EE 569
PROJECT #4

BY

PRANAV GUNDEWAR

USC ID: 4463612994

EMAIL: gundewar@usc.edu

Table of Contents

PROBLEM 1 – CNN TRAINING AND ITS APPLICATION TO THE MNIST DATASET

CNN ARCHITECTURE AND TRAINING

Task: Explain the architecture and operational mechanism of convolutional neural networks

ABSTRACT AND MOTIVATION

Neural network is state of the art way to model a data set that mimics human intelligence in system. Its behaviour is analogous to the neurons present in the human brain. CNN has been developed to bypass tedious and time taking process of feature extraction in traditional machine learning and computer vision applications.

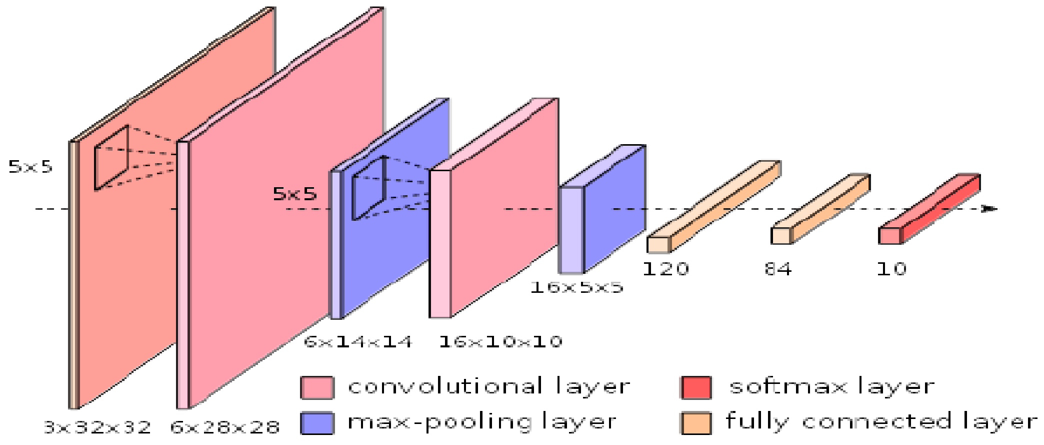


Figure 1 – Baseline CNN architecture derived from LeNet5

Convolutional neural network is a type of feed forward neural network where connectivity between layers is inspired by extremely complex animal visual cortex. Depending upon the architecture of network, neurons are connected to each other in certain manner. We need to give huge information which will help the system to gain from its complexities and remember the hidden patterns of the given information. After we train the network using input data, it can classify input data into respective categories.

APPRAOCH AND PROCEDURE

The general overview for basic operations of convolutional neural network is given below.

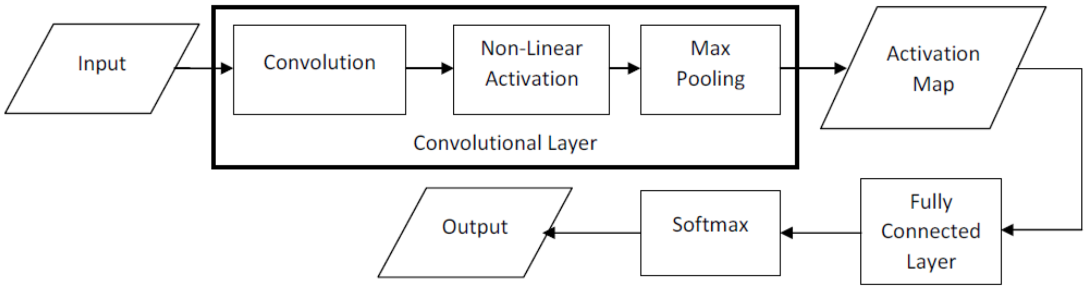


Figure 2 – Overview of CNN

Input for the system will be input image from particular train dataset. CNN has multiple layers through which layers are connected and function of each layers are explained in detail.

1. ***Convolutional Layer***

This layer forms the base of CNN, it is core building block. It performs core operations of training and firing neurons. While dealing with high dimensional input data, it is not possible to connect all neurons to the neurons in consecutive next layer. This layer has set of learnable filters of required depth but small spatial dimensions.

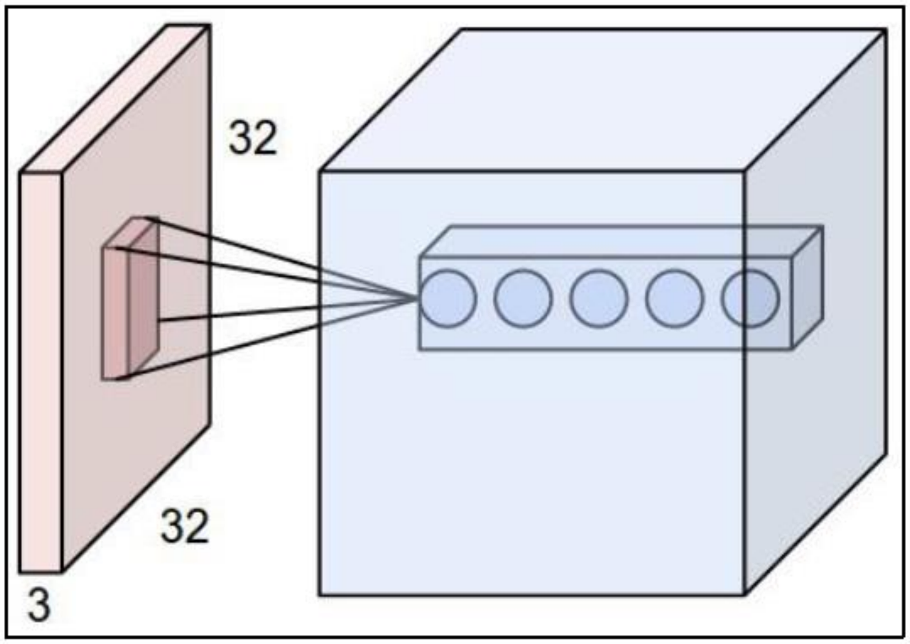


Figure 3 – Convolutional Layer and input image

A smarter approach would be to connect neurons in the next layer to local region of input. This can be visualized from figure 3 where all the neurons are connected from the local patch only. The input image is convolved with every filter across the dimensions of filter. Hence, for each filter a 2-D activation map is generated for every spatial position. Every convolutional layer has own 2-D activation map stacked along the depth and an output image is formed. CNN can have multiple convolutional layers which can be stacked.
Important parameters are-
Filter Size: Dimensions varied depending upon the application and time constraints.
Number of filters: We choose number of filters depending upon position of layer and application of CNN.
Stride: It is how much should each filter should shift for doing next convolution. We can reduce number of weights using this parameter.
Padding- We pad the image using valid (zero padding) or same (reflection padding) for the pixels around the edges.

2. ***Pooling Layer***

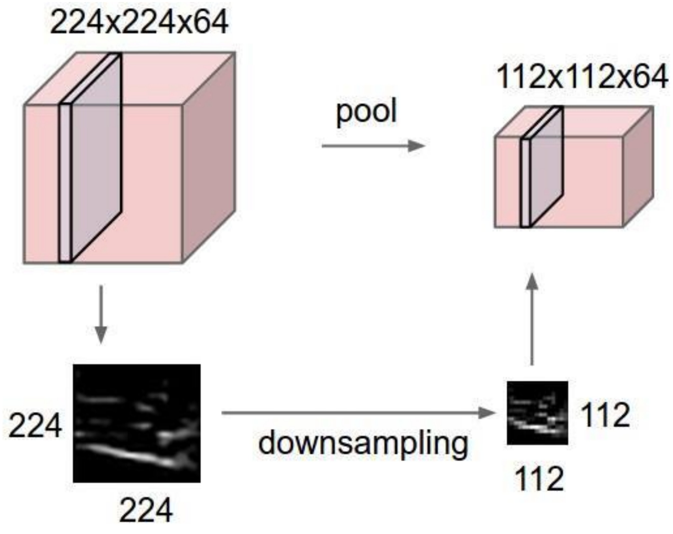


Figure 4 – Pooling Layer Overview

This layer usually subsamples the output of preceding layer. It is generally used after convolutional layer. Its significant capacity is to decrease spatial size of convolutional yield picture to diminish the quantity of weights and association with a specific end goal to lessen the danger of overfit. This layer operates independently on slice of every depth resizes it using max, min or average function in the window and it has no learnable parameters. It works using mathematical functions.

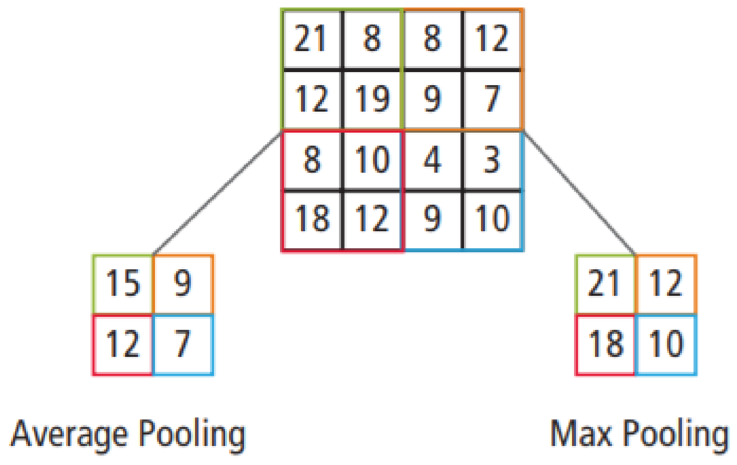


Figure 5 – Working of Max-Pooling

For a 2*2 window, we will discard about 75% activations output. The depth of the dimensions of input remains unchanged. It is non-linear down sampling to reduce the information and making it more robust. An example of max pooling with 2*2 filter and stride 2 is shown in figure 5. A compact portrayal of data is done in this layer.

Fully Connected Layer

This layer is usually the last layer in CNN architecture. This has full associations with every one of the activation in past layer. The combined activations can be calculated using matrix multiplication and using bias offset. It generally takes input from pooling layer and output N dimensional vector where N is number of classes that depicts probability of each class.

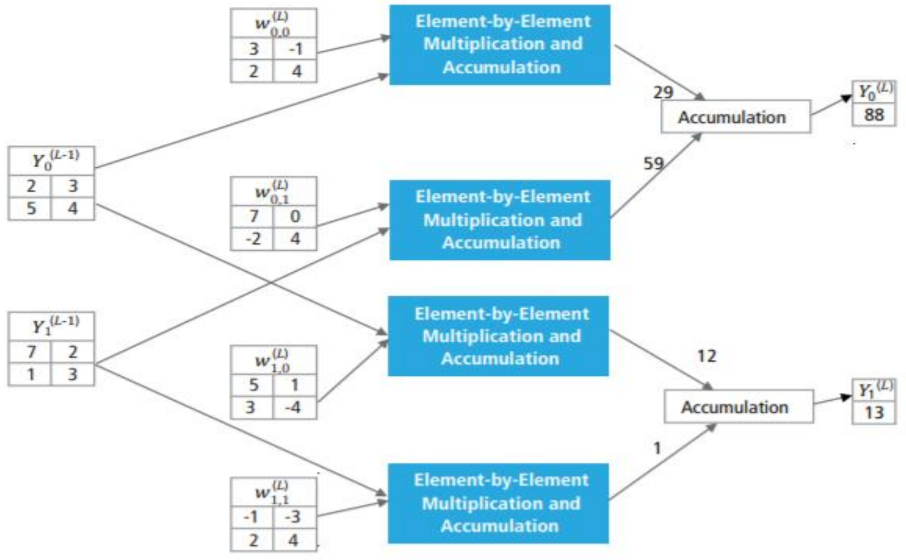

$w_{0,0}^{(L)}$
3 -1
2 4
$w_{0,1}^{(L)}$
7 0
-2 4
$w_{1,0}^{(L)}$
5 1
3 -4
$w_{1,1}^{(L)}$
-1 -3
2 4
$Y_0^{(L-1)}$
2 3
5 4
$Y_1^{(L-1)}$
7 2
1 3
Element-by-Element Multiplication and Accumulation
Element-by-Element Multiplication and Accumulation
Element-by-Element Multiplication and Accumulation
Element-by-Element Multiplication and Accumulation
29
59
12
1
Accumulation
Accumulation
$Y_0^{(L)}$
88
$Y_1^{(L)}$
13

Figure 6 – A fully connected layer

A fully connected layer checks which abnormal state highlights to the specific class and has the particular weights with the goal that we get probabilities for each class by figuring the result of weights and past layers.

3. ***The Activation Function***

The activation function of a node defined the output given set of inputs. It is non-linear function that allows us to solve difficult problems. Activation function enable us to present the non-linearity in the system. Non-linearity implies yield can't be precisely duplicated from direct mix of information. This function decided whether the neuron should get fired or not.

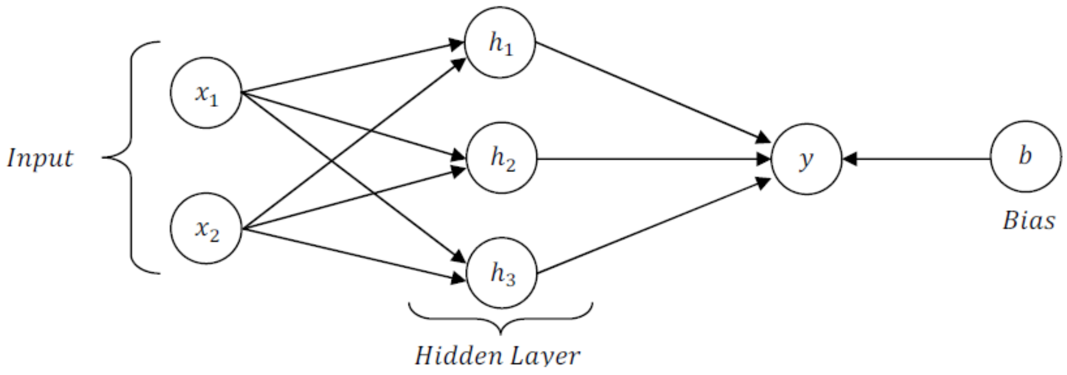


Figure 7 – A single hidden layer network

For a single hidden layer network,

$$h_1 = (x_1 * W_{x_1,h_1}) + (x_2 * W_{x_2,h_1})$$
$$h_2 = (x_1 * W_{x_1,h_2}) + (x_2 * W_{x_2,h_2})$$
$$h_3 = (x_1 * W_{x_1,h_3}) + (x_2 * W_{x_2,h_3})$$

Where $W_{i,j}$ are the weights between ith input layer and jth output layer. The final output ''y" will be

$$y = (h_1 * W_{h_1,y}) + (h_2 * W_{h_2,y}) + (h_3 * W_{h_3,y}) + b$$

Here, substituting h_1, h_2, h_3 in the equation above,

$$y = x_1\left(W_{x_1,h_1}W_{h_1,y} + W_{x_1,h_2}W_{h_2,y} + W_{x_1,h_3}W_{h_3,y}\right) + x_2\left(W_{x_2,h_1}W_{h_1,y} + W_{x_2,h_2}W_{h_2,y} + W_{x_2,h_3}W_{h_3,y}\right) + b$$

This is linear in x_1, x_2

$$y = x_1W_1 + x_2W_2 + b$$

Typically, in machine learning applications, there are lot of activation function that have been used

a) Step Function-

Activation function is based on threshold.

