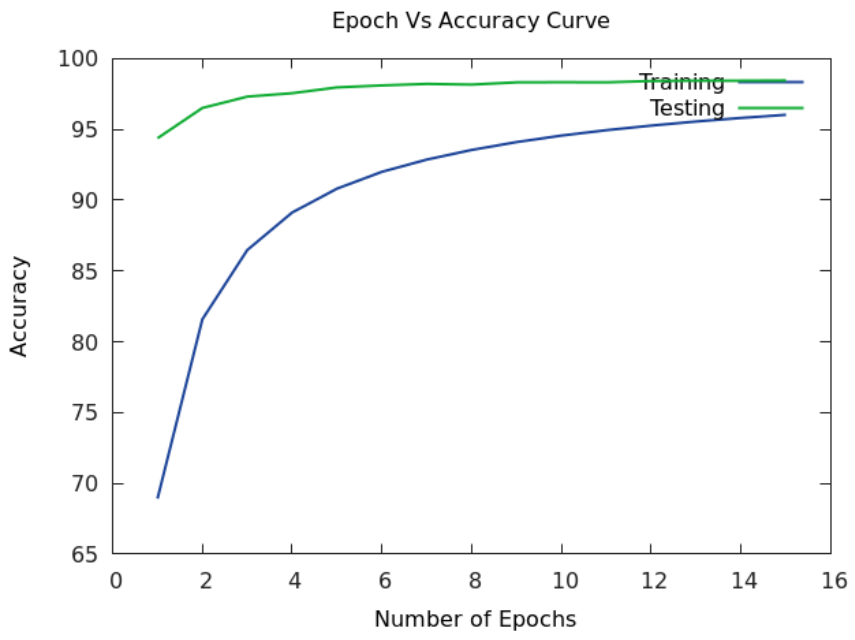

Figure 13: Epoch-Accuracy plot when trained with 50% inverted trainset

b) Application of LeNET-5 to MNIST Images with Background

The first 12 samples of the modified MNIST dataset is shown below

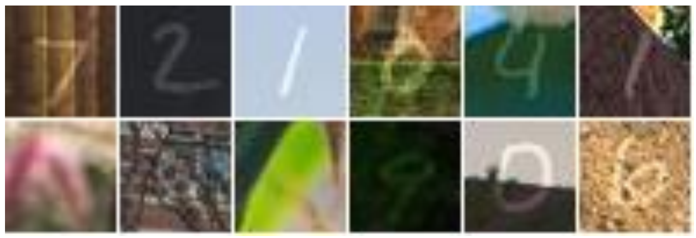

Figure 14: First 12 samples of modified MNIST dataset

The performance of the LeNet-5 with background images using parameter setting 1 is shown below Training was performed for 15 epochs.

Parameter setting	Training Accuracy	Testing Accuracy	mAP
1(Random Initialization)	78.65%	80.31%	0.808

Table 5: Performance of LeNet-5 on MNIST with Backgound

Shown below is the Confusion Matrix obtained on testing.

c) **Checking Translation Invariance of LeNet-5**

Some of the translated data generated is shown in the image below

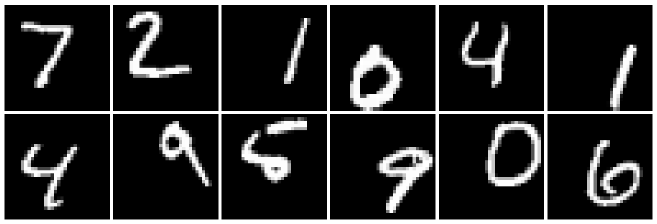

Figure 15: Examples of generated translated dataset

Shown below is the performance of the LeNet-5 on translated dataset using the networks trained on parameter setting 1 and 2 respectively.

Pixel Translation range in x and y directions. [-a,a]	Test Accuracy on network with parameter setting 1	Test Accuracy on network with parameter setting 2
[-1,1]	97.76%	97.58%
[-2,2]	94.11%	94.18%
[-3,3]	84.73%	83.76%
[-4,4]	69.92%	68.57%
[-5,5]	55.97%	54.31%
[-6,6]	45.29%	43.96%

Table 6: Performance of LeNet-5 on Translated MNIST Dataset

It can be seen above that the accuracy drops with increase in translation

Approach 1 – Increasing the Max-Pooling window size

The performance of the modified network with max-pooling window of 3x3 with stride 2 is shown below. Both networks were trained for 30 epochs

Pixel Translation range in x and y directions. [-a,a]	Test Accuracy on network with parameter setting 1	Test Accuracy on network with parameter setting 2
[-1,1]	98.2%	98%
[-2,2]	95.38%	95.72%
[-3,3]	88.17%	87.28%
[-4,4]	75.16%	72.8%
[-5,5]	61.91%	59.28%
[-6,6]	50.27%	47.9%

Table 7: Performance of LeNet-5 with 3x3 max-pool on Translated MNIST

Thus, a jump in accuracy can be seen when a larger max-pooling window is used.

Approach 2 – Training on translated and non-translated MNIST dataset

The LeNet-5 was trained for 15 epochs using training data that contained translated and non-translated images. The maximum translation in the pixels in the training data was 6. The parameter setting 1 was used to train the network.

The test data also includes translated images with translation within the range of [-6,6] pixels.

Parameter Setting	Training Accuracy	Test Accuracy
1 (Random Initializing)	94.32%	95.94%

Table 8: Performance of LeNet-5 when trained on translated Dataset

2.4. Discussion

a) Application of LeNet-5 to Negative MNIST Images

Table 1 shows the performance of the LeNet-5 architecture when testing on Negative MNIST dataset. The performance is poor on Negative MNIST and it can be seen that the network trained with Heuristic Initialization gives better Accuracy and mAP of 47.95% and 0.499 as compared to the network trained using the Random Initialization. This performance is also superior to the result obtained in [4] and thus the SGD parameters chosen in the parameter setting 1 and 2 are more efficient.