Activation function A = 1 if y > threshold,

0 otherwise

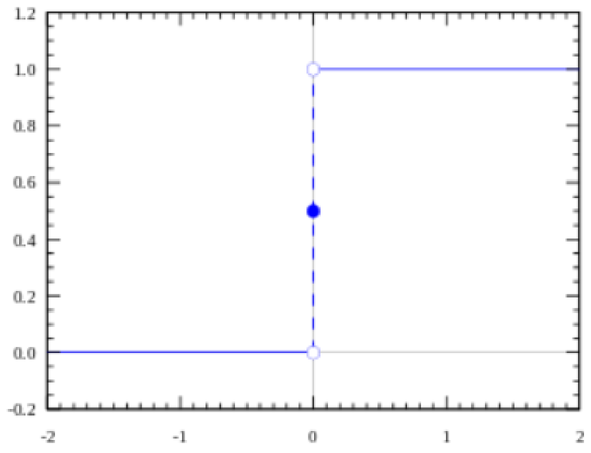


Figure 8 – Step Function

b) Sigmoid Function-

Any real valued function is converted into 0-1 using this function. It tends to bring activation to the either side of the curve. But, it has problem of saturation and vanishing gradients. For near horizontal part of the curve, slow convergence can be observed,

$$A = \frac{1}{1+e^{-x}}$$

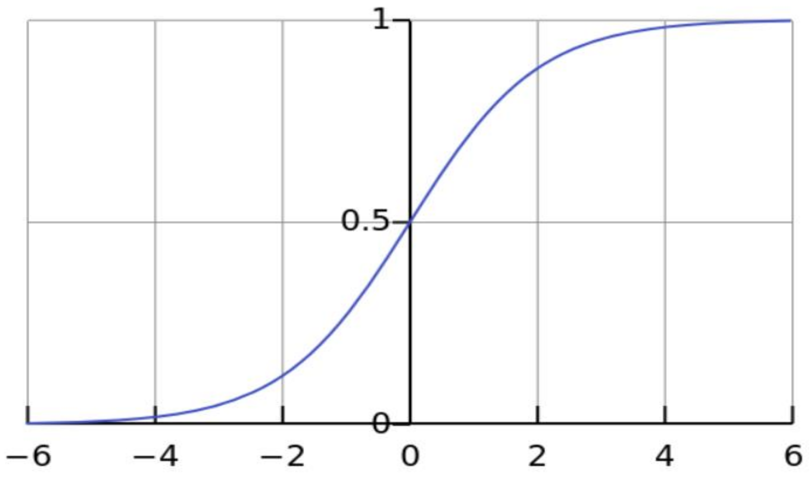
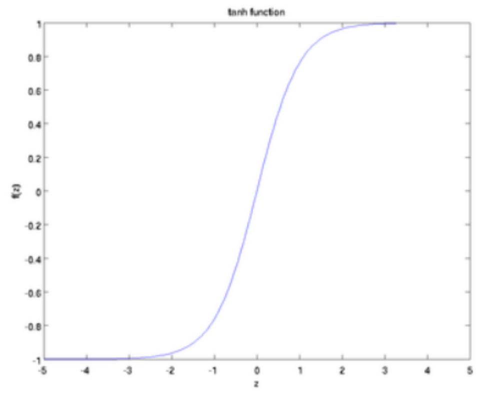


Figure 9 – Sigmoid Function

c) Tanh Function-

This is scaled sigmoid function. It has similar characteristics to sigmoid function. It is non-linear hence we can stack up the layers and it has range of -1 to 1. Gradient is stronger than sigmoid and derivatives are steeper. It also has vanishing gradient problems. It is very popular and widely used function.



$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

Figure 10 – Tanh Function

d) ReLu Function-

ReLu function gives an output equal to input when input is greater than 0 and 0 if it is negative.

$$A(x) = \max(0,x)$$

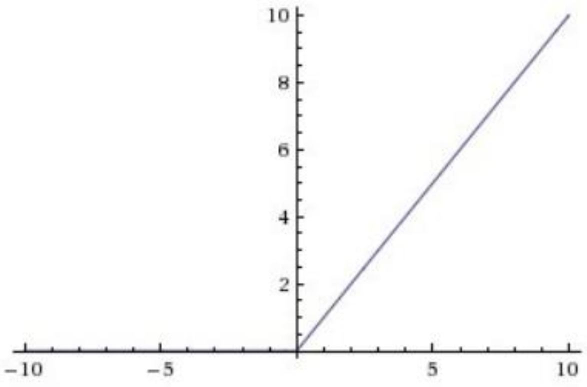


Figure 11 – ReLu Function

ReLu is non-linear in nature. It is very good approximator as any function can be approximated using this function. The range of ReLu is o to inf means it can blow up the activation. Ideally, we want fewer neurons to fire and making the activation proficient and sparse.

Because of characteristics or ReLu, almost 50% yields 0 activation. But assume horizontal line for negative part of x, gradient can go to zero. This means ReLu has dying problem as neurons which go into zero state become irresponsive to changes in input. Henceforth, the primary thought is to give gradient a chance to be non-zero and recoup amid in the long run. It is computationally inexpensive.

e) Leaky ReLu Function-

To address the dying problem of ReLu, leaky ReLu can be used. For the negative part of x, it will have a small gradient instead of zero shown in figure.

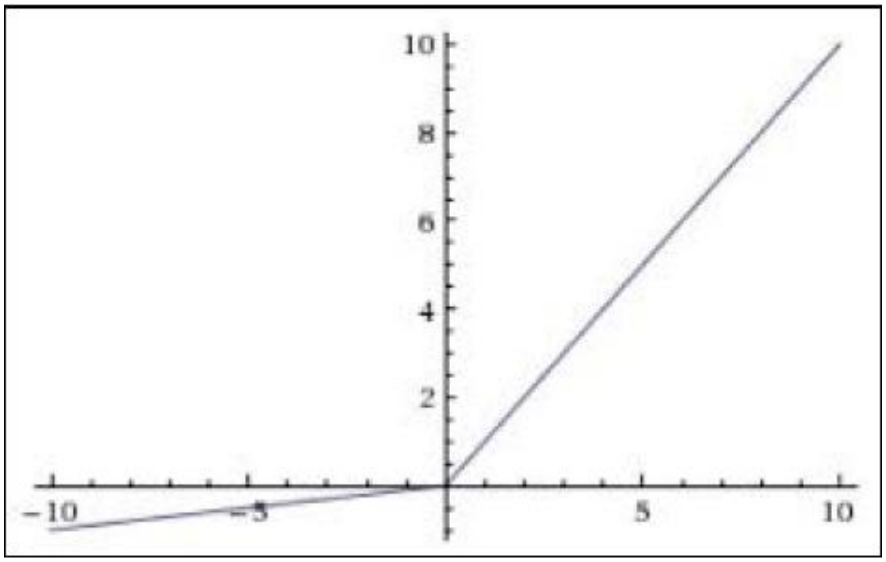


Figure 12 – Leaky ReLu Function

For negative part of x, it will have small constant A which makes sure Leaky ReLu will not saturate. The output is not zero centered and will converge faster.

f) Softmax function-

The softmax work is additionally a sort of sigmoid capacity however is helpful when we are attempting to deal with classification issue. It squeezes the output for each class into 0 to 1 and would also divide by the sum of outputs. This ultimately gives the probability of input being in each class. Basically, it is just the generalization of logistic regression.

$$y = softmax(evidence)$$

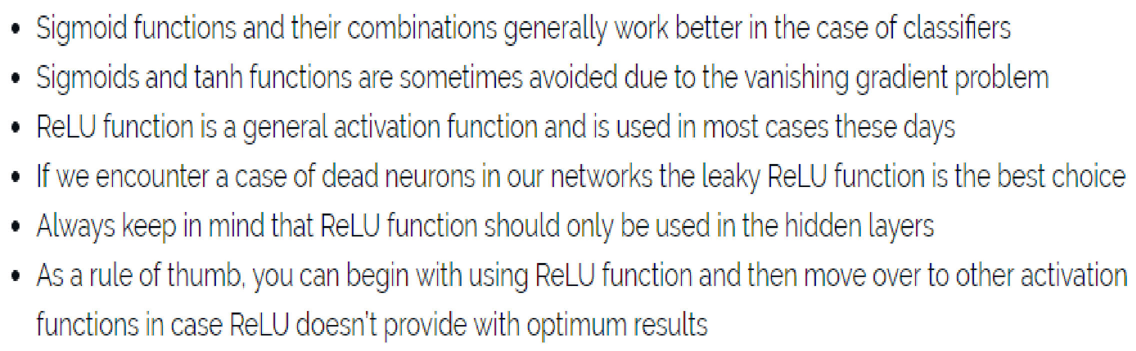
$$softmax(z) = normalize(\exp(z))$$

$$softmax(z) = \frac{\exp(z)}{\sum_j \exp(z_j)}$$

where $z = Wx + b$.

Choosing the right activation function-

Contingent on the properties of issue, we may have the capacity to settle on better decision of activation function for faster convergence.



- Sigmoid functions and their combinations generally work better in the case of classifiers
- Sigmoids and tanh functions are sometimes avoided due to the vanishing gradient problem
- ReLU function is a general activation function and is used in most cases these days
- If we encounter a case of dead neurons in our networks the leaky ReLU function is the best choice
- Always keep in mind that ReLU function should only be used in the hidden layers
- As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn't provide with optimum results

Figure 13 – Different activation Function

Overfitting Issue in Model Training-

Overfitting model alludes to show that it prepares the information too well. It, for the most part happens when model takes in the information detail and clamor too well to the degree that it hampers the execution of the model to the new data.

In statistics, overfitting is production of analysis that corresponds too closely to the training data and hence might not be able to fit the future data into trained system. An overfitted model contains more parameters that that can be justified by the data. Underfitting occurs when the network is not able to capture structure of input data adequately. An underfitted model usually has less parameters specified than that would appear in correctly specified model. This makes it inflexible to in learning the information from the dataset.

Simple learners (underfit) tends to have less variance and in the predictions but more bias towards wrong outcomes. On the other hand, complex learners (overfit) tends to have more variance in the predictions but les bias towards wrong outcomes. This is known as The Bias-Variance Tradeoff.

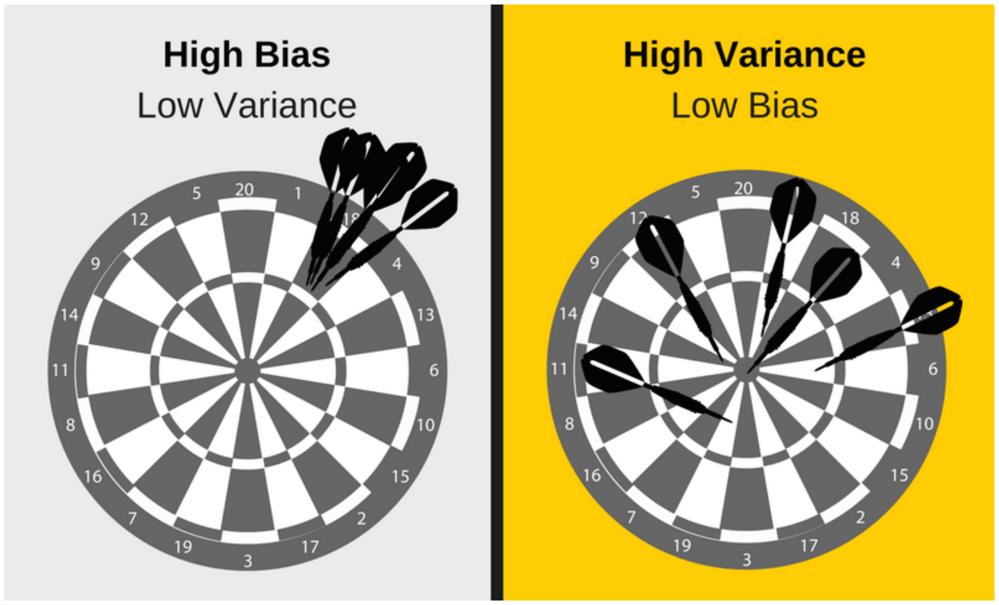


Figure 14 – Bias vs Variance

Overall, all the three systems can be summarized in figure 15. A decision boundary for simple, balanced and complex model can be observed below.

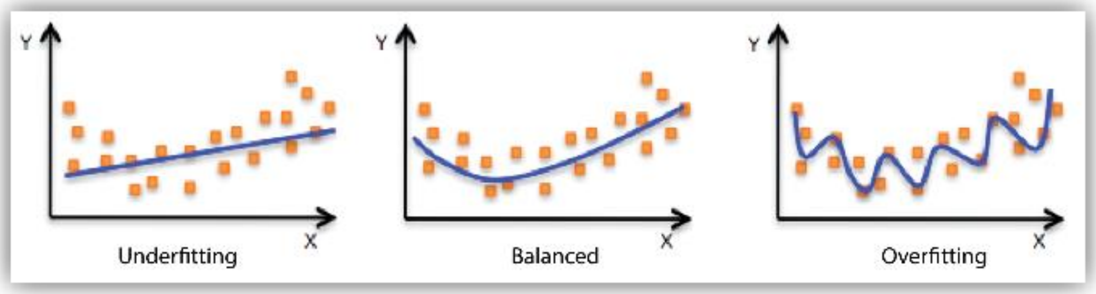


Figure 15 – Different predictive models

HOW TO DETECT OVERFITTING-

A key challenge with overfitting and with machine learning that we can't know the performance of model until we test the data. To address this issue, we can split the input data into separate training and testing subsets.

If our model does very good on training data and performs poorly on testing data, then we are likely to have overfit model.

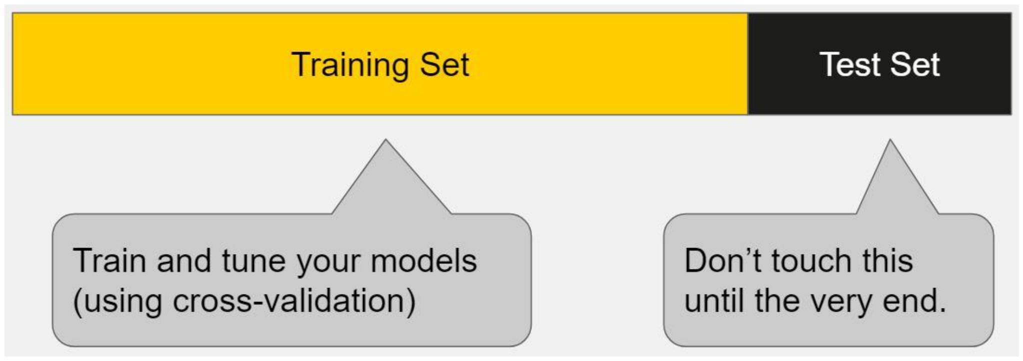

Training Set
Test Set
Train and tune your models (using cross-validation)
Don't touch this until the very end.

Figure 16 – Train-Test split

As shown in figure 16, we split the data into training and testing as validation set. Best way is to start with a very simple model to serve as benchmark. Then as you try more complex algorithms, we can compare with stated base benchmark to check whether its worth the additional complexity.

As a general rule, start by overfitting the model, then take measures against overfitting.

Cross Validation-

This is one of the most popular method to prevent overfitting. The basic idea is use your initial data to use many train test splits and use this data to train and validate your model.

One way to demonstrate is in your standard k-fold cross validation, we partition the data into k subsets. Iteratively, we train the model on k-1 subsets and test on remaining subset called holdout data.

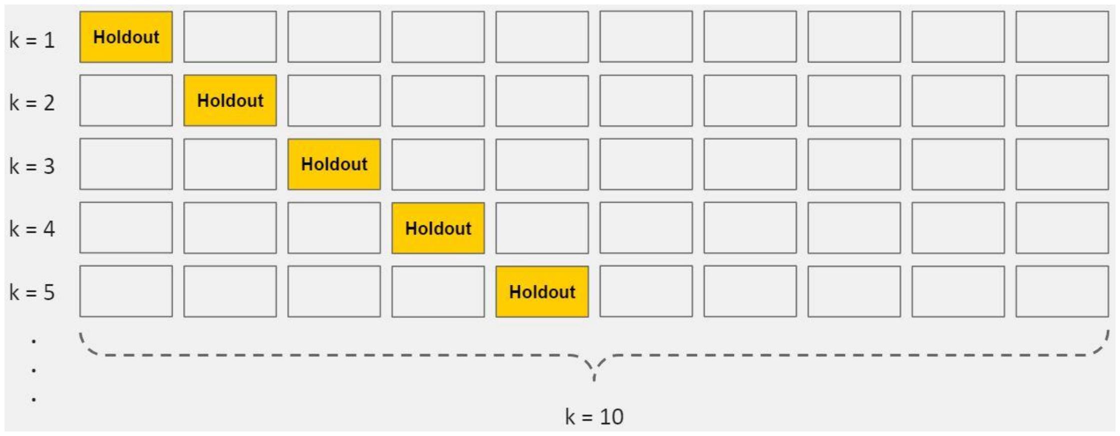


Figure 17 – K fold cross validation

This allows you to finely tune your parameters for the original data set which improves your maximum accuracy.

Train with more data & Data Augmentation-

In these methods, model might be benefitted more by training with more data sometimes. However, in most cases, this is not possible or having more data does not guarantee better performance. We have to ensure that input data is always clean and relevant to the application.

Hence, data augmentation is generally preferred where we randomly rotate, performs morphological operations, zooming and changing color filter etc. This ensures that the model get more data samples to train by augmenting or bootstrapping the data. We have to be careful by augmenting data for training only not for augmentation. Also, if we zoom so much that features of object are no longer visible, then this additional data does not help model getting more accuracy. We should not use too much data augmentation.

Early Stopping-

When we are training our model iteratively, we can measure the performance of every iteration. Until certain iterations, model has improvement in performance.

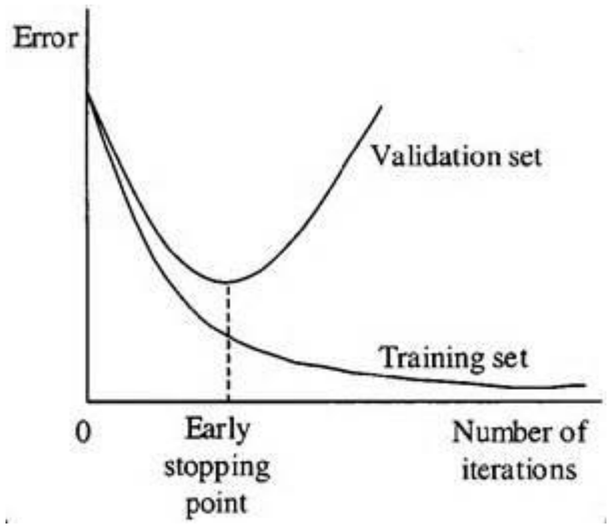


Figure 18 – Early stopping

As observed in figure 18, after certain iterations, model's ability to generalize can weaken as it overfitting problem arises. So, one way to tackle this is to stop where performance is good.

Regularization-

Regularization refers to broad variety of techniques which tries to make model simpler. Three most popular options are dropout, L1 regularization, L2 regularization.

Dropout is most famous techniques used is almost all deep learning models. It deletes random number of activations by making them zero shown in figure 19. This reduces number of connections to the next layers which might cause loss of information but reduces chances of overfitting. Dropout prevents network from becoming dependent om any neuron. Therefore, a good practice is to start with low dropout in first layer and then gradually increasing it hence less information will be lost.

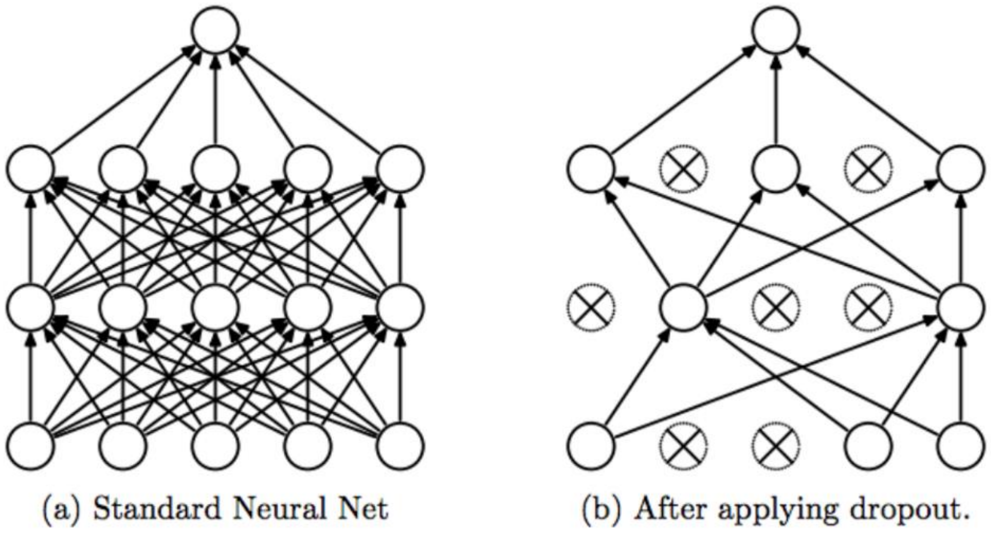


Figure 19 – Dropout

Another common type of regularization is *L2 regularization*. It can be implemented by increasing the error function with the squared magnitude of all weights in the neural network. In other words, for every weight w in the neural network, we add *1/2 λw^2* to the error function [2]. The basic idea to penalize peaky weight vectors and encouraging diffuse weight.

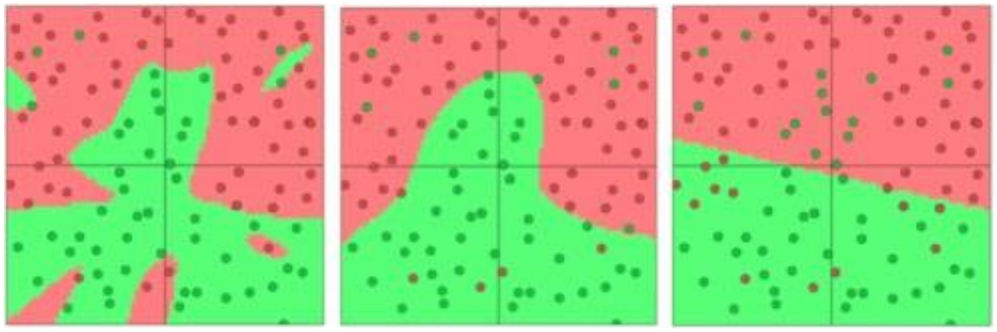


Figure 20– Separating the dots with L2 regularization with strengths 0.01,0.1,1

It encourages network to use all of the available weights rather than using some weights extensively. Using L2 regularization decays the weight linearly to zero. This phenomenon is known as weight decay.

Another type of regularization is L1 regularization. Here, we include the term *λ|w|* for every weight with the neural network. The *L1* regularization drives the weight vectors to become sparse amid optimization (i.e. very close to exactly zero). In other words, neurons with L1 regularization wind up utilizing just a little subset of their most essential sources of info and turn out to be very impervious to noise in the data sources [2]. We usually prefer L2 because it empirically performs better.

COMPARISON BETWEEN CNN AND TRADITIONAL METHODS IN COMPUTER VISION-

CNN are used in almost all the computer vision applications as it offers state of the art architecture and simplicity that helps us to solve tedious challenging problems in the field of object detection and tracking, image segmentation and classification etc.

Traditional computer vision methods for the stated applications above can be divided in to two parts mainly- feature extraction and classification. In first part, feature extraction, the process is tedious and is unique for every problem hence can-not be generalized. Hence, it requires lots of efforts and expertise in every field. Thus, second classifier is heavily dependent on feature extraction and hence classification accuracy depends on ability and knowledge of designer. This makes traditional methods less accurate and less robust.

In case of CNN, great amount of efforts has been taken and devoted to the interpretability of CNNs based on various tools and applications. Here, algorithm itself learns useful parameters and features from input.

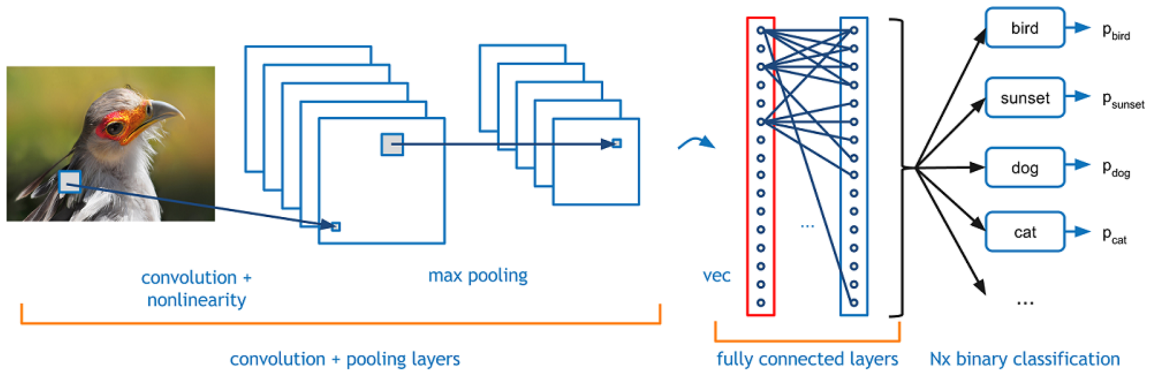


Figure 21– Image classification using CNN

Consider example of image classification of birds using CNN as depicted in figure 21. Same CNN architecture can be applied to the input data of set of layers. Sequential layers learn important features and filters on its own thus making system generalized.

Advantages of CNN-

Robustness to shifts and distortion in the image:

CNNs are location and displacement invariant since weight configuration is similar across the image. CNN has fully connected layer to get shift invariant features. While getting the features, CNN considers variation across the image to make it invariant thereby making it robust for all applications.

Fewer memory requirements:

For any size input image, CNN uses same coefficients across different locations which results into reducing memory requirement. For traditional methods, different coefficients will be used which will result in very high memory requirement.

Easier and better training:

For traditional method like ANN and NN, training time is proportional to number of connections and parameters to be trained. But in case of CNN, since number of parameters are drastically reduced, training takes less time compared to ANN.

Highest classification accuracy:

CNN always gets highest classification accuracy using less parameters compared to the all other methods.

THE LOSS FUNCTION-

Loss or cost function is the function that denoted difference between actual output and desired output. The smaller the cost function is, better is our model performance.

“A loss function or cost function is a capacity that maps an occasion or estimations of at least one factors onto a real number naturally speaking to some "cost" related to the occasion. An optimization issue looks to minimize a loss function. A target work is either a misfortune capacity or its negative (at times called a reward function, a benefit work, a utility function, a wellness work, and so on.), in which case it is to be maximized.” [3]

To determine the loss in our model, we will be using cross-entropy given by

$$Cross\ Entropy = -\sum_i y_i' \log(y_i)$$

Where y'_I is the ground truth label of ith training instance and y_I is the prediction result of your classifier for the iTH training instance.

MSE (Mean Squared Error) Loss:

$$MSE := \frac{1}{n}\sum_{t=1}^{n} e_t^2$$

MSE is the straight line between two points in Euclidian space.

L1 Loss:

It is most intuitive loss function.

$$S := \sum_{i=0}^{n} |y_i - h(x_i)|$$

Where S is the L1 loss, y_i is the ground truth and $h(x_i)$ is the inference output of your model.

BACK PROPAGATION-

"The backward propagation of errors or backpropagation, is a typical technique for preparing artificial neural systems and utilized as a part of conjunction with an enhancement strategy, for example, gradient descent. The calculation rehashes a two-stage cycle, propagation, and weight update. At the point when an information vector is exhibited to the system, it is propagated forward through the system, layer by layer, until the point when it reaches the output layer. The output of the system is then compared with the coveted yield, utilizing a loss function, and an error value is computed for every one of the neurons in the output layer. The error values are then propagated in reverse(back) direction, beginning from the output, until the point that every neuron has a related error value which generally speaks to its contribution to the original output." [4].

When we give the input image to the network, output is passed along the layers until the final error where using loss function, we calculate the error in estimation and then it is propagated backwards through the respective neurons until every neuron in every layer has been associated with an error value which represents its contribution for the actual output.

The fundamental thought behind back-propagation is to prepare multi-layered system with the end goal that it can learn the parameters to map to output without any other contribution from input. That's why we try to minimize the loss function is order to make best approximations.

TRAIN LENET-5 ON MNIST DATASET

Task: Train the CNN on MNIST dataset and discuss the settings and output.

ABSTRACT AND MOTIVATION

LeNet-5 architecture is very famous and well-organized architecture presented by Yann LeCun. It is multilayers neural network trained with back-propagation algorithm with gradient based learning techniques.