Approach 1 – Modified LeNet-5

In order to improve upon the performance on Negative MNIST dataset, the procedure followed in [4] was performed. Specifically, the weights of the first convolutional layer were inverted and better performance were observed while testing in Negative MNIST dataset. **Table 2** shows this performance.

Why this works?

An intuitive explanation is as follows. Figure 9 shows the weights learnt by the Convolution 1 layer. It shows the gradient information at various angles which produce different activation pattern for different digits. This pattern of weights is learnt for the original MNIST dataset which consists of white digits in black background. Thus at the edges of the digits there will be a gradient change from dark to white and white to dark respectively.

When negative MNIST is tested, the gradient change in inverted, i.e the gradient changes from white to dark and dark to white respectively. This represent an inversion of the sign of the weights as what was once white is now dark and vice versa. Thus, the inversion of the sign of the weights in the first convolutional layer helps produce good activation on negative MNIST.

Further, **Table 3** shows the performance obtained when the modified LeNet-5 with 12 filters in the Conv1 layer is used as proposed in [4]. The first 6 have the same weights but the next 6 have the weights of the first 6 layers multiplied by -1. Thus, when the MNIST dataset is forward propagated through this network, the first 6 filter outputs of the Conv1 produce high outputs whereas the last 6 do not get activated much. When the negative MNIST is forward propagated through this network, the last 6 filter output show high activation whereas the first 6 do not.

Table 3 shows that the modified network gives a good performance on MNIST of about **97.23%** on testdata and gives **73.25%** test accuracy when tested on Negative MNIST data. This also shows that the Heuristic Initialization scheme proposed in [3] gives a good performance on MNIST as well as Negative MNIST datasets. The performance of the 12 Layer network is superior to ordinary LeNet-5 architecture.

Approach 2 – Training with 50% Negative MNIST samples

The second approach adds 50% negative MNIST data to the training set, and the LeNet-5 is used to train using the 1st parameter setting. Now the network learns to recognize the digits in both backgrounds and features are learnt to recognize only the digits.

The newly trained network gives good accuracy on both MNIST and Negative MNIST datasets of **98.46%** and **95.78%** respectively. **Table 4** shows this result. This performance is expected as the network features that are common in both backgrounds.

b) Application of LeNET-5 to MNIST Images with Background

Table 5 shows the result obtained when the network is trained using the modified MNIST dataset that contain random background. It can be seen LeNet-5 does poorly in this case and the test accuracy saturates at **80.31%** with **mAP** of **0.808.** This is perhaps because the gradient information is too subtle in some samples which fails to give good activation after convolution.

Further, the background information contains edge patterns that get activated falsely when an edge filter is applied. This decays the result as it gives wrong activations cause false predictions.

c) **Checking Translation Invariance of LeNet-5**

CNNs are by nature translation invariant. This is because of the convolution operation in calculus is shift invariant. Further, if a max-pooling layer is present, it takes into consideration inherent translation in the image. This helps CNN be translation invariant.

Reason for decrease in performance

The LeNet-5 contains max-pooling layers which have a 2 x 2 window with 2-pixel stride. A 2x2 man-pooling layer is robust to translation of only 1 pixel. This is because for every pixel in an image, there are 8 neighborhood locations to which the pixel can be translated by 1 pixel. When a 2 x 2 max-pooling operation is performed, 3 out of the 8 neighbors have same output as before. So a 2x2 pooling gives the same output 3/8ths of the time.

This is reflected in **Table 6** in which a 1-pixel translation gives a good accuracy of about **97.76%,** however the accuracy drops with increasing translation. Thus, the LeNet-5 is translation invariant for up to 1 pixel.

The 2x2 max-pooling gets replaced by 3x3 max-pooling layer with a pixel stride of 2 pixels. Thus three is an overlap of 1 pixel. When a 3 x 3 max-pooling layer is used, 5 out of 8 combinations produce the same result as before with 1-pixel stride. Thus, this was tested and **Table 7** shows the result for using 3 x 3 max-pooling which was trained for 15 epochs.

Table 7 shows an average increase in accuracy of **3% - 4%** when tested with translated data, however the accuracy continues to drop if the object is not centered in the image.

Approach 2 – Training on translated and non-translated MNIST dataset

In this approach, training data consisted of both translated and non-translated data. The range of pixel translation is from [-6, 6]. Thus the LeNet-5 is then trained to learn these translated digits and the weights are tuned accordingly.

When tested on a testset which consisted of randomly translated dataset, it gave an accuracy of **95.84%** which is superior to the performance than approach 1.

Further, the trained network gave consistently good performance on datasets translated by 1, 2 ,3 ,4 pixels and the average accuracy was **97.3%.**

References

[1] Yann LeCun, Leon Botton, Yoshua Bengio, and Patrick Haffner, "Gradient-Based Learning Applied to Document Recogniton" proceedings of the IEEE 86.11 (1998): 2278-2324

[2] Online http://yann.lecun.com/exdb/mnist/

[3] Yann LeCun, et al, " Efficient BackProp"

[4] C.-C. Jay Kuo, "Understanding CNN with a mathematical model" 2016, arXiv 1609.04112.

[5] [online] http://cs231n.github.io/

[6] [online] http://deeplearning.net/tutorial/lenet.html

[7] Weight Initiation https://github.com/e-lab/torch-toolbox/tree/master/Weight-init