APPROACH AND PROCEDURES

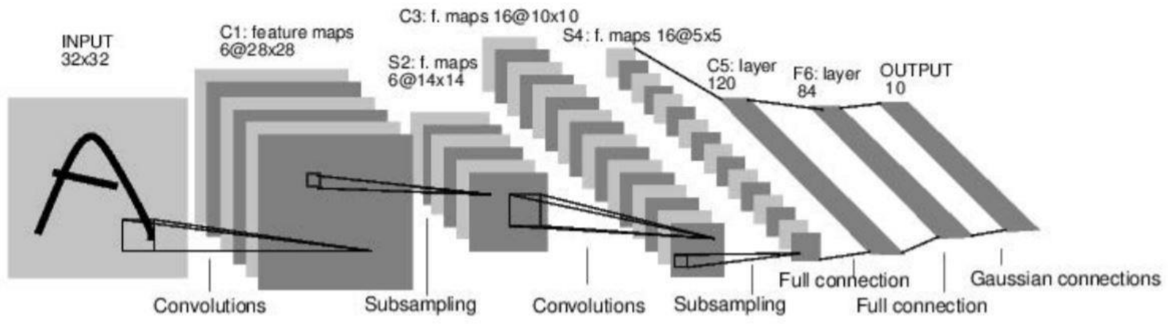


Figure 22– The LeNet 5 architecture

Figure shows LeNet 5 architecture. It consists of 3 convolutional layers denoted by C1, C2 and C3 and 2 average pooling subsampling layers and lastly 2 dense layers.

<u>Input Layer:</u>

Input layer contains 60000 images from MNIST of dimension 28*28 and 10000 test images. We do pad the images to make 32*32 to use LeNet 5 architecture.

<u>Convolutional Layer C1:</u>

This layer contains 6 filters of size 5*5 which are being convolved around input image to obtain 6 feature maps of 28*28. For the high dimensional input data, number of connections would be very high hence smart option would be choosing local patch and connect that patch with output neuron. This lessens the receptive field of every neuron which is adjusted for in the following layers of this engineering. In the LeNet-5, the receptive field was chosen to be 5x5.

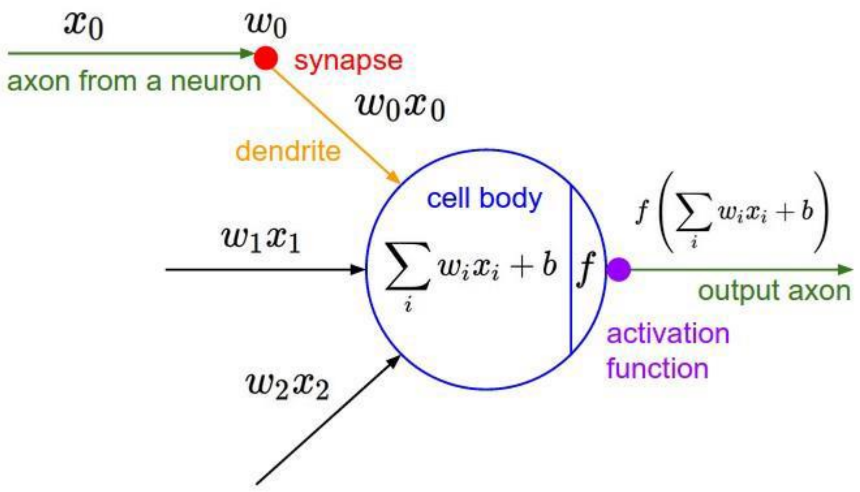


Figure 23– Convolution logic

The basic convolutional logic is explained in figure 23. The given architecture has 6 filters of dimensions 5*5 of stride 1 with zero padding. Mode; summary in stated in the last part.

Subsampling Average Pooling Layer S2:

This layer is average pooling. In order to improve the receptive field, average pooling is used which reduces the number of weights. It has stride parameter of 2. This layer does not have any weights.

Sigmoid Activation Function:

This non-linear function is used alongside pooling layer which is applied element by element. It converts input response to 0-1 but usually has slower convergence. The reason for utilizing a non-linear activation layer is to present this non-linearity in the system. Certifiable information and procedures can be non-linear in nature. This decided which neuron should get fired and which should not.

Convolutional Layer C3:

This layer contains 16 filters of dimension 5*5. It takes input of 6*14*14 and convolves 16 filters which results into 16*10*10 output.

Subsampling Average Pooling Layer S4:

This layer takes input from C3 layer. It averages over 2*2 region with stride parameter of 2. This reduces dimension from 10*10 to 5*5 resulting 16*5*5.

Convolutional Layer C5:

This layer consists of 120 neurons i.e. weights. 120 filters of size 5*5 have been used. 5*5 convolved 5*5 input image results into single pixels hence it contains only 120 weights.

Fully Connected Layer F6:

This is fully connected and it has 84 weights. This layer has the maximum receptive field and every neuron has the connection with input image.

Output Layer:

This is final layer consists 10 neurons representing expected output labels. We have 10 neurons corresponding to 10 numbers with activation function softmax.

Entire model summary at every step is mentioned below.

In total, network has 61706 parameters to train.

Model Parameters:

<ins>Batch Size:</ins>

We use mini-batch gradient descent. For every batch size, we update the weights of the network. This parameter is usually determined using brute force. Using batch training, we reduce the variance of filter weight change. It contributes towards learning rate. Larger batch size will result in slower convergence.

<ins>Learning Rate:</ins>

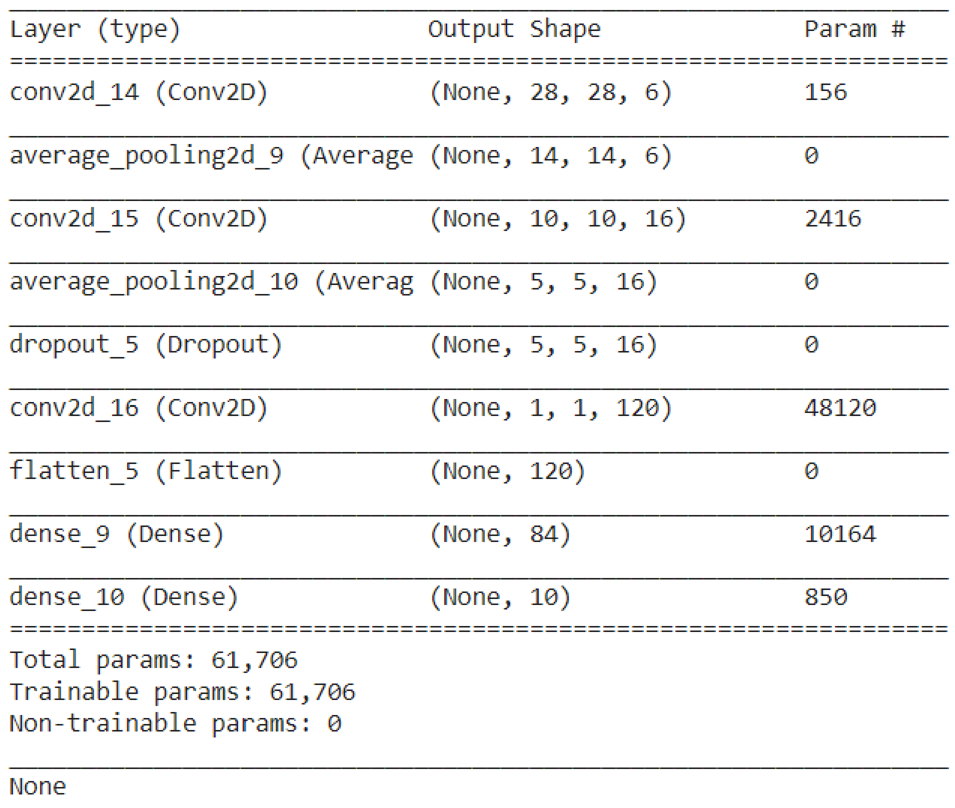
This parameter determines convergence time of the model. Smaller learning may converge very slowly but larger convergence may not reach global minima in the first place. These can be taken care by decaying learning rate.

<ins>Momentum:</ins>

As stated above in mini batch SGD approach, it has oscillations problems which can be harmful for convergence. Momentum is used to prevent or limit those parameters.

<ins>Drop Out:</ins>

To reduce or limit overfitting of data, dropout is usually used after sub sampling layer. It neglects certain neuron to get fired thereby reducing number of connections.

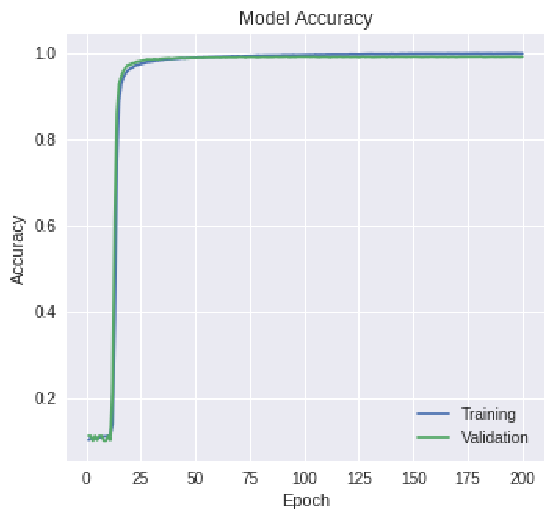


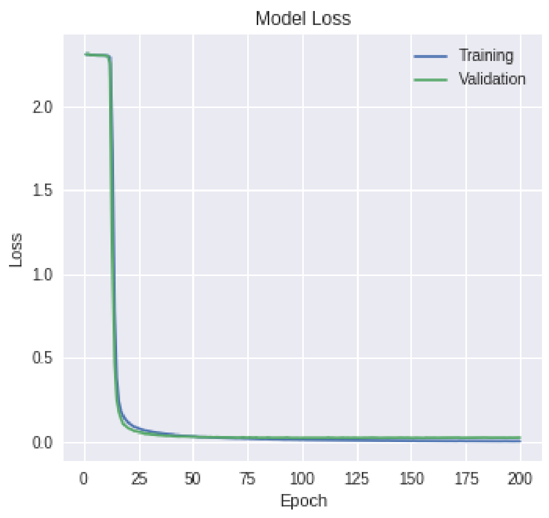
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_14 (Conv2D)           (None, 28, 28, 6)         156       
_________________________________________________________________
average_pooling2d_9 (Average (None, 14, 14, 6)         0         
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 10, 10, 16)        2416      
_________________________________________________________________
average_pooling2d_10 (Averag (None, 5, 5, 16)          0         
_________________________________________________________________
dropout_5 (Dropout)          (None, 5, 5, 16)          0         
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 1, 1, 120)         48120     
_________________________________________________________________
flatten_5 (Flatten)          (None, 120)               0         
_________________________________________________________________
dense_9 (Dense)              (None, 84)                10164     
_________________________________________________________________
dense_10 (Dense)             (None, 10)                850       
=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 24– Model Summary

EXPERIMENTAL RESULTS

Kernel Initializer: Random Normal   Learning Rate: 0.1   Epochs: 200
Momentum: 0.5   Batch Size: 128

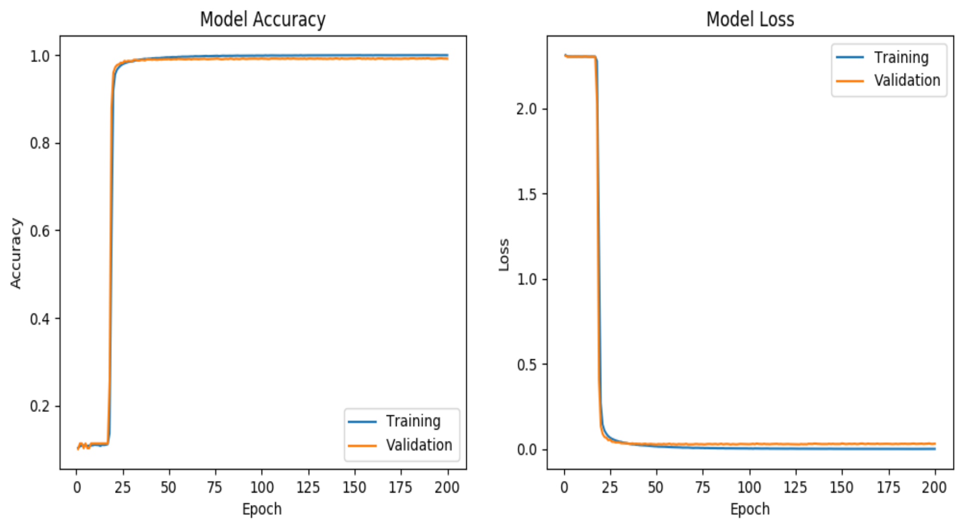




Training Accuracy: 99.93   Train Loss: 0.34
Test Accuracy: 99.21   Test Loss: 2.59

Kernel Initializer: Random Normal   Learning Rate: 0.3   Epochs: 200

Momentum: 0.5   Batch Size: 128



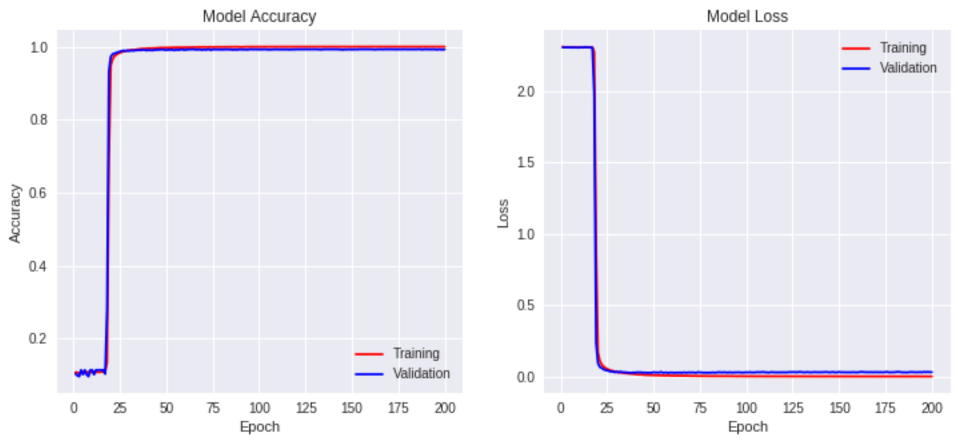
Training Accuracy: 99.97   Train Loss: 0.14

Test Accuracy: 99.20   Test Loss: 3.17

Figure 25– Accuracy and Loss curves

Kernel Initializer: Random Normal   Learning Rate: 0.5   Epochs: 200

Momentum: 0.5   Batch Size: 128

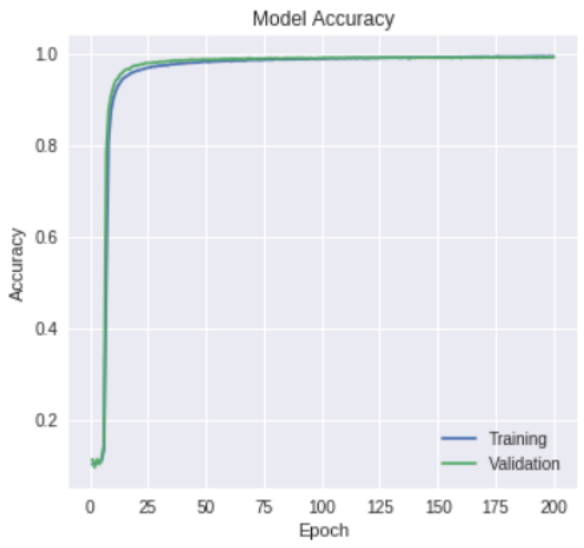


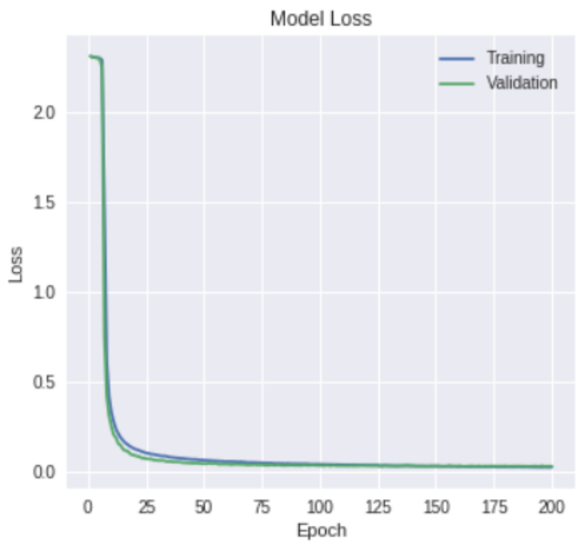
Training Accuracy: 99.98   Train Loss: 0.11

Test Accuracy: 99.26   Test Loss: 3.12

Kernel Initializer: Random Normal Learning Rate: 0.1 Epochs: 200

Momentum: 0.25 Batch Size: 64



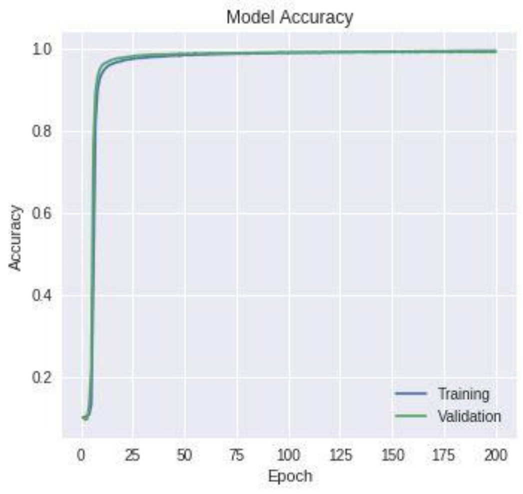


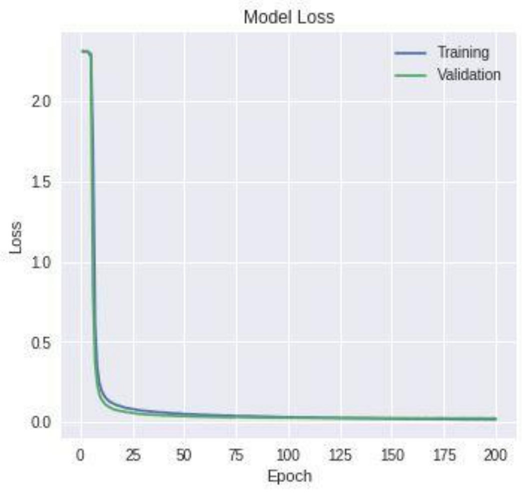
Training Accuracy: 99.38 Train Loss: 1.67

Test Accuracy: 99.10 Test Loss: 2.62

Kernel Initializer: Default Learning Rate: 0.1 Epochs: 200

Momentum: 0.5 Batch Size: 128



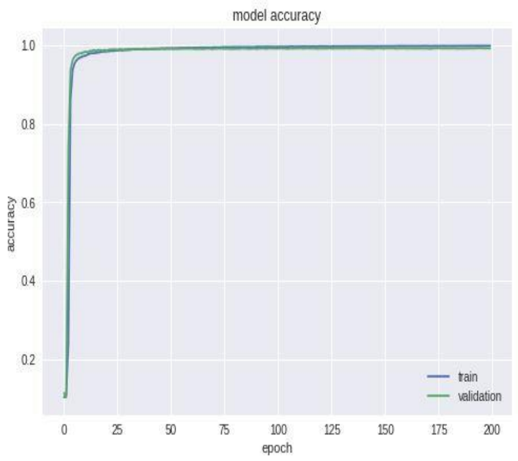


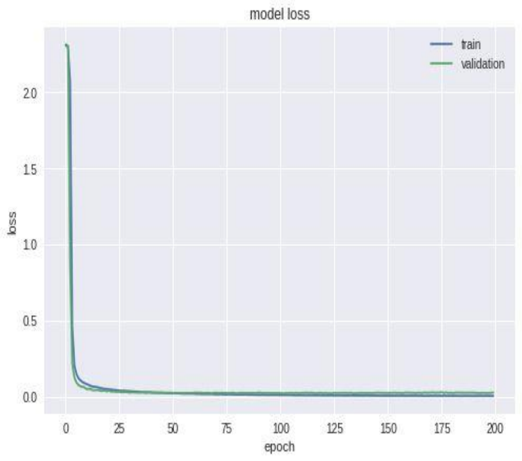
Training Accuracy: 99.48 Train Loss: 1.62

Test Accuracy: 99.14 Test Loss: 2.22

Kernel Initializer: Default Learning Rate: 0.3 Epochs: 200

Momentum: 0.5 Batch Size: 128



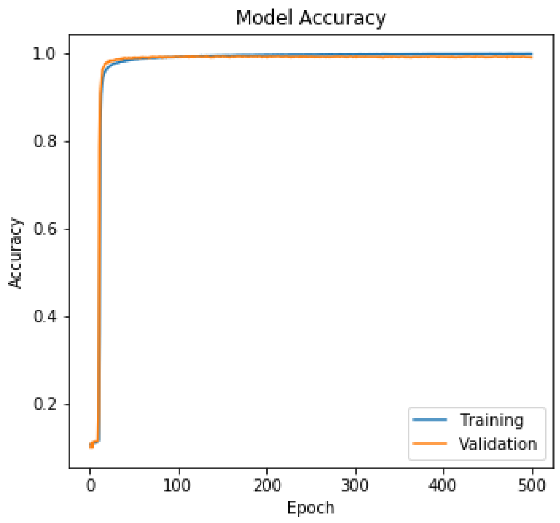


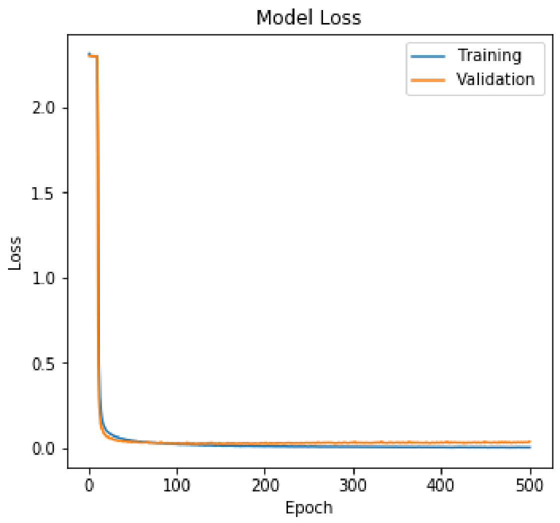
Training Accuracy: 99.84 Train Loss: 0.55

Test Accuracy: 99.19 Test Loss: 2.72

Kernel Initializer: Default Learning Rate: 0.5 Epochs: 500

Momentum: 0.5 Batch Size: 128



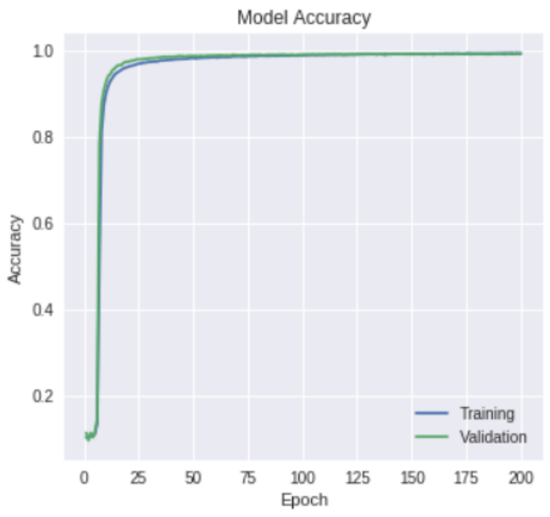


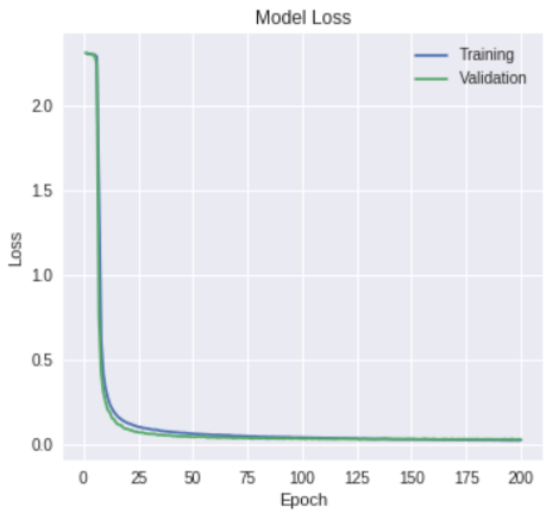
Training Accuracy: 99.86 Train Loss: 0.52

Test Accuracy: 99.29 Test Loss: 2.2

Kernel Initializer: Default Learning Rate: 0.1 Epochs: 200

Momentum: 0.25 Batch Size: 64



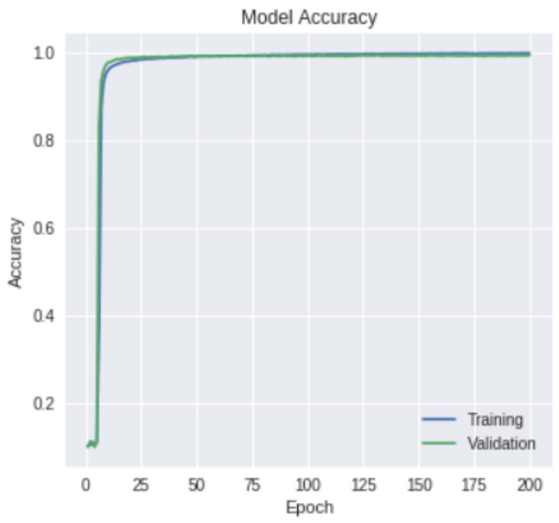


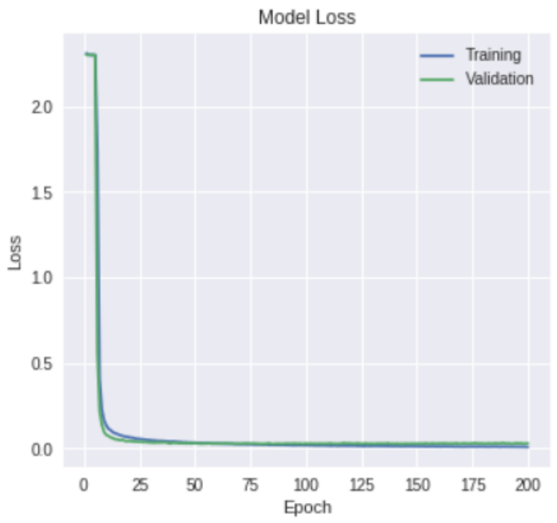
Training Accuracy: 99.30 Train Loss: 2.02

Test Accuracy: 99.09 Test Loss: 2.62

Kernel Initializer: Default Learning Rate: 0.1 Epochs: 200

Momentum: 0.5 Batch Size: 32



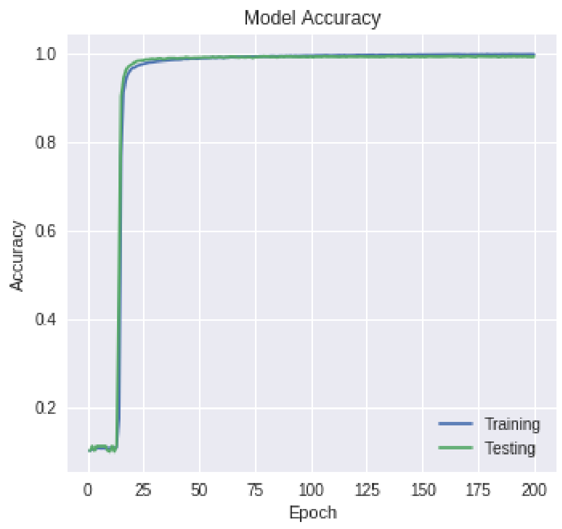


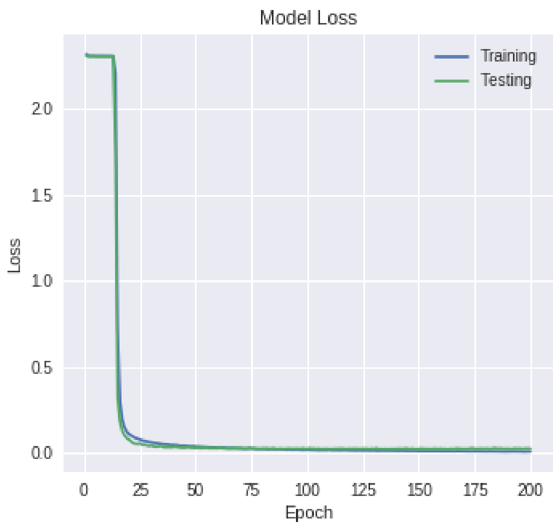
Training Accuracy: 99.70 Train Loss: 0.82

Test Accuracy: 99.27 Test Loss: 2.92

Kernel Initializer: Random Uniform   Learning Rate: 0.1   Epochs: 200

Momentum: 0.25   Batch Size: 64



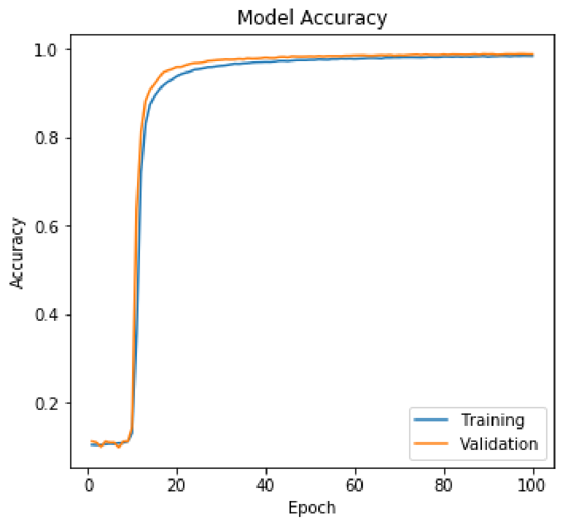


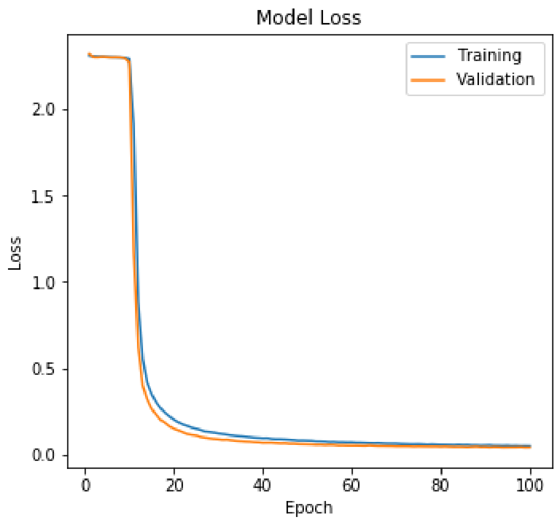
Training Accuracy: 99.70   Train Loss: 0.92

Test Accuracy: 99.17   Test Loss: 2.52

Kernel Initializer: Random Uniform   Learning Rate: 0.1   Epochs: 100

Momentum: 0.5   Batch Size: 32



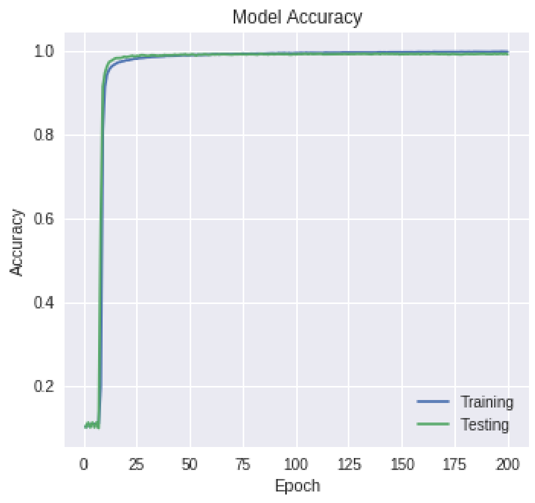


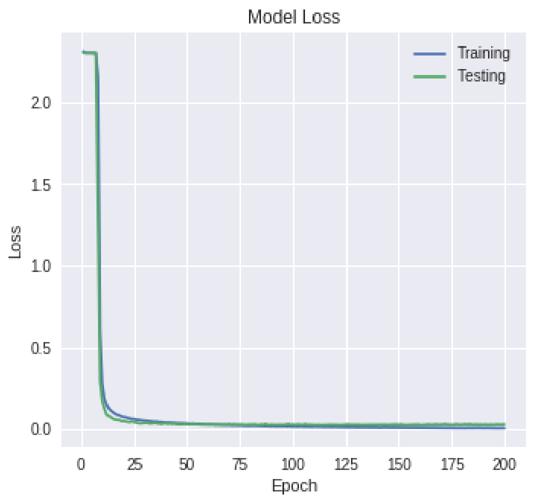
Training Accuracy: 99.63   Train Loss: 0.97

Test Accuracy: 99.13   Test Loss: 2.72

Kernel Initializer: Random Normal Learning Rate: 0.1 Epochs: 100

Momentum: 0.5 Batch Size: 32





Training Accuracy: 99.73 **Train Loss: 0.82**

Test Accuracy: 99.36 **Test Loss: 2.70**

```
[[ 974    0    1    0    0    0    3    1    1    0]
 [    0 1131    1    0    0    1    1    0    1    0]
 [    0    1 1028    0    1    0    0    2    0    0]
 [    0    0    2  999    0    6    0    1    2    0]
 [    0    0    0    0  975    0    5    1    0    1]
 [    1    0    0    2    0  887    1    0    1    0]
 [    2    2    0    0    1    2  950    0    1    0]
 [    0    2    5    0    0    0    0 1017    1    3]
 [    2    0    2    1    1    2    0    0  964    2]
 [    0    1    0    0    8    3    1    1    1  994]]
```

Figure 25– Best accuracy plot and Confusion Matrix

After best setting, I tried training and testing the model for negative (inverted) images.

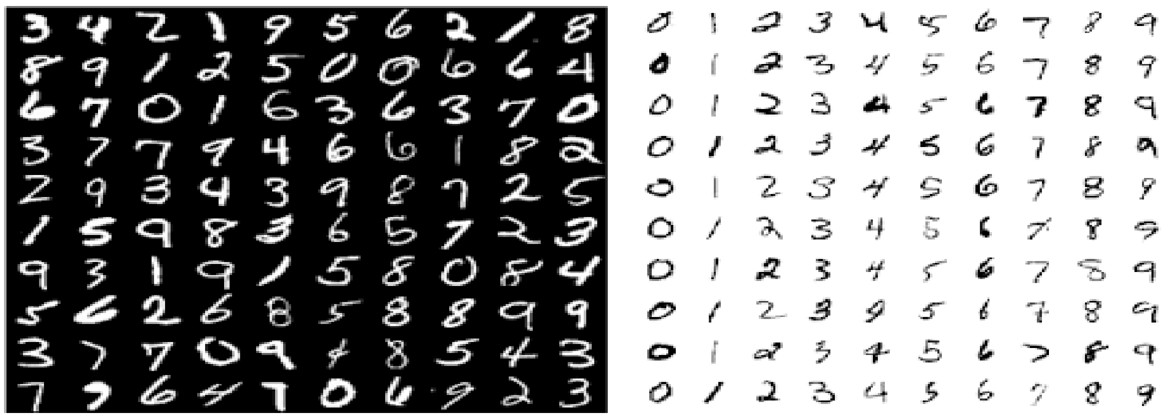
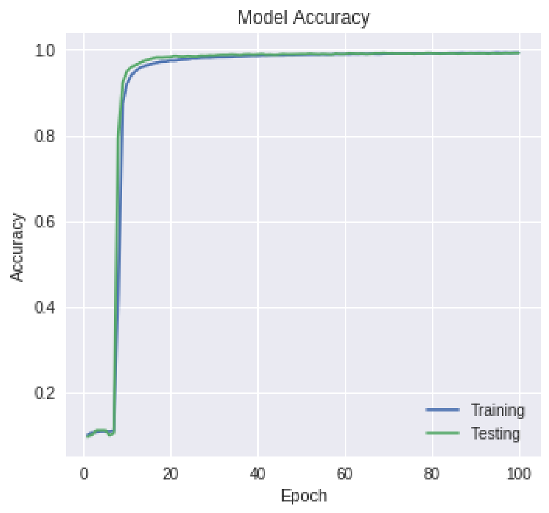
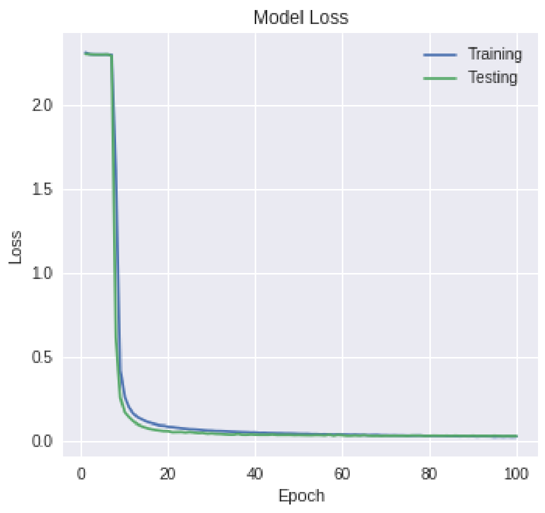


Figure 26– Original and Negative images

Kernel Initializer: Random Normal Learning Rate: 0.3 Epochs: 100

Momentum: 0.5 Batch Size: 128





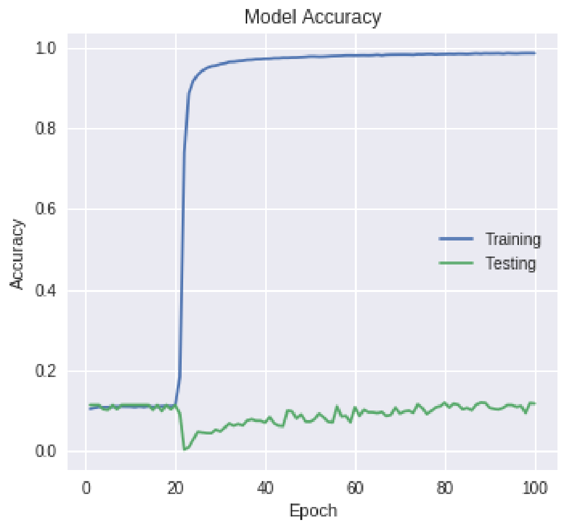
Training Accuracy: 99.23 Train Loss: 2.32

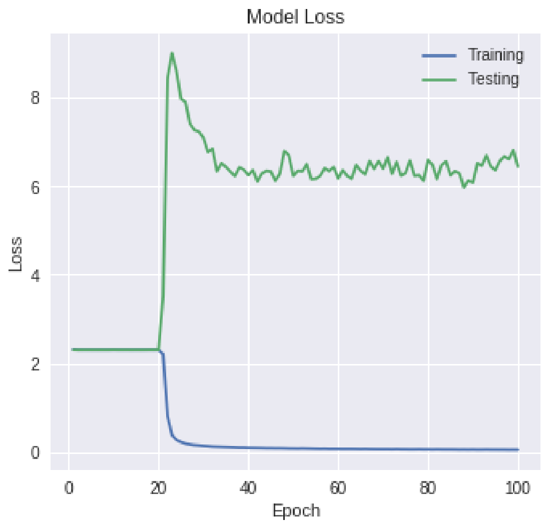
Test Accuracy: 99.17 Test Loss: 2.67

After this effort, I trained the network on negative image but tested the network on original images.

Kernel Initializer: Random Normal Learning Rate: 0.3 Epochs: 100

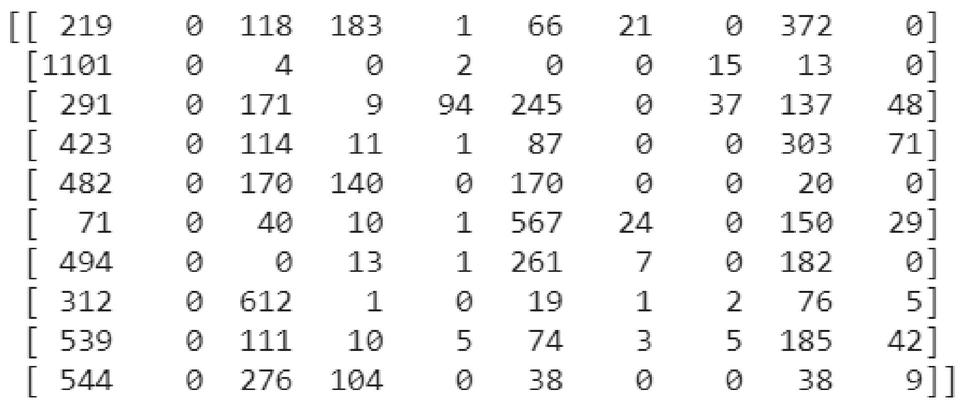
Momentum: 0.5 Batch Size: 128





Training Accuracy: 98.57 Train Loss: 4.29

Test Accuracy: 11.17 Test Loss: 6.47



```
[[ 219    0  118  183    1   66   21    0  372    0]
 [1101    0    4    0    2    0    0   15   13    0]
 [ 291    0  171    9   94  245    0   37  137   48]
 [ 423    0  114   11    1   87    0    0  303   71]
 [ 482    0  170  140    0  170    0    0   20    0]
 [  71    0   40   10    1  567   24    0  150   29]
 [ 494    0    0   13    1  261    7    0  182    0]
 [ 312    0  612    1    0   19    1    2   76    5]
 [ 539    0  111   10    5   74    3    5  185   42]
 [ 544    0  276  104    0   38    0    0   38    9]]
```

Figure 27– Plots and confusion matrix for negative training images

DISCUSSION

I simulated LeNet 5 model using various parameters. There are various parameters like batch size, epochs, kernel initializer, learning rate, momentum and optimizers to change and I plotted training and testing accuracy as well as loss.

Preprocessing:

MNIST dataset contains 28*28 image with global mean subtracted from every image as well as data was already zero centered. This prevents high-frequency components to dominate over image. As data was already pre-processed, I have just normalized the data to 0-1.

Plots and Data Interpretation:

From the plots shown above, initially CNN has very low accuracy and very high loss. After every iteration, it uses backward propagation to reduce the error and hence we can observe accuracy increases at some iterations and simultaneously loss decreases.

All the plots were plotted for training and testing from using tensor board. The path and link to view the outputs using tensor board has been given in the code.

Using basic LeNet 5 architecture, I maintained the architecture by tweaking few parameters to check the effect on accuracy and loss.

Results:

From above plots, sometimes we observe little increase in loss after certain iterations and this is known as overfitting. To tackle this problem. I have introduced dropout layer to reduce risk of overfitting. Secondly, if learning rate is very high, gradient traverse from one point to another point constantly and hence it might not be able to global optimum which is undesirable. Hence finding optimum learning rate is very optimum.

Having very low learning rate results into very low convergence rate and hence takes very long time to converge. The weights are largely away from optimum value but as we reduce the error, learning rate also reduces owing to exponentially decaying function. This results into localization of global minima.

Best Parameters:

I have got the best training accuracy for last setting plotted in last plot. This setting is optimum versions containing perfect tradeoff between accuracy and convergence and loss parameters. We have dropout to avoid overfitting.

Negative setting:

When we train and test the system using negative images, results are very good and no change can be seen. But when we train the model on negative images and test the model on original images, then performance is very bad as stated in approach, CNN does not have flexibility and is not so robust

For negative images, accuracy is relatively low which indicates that CNN does not perform good for black background and white characters. This happens due to CNN's dependency on pixel values. It doesn't usually change the dependency that easily hence we change the training parameters to adjust for the lack of elasticity in model.

IMPROVEMENT OVER LENET 5 MNIST

APPROACH AND PROCEDURES

The LeNet 5 architecture mentioned above has several bad performing sections where improvement can be made.

Filter Numbers: As we increase number of filters, we get more features that improves feature efficiency and usually results into better classification. I increased number of filters from 6 to 32,64 etc.

Filter Size: I tried 3*3, 5*5 and 7*7 filters. Depending upon input images, for some settings, I got better output using 3*3 filters.

Activation Function: Because of function curve of ReLu activation function, it improves the receptiveness of network. Hence, instead of sigmoid, I used ReLu activation function everywhere because of its unique to decide whether to fire the neuron or not.

Number of Layers: I increased number of layers by adding convolutional and pooling layers and cascaded those layers to get better output.

Batch Size: Batch size is the number of training images that are being trained simultaneously. I varied batch size but this setting does not help to improve the accuracy.

Kernel Initializer: I have tried Random Normal, Random Uniform and default Glorat Normal kernels for different parameters and comparison has been shown in figure.

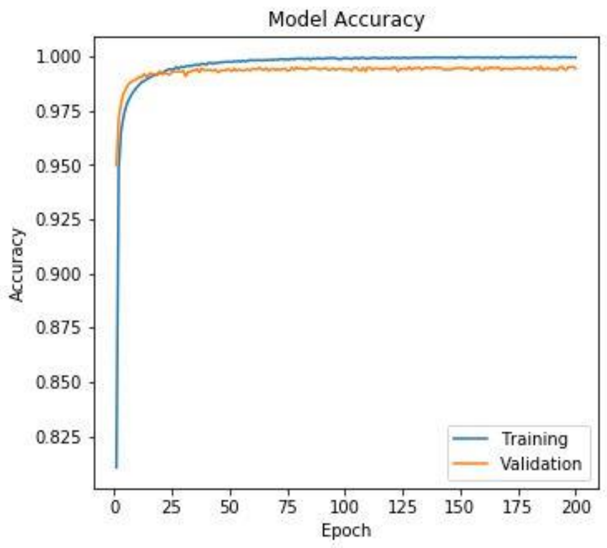
Optimizer: this plays most important role for improving the performance of model. I have used Adam, Adam delta and SGD. SGD has very slow convergence which can be tackled using Adam optimizer.

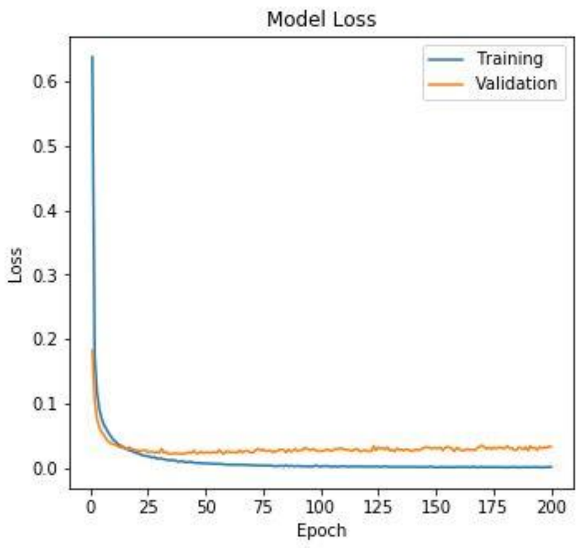
Drop Out: This improves the performance by making sure overfitting is being avoided.

EXPERIMENTAL RESULTS

Setting 1:

Conv1 Filters: 50, Conv2 Filters: 100, Dense Layers: 120, 84 Activation Function: ReLu, Optimizer: Adam (lr:0.0001)

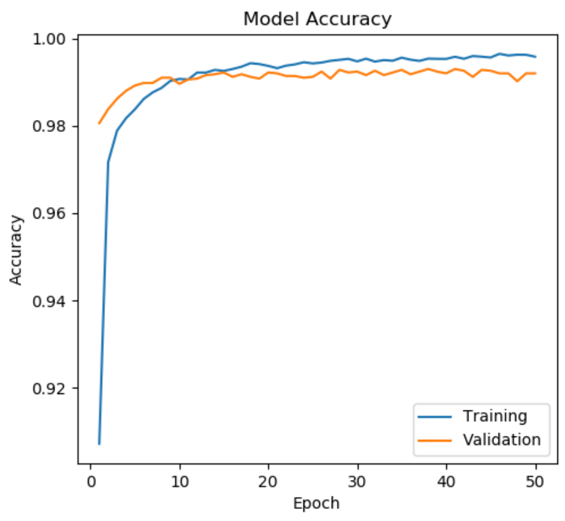


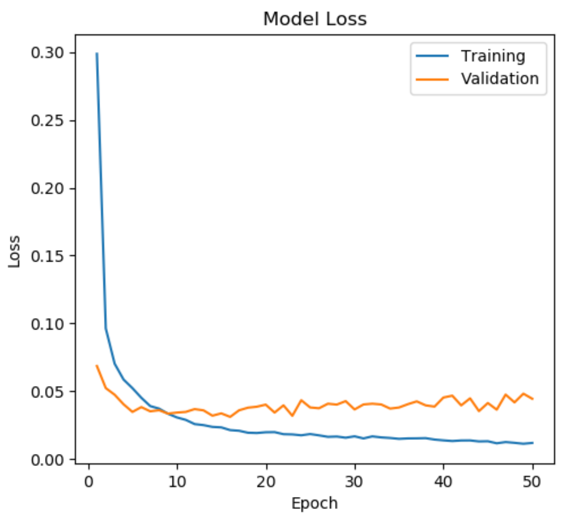


Training accuracy: 99.98%, Test Accuracy: 99.42%

Setting 2:

Conv1 Filters: 32, Conv2 Filters: 64, Dense Layers: 120 Activation Function: ReLu, Optimizer: Adam (lr:0.0001)

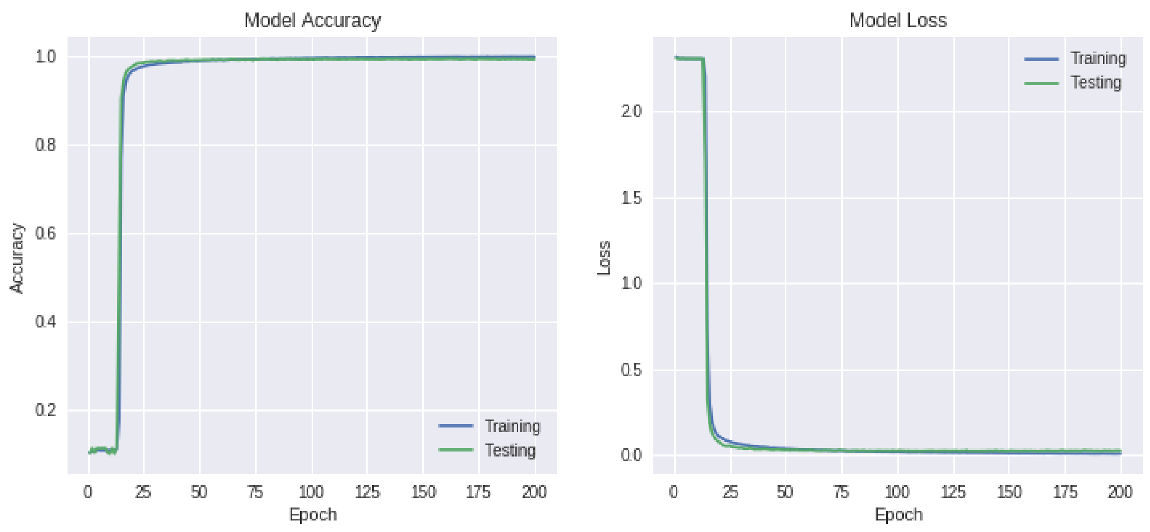




Training accuracy: 99.94%, Test Accuracy: 99.47%

Setting 3:

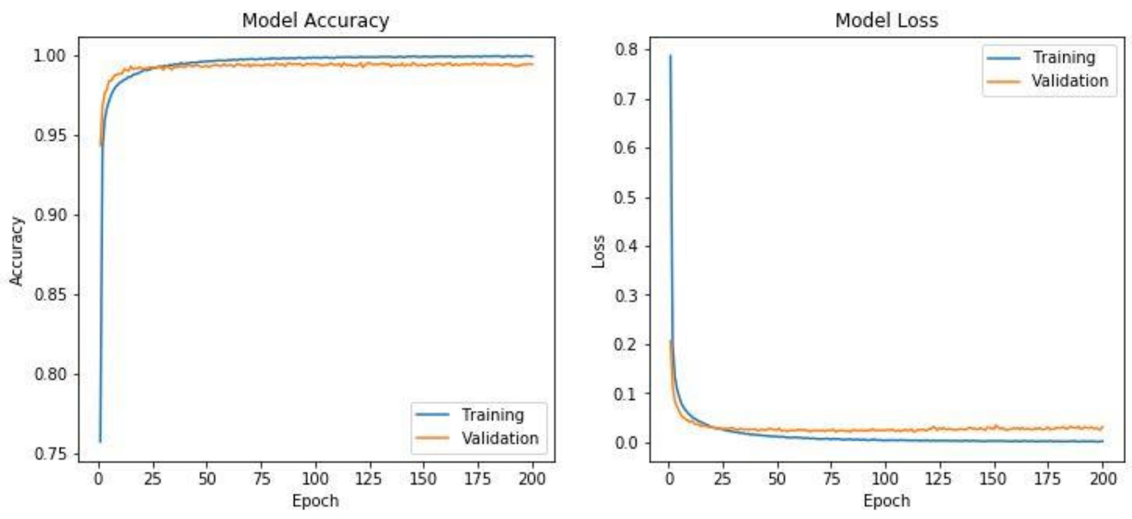
Conv1 Filters: 32, Conv2 Filters: 64, Dense Layers: 128,84,32 Dropout: 0.5Activation Function: ReLu, Optimizer: Adam



Training accuracy: 99.95%, Test Accuracy: 99.39%

Best Setting:

Conv1 Filters: 32, Conv2 Filters: 64, Dense Layers: 120, 84 Activation Function: ReLu, Optimizer: Adam (lr:0.0001)



Training accuracy: 99.97%, Test Accuracy: 99.54%

DISCUSSION

I have stated different parameters that I tinkered with to check their effect on performance.

Batch Size:

I varied batch sizes from 32 to 128 for different parameters and found out that in does not affect to an extent to the accuracy and convergence.

Epochs:

Number of epochs depends upon learning rate and batch size. For low learning rate, number of epochs are usually larger and vice-versa.

Number of Layers:

Number of layers usually gives better approximations then lesser number of layers but it is not always true. I tried from 2 to 4 sets of layers and compared for different setting. I got the best output for 2 convolution and pooling layers. I also increased number of fully connected layers to test effects of more layers. I did not find much difference by changing number of layers.

Number of Filters:

More number of filter gives better feature approximations. I varied number of filters from 6 to 64. I found out that filters 32 and 64 in respective layers give better output. This plays very important role in accuracy.

Activation Function:

I used ReLu activation function instead of sigmoid function owing to various advantages like better convergence, does not saturate etc. This factor has great impact on accuracy and convergence.

Optimizer:

This is most important parameter that affects accuracy. SGD optimizer is very slow in convergence and does not guarantee global minima. I have used Adam optimizer which has high convergence with annealing method for learning rate.

PROBLEM 2 – SAAK TRANSFORM

COMPARISON BETWEEN SAAK TRANSFORM AND CNN

Task: Understand the SAAK transform and compare SAAK transform and CNN

ABSTRACT AND MOTIVATION

SAAK (Subspace approximation with augmented kernels) transform is an efficient and robust method to recognize and classify handwritten digits recognition problem. The basic idea behind the SAAK transform is it uses multiple stages to extract group of SAAK coefficients for input image and use these coefficients to classify the input using classifier like support vector machine, random forest etc.

APPROACH AND MOTIVATION

SAAK transform consists of 2 steps- subspace approximations and kernel augmentation. To achieve optimal subspace approximation, it analyzes and selects best orthonormal eigenvectors from set of available covarinace matrix as transform kernels. Here, the problem faced is usually the training data space is very large hence multiple transforms are cascaded. While cascading multiple stages of trasnform, sign confusion arises hence rectified linear unit (ReLu) activation function to limit the confusion. But due to non-linearity about activation function, there is some information loss and that can be tackled by kernel augmentation. Hence, after every SAAK stage, we get both original and augmented kernel.

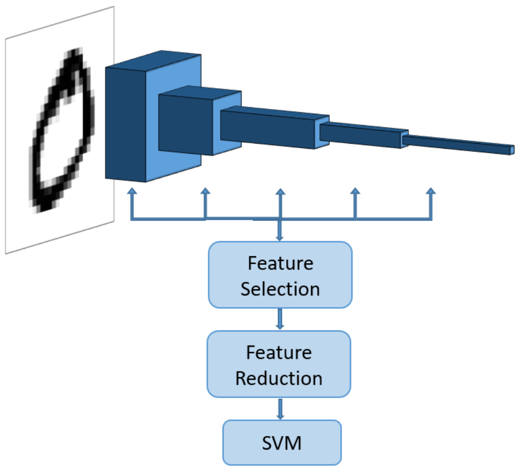


Figure 28– SAAK transform approach

Kernel augmentation can be performed using augmenting with negative kernels. The integration of kernel augmentation and ReLu is similar to S/P format conversion of SAAK coefficients. So this way, we are able to cascade different stages to transform images of exceptionally high sizes and dimensionality.

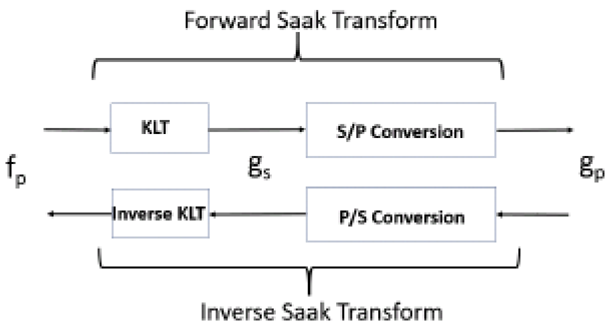


Figure 29– SAAK transform block diagram

Multi-stage SAAK transform offers full spatial and spectral domain representation. The extracted SAAK coefficients are then used as features and fed into SVM or any other classifier for classification procedure. Figure 24 shows detail working of approach. After every stage, we select important features, we use PCA to reduce the dimensions and finally affer all stages, we use SVM classifier to to classify the data. Usually, due to input data size, we adopt loss SAAK transform to reduce the number of coefficients by performing Principal Component Analysis(PCA). This makes SAAK transform more robust to small pertubations by focusing on main components.

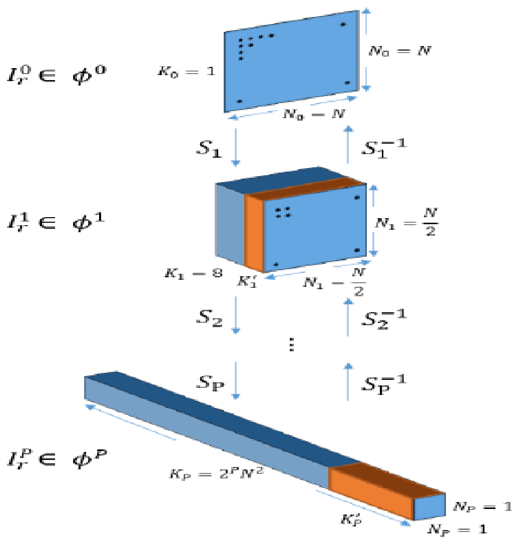


Figure 30– SAAK transform multiple stage representation

SAAK transform and CNN has some advantages and disadvantages from the proposed structure which has been stated below.

Model Design:

CNN architecture features end to end sequential model with connected layers and optimized cost function. All the mentioned layers above in CNN architecture are connected and after classification, using back-propagation, weights have been optimized. On the other hand, just like another traditional method, SAAK transform has 2 separate blocks of feature extraction and classification. Hence, the classification accuracy based on first block.

Design Flexibility:

If number of classes in fixed, then end-to-end system like CNN has better optimized cost function and filter weights. However, even from small perturbations in input data, we have to train CNN once again. But for SAAK method, it has very good robustness for small changes in data because KLT transform is invariant to small shifts hence you do not need to find features again. The small change will not affect leading last-stage Saak coefficients

If we change number of classes in final layer, we need to train the system once again as number of neurons change in the last layer. However, for SAAK transform, it is against intuition filter weights for feature extraction part from SAAK does not change as they both are not dependent upon other parts.

CNN needs very large training data set and takes long time and is computationally inefficient. On the other hand, SAAK takes less computations to train and is computationally efficient. This statement has been explained in detail in results section.

Generative model vs Inverse Transform:

Since the kernel of SAAK transform are orthonormal basis, its corresponding inverse transform can be easily computed. Hence, cascading of forward and inverse transform shown in figure 30. It makes an identity operator. But on the other side for CNN, Generative Adversarial network(GAN) has to be trained first and then using encoder and decoder, we can generate the stage-wise images.

Theoretical Foundation:

CNN has sequential model and cascaded filters along every layer. The problem with CNN is visualization of responses from every filter is lacking and since we don’t know what are the stage wise responses, its very difficult to keep the track of the output. As the complexity of CNN increases, behavior of model and filter responses become intractable. On the contrary side, SAAK transform has algebraic base and supported by proper statistics. At every stage, SAAK transforms are well explained.

Filter Weights and Feature Selection:

CNN learns best filter weights optimizes them by processing data and its associated labels. It optimizes by reducing the error by back-propagation. In SAAK transform, it selects its augmented kernel using KLT at every stage. It doesn't need any data and its associated label. Hence back-propagation is also not required. Kernel selection is based upon second-order statistics of data. Data labels are only needed in classification part, not for the feature extraction.

Efficiency:

For any given dataset, SAAK transform has better classification accuracy for minimal data for training. The comparison table has been shown in result section. CNN requires huge amount of data to train in order to produce very high accuracy output. While SAAK produces very good output even when training data is very low in amount.

APPLICATION OF SAAK TRANSFORM TO MNIST DATASET

Task: Train and test the MNIST dataset using SAAK transform method

EXPERIMENTAL RESULTS:

	Support Vector Machines		Random Forest	
PCA Components	Training Acc	Testing Acc	Training Acc	Testing Acc
No PCA used	97.34	94.30	Can-not run	On CPU
32	**99.22**	**98.32**	**99.92**	**93.40**
64	98.87	98.20	99.57	92.37
128	98.02	97.60	99.31	90.70

PCA = 32	Support Vector Machines		Random Forest	
Train Data Size	Training Acc	Testing Acc	Training Acc	Testing Acc
10000	99.07	96.79	99.87	88.96
20000	99.10	97.48	99.90	90.77
30000	99.20	97.86	99.91	92.23
40000	99.23	98.13	99.93	92.67
50000	99.21	98.20	99.91	92.81
60000	**99.22**	**98.32**	**99.92**	**93.40**

DISCUSSION

I have trained the 60000-input data and tested the system on 10000 test data. SAAK coefficients obtained were of dimensions 2048 due to filter importance. For every stage, I used **4,12,10,20,16** important components which gave me feature vector consisting of **2048** features. Table shows training and testing accuracy for SVM and Random Forest classifier using PCA for different components and without using PCA as well. From Table 1, we can deduce 3 points. I have used **F-test score** measure or chi_sqaure test to find best **1000** features out of features at every stage then used only those important features while moving forward with next stage.

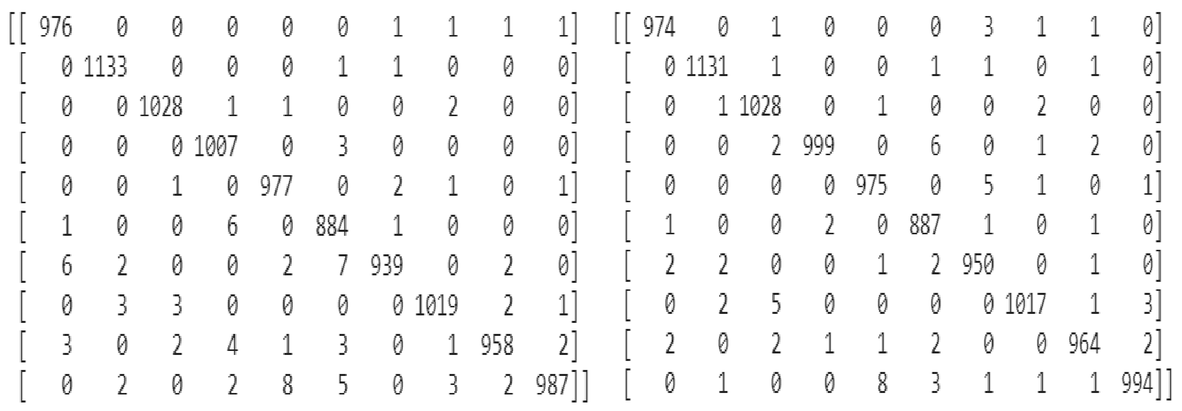
I have feature vector of dimension 2048. If I don't use PCA for dimensionality reduction, I am getting very good training and testing accuracy for SVM. Random forest classifier is computationally expensive and system crashes when tried to run the code using CPU for windows. As I implement PCA for stated component, my training and testing accuracy slowly starts decreasing as I increase the number of components in PCA.
Comparison between SVM and RF can be observed from the table. SVM performs very well where Random Forest algorithm has overfitting issue as it has high training accuracy but relatively low testing accuracy. We can tackle overfitting by various ways stated in first question. Random Forest works well with a mixture of numerical and categorical features. When features are on the various scales, it is also fine. Roughly speaking, with Random Forest you can use data as they are. SVM maximizes the "margin" and thus relies on the concept of "distance" between different points. As a consequence, one-hot encoding for categorical features is a must-do. Further, min-max or other scaling is highly recommended at preprocessing step.
Best classification score for both the table
In second table, for 32 components using PCA, I have calculated training and testing accuracy for different size of data used. Table shows that SAAK performs exceptionally good even with less amount of data is available to train whereas CNN requires very high amount of data to be trained in order to give very good classification accuracy. I am getting highest accuracy for all the 60000 images for training data set but even compared with very small train and test data set, it still performs very good.
This is one of the main advantages of SAAK transform over CNN.
Comparing performance of CNN with SAAK, SAAK offers robustness, scalability and requires less amount of training data to get similar classification accuracy.
When introduced any sort of noise in an image or when noisy images are being used to train and test the model, SAAK has better performance than CNN which can be supported by the result table given in [1]. For negative images (inverted images), CNN performs very poorly as shown in part b of question 1. Whereas for SAAK, during KLT transform, we augment the kernel using its negative part to take care of negative training

images so that at every stage, we have original and augmented kernel which can be used for original or inverted images.

ERROR ANALYSIS

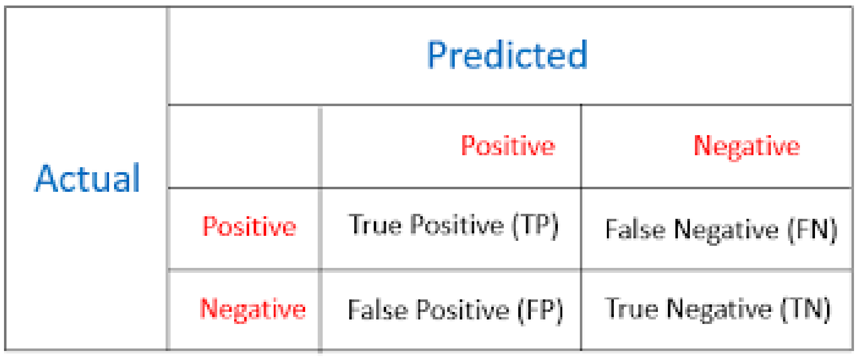
Task: Compare and comment on classification error cases and propose method to improve performance of CNN and SAAK.

EXPERIMENTAL RESULTS



```
[[ 976    0    0    0    0    0    1    1    1    1]  [[ 974    0    1    0    0    0    3    1    1    0]
 [   0 1133    0    0    0    1    1    0    0    0]   [   0 1131    1    0    0    1    1    0    1    0]
 [   0    0 1028    1    1    0    0    2    0    0]   [   0    1 1028    0    1    0    0    2    0    0]
 [   0    0    0 1007    0    3    0    0    0    0]   [   0    0    2  999    0    6    0    1    2    0]
 [   0    0    1    0  977    0    2    1    0    1]   [   0    0    0    0  975    0    5    1    0    1]
 [   1    0    0    6    0  884    1    0    0    0]   [   1    0    0    2    0  887    1    0    1    0]
 [   6    2    0    0    2    7  939    0    2    0]   [   2    2    0    0    1    2  950    0    1    0]
 [   0    3    3    0    0    0    0 1019    2    1]   [   0    2    5    0    0    0    0 1017    1    3]
 [   3    0    2    4    1    3    0    1  958    2]   [   2    0    2    1    1    2    0    0  964    2]
 [   0    2    0    2    8    5    0    3    2  987]]  [   0    1    0    0    8    3    1    1    1  994]]
```

Classification error of a model is number of samples incorrectly classified (false positive or false negative) to the total number of samples. It can be elaborated more from the figure 31. From the figure it is clear that diagonal elements should be higher to achieve better classification error. Table above shows confusion matrix for SAAK and CNN. As we can observe, for 60000 training images and 10000 test images, I have plotted confusion matrix for test data. We can observe that diagonally leading elements are very high almost equal to 1000 per digit. Off-diagonal elements are the elements which are not classified correctly.



		Predicted	
		Positive	Negative
Actual	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Figure 31– Classification error representation

Classification error usually arrives due to multiple reasons like data imbalance, wrong pre-processing or insufficient pre-processing, less training data or overfit.

DISCUSSION

We have used online dataset MNIST which has no data imbalance hence even for less training data, given its many advantages, it gives good classification accuracy. But if the dataset has data imbalance, then SAAK transform method does not produce similar accuracy if imbalance is not present.

From confusion matrix and accuracy plots stated above, CNN still has slightly better classification accuracy compared to SAAK using SVM classifier assuming 60000 data is sued to train. From all the different classification error types, classification errors are almost the same for CNN and SAAK and there is not much of a difference. Both CNN and SAAK, percentage of error is around 2% which is very low and hence we can conclude that both the models perform exceptionally good.

PERFORMANCE IMPROVEMENT:

CNN performance improvement can be divided into 3 main parts-

Improvement in Data:

We can tinker with data such as data augmentation, feature selection based on importance or correlation or just simply getting more data.

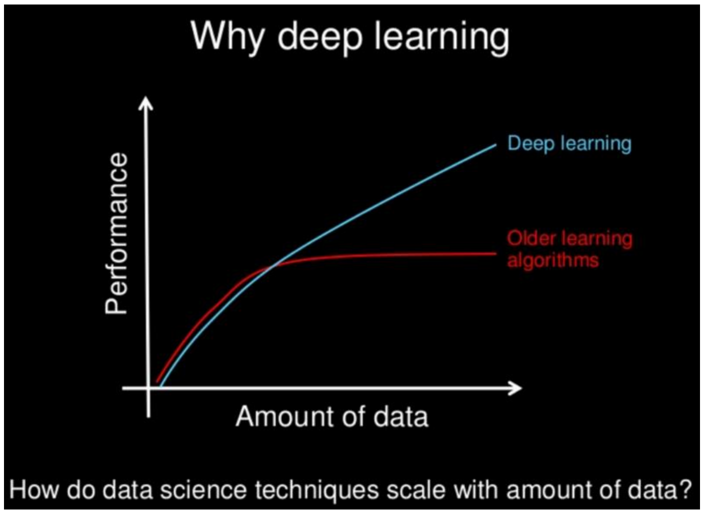


Figure 31– Data vs Performance curve

We can see that performance of CNN improves as we get more. We can transform the same data or get more.

Improvement in Algorithm:

Since we do not know which algorithm performs better, we can use trial and error for available algorithms to evaluate the performance and choose better algorithm.

Resampling Methods:

We can resample the data to tackle data imbalance to get better classification accuracy.

I have explained the methods for improvement in part c of first question.

SAAK Improvements:

SAAK transform method can be made generalized by combining feature extraction and classification model by making it end to end architecture.

We selected best features at every stage using F measure. But this does not always guarantee best features. We have evaluated performance using chi-sqaure or correlation matrix or mutual information to check which features contribute more towards better classification accuracy. Hence, we can find more effective method to extract important features from available features.

We can also expand the number of applications where SAAK transform inverse can be used and compare its efficiency with Generative Adversarial Networks(GAN).

REFERENCES

[1] A SAAK TRANSFORM APPROACH TO EFFICIENT, SCALABLE AND ROBUST HANDWRITTEN DIGITS RECOGNITION (arXiv:1710.10714v1 [cs.CV] 29 Oct 2017)

[2]https://www.kdnuggets.com/2015/04/preventing-overfitting-neural-networks.html/2

[3] https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

[4] https://en.wikipedia.org/wiki/Loss_function

[5] http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/

[6]https://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/

[7]https://www.researchgate.net/publication/320727652_A_Saak_Transform_Approach_to_Efficient_Scalable_and_Robust_Handwritten_Digits_Recognition

[8] On Data-Driven Saak Transform (arXiv:1710.04176v2 [cs.CV] 14 Oct 2017)

[9] https://machinelearningmastery.com/improve-deep-learning-performance/