EEE 569 DIGITAL IMAGE PROCESSING
HOMEWORK #4

ABINAYA MANIMARAN

MANIMARA@USC.EDU

SPRING 2018

SUBMITTED ON 04/29/2018

PROBLEM 1
CNN TRAINING AND ITS APPLICATION TO THE MNIST DATASET

A) CNN ARCHITECTURE AND TRAINING

➢ ABSTRACT AND MOTIVATION:

The main purpose of this part of the question is to understand deeply the Convolutional Neural Networks and its various layers. The understanding will help us to create CNNs for the MNIST Digit Image Classification. Over the next section, I have explained each layer and its operational mechanism. Convolutional Neural Networks get their name from "Convolution" Operation. With this understanding, let us proceed to the next session. I have skipped Experimental Results and Discussion sections for this part of the question since it needs theoretical explanation of various questions.

➢ THEORETICAL ANSWERS:

1. EXPLANATION FOR: 1) THE FULLY CONNECTED LAYER, 2) THE CONVOLUTIONAL LAYER, 3) THE MAX POOLING LAYER, 4) ACTIVATION FUNCTION, AND 5) SOFTMAX FUNCTION

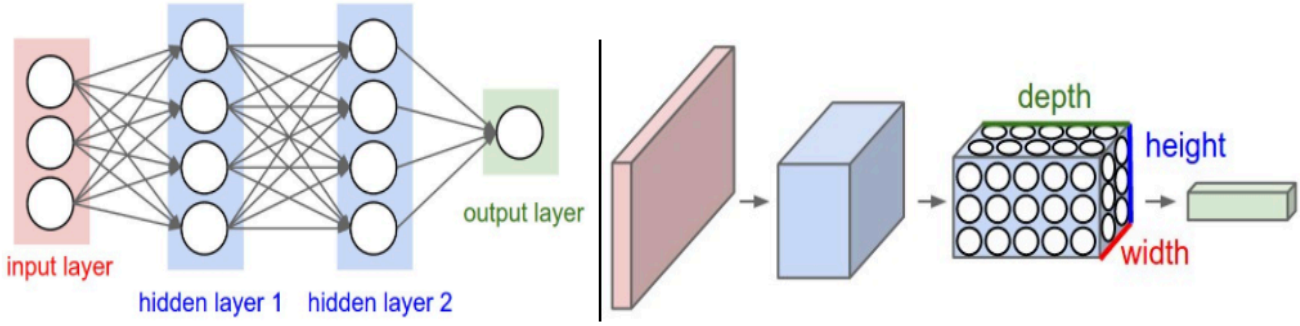
We know that a regular neural network, gets input via the input layer and passes them through various hidden layers to extract features basically. Each hidden layer is made up of many neurons called as hidden units. The output layer gives out the output of classification and mostly are preferred to be probabilities for every possible class.

The main problem of regular neural networks is that they don't scale well for images. For example, a fully connected layer in the first hidden layer of a neural network will have:

number of weights = width * image height * number of channels

Thus, number of weights might be very huge as the image size increases. This would lead to huge number of parameters and might result in overfitting.

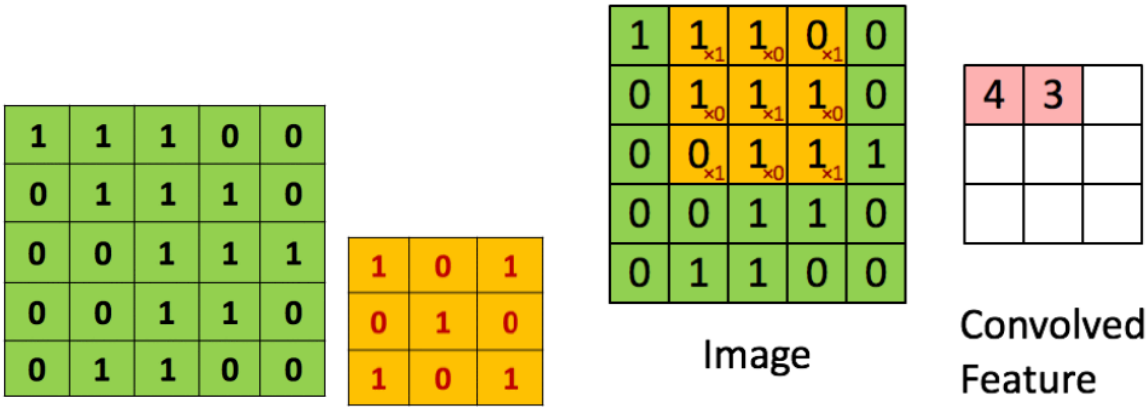
Thus, Convolutional Neural Network takes advantages of the inputs as images. Unlike the regular Neural Networks, the layers of a Convolutional Neural Networks are arranged in a three-dimensional way with dimension sizes be width, height and depth as shown below.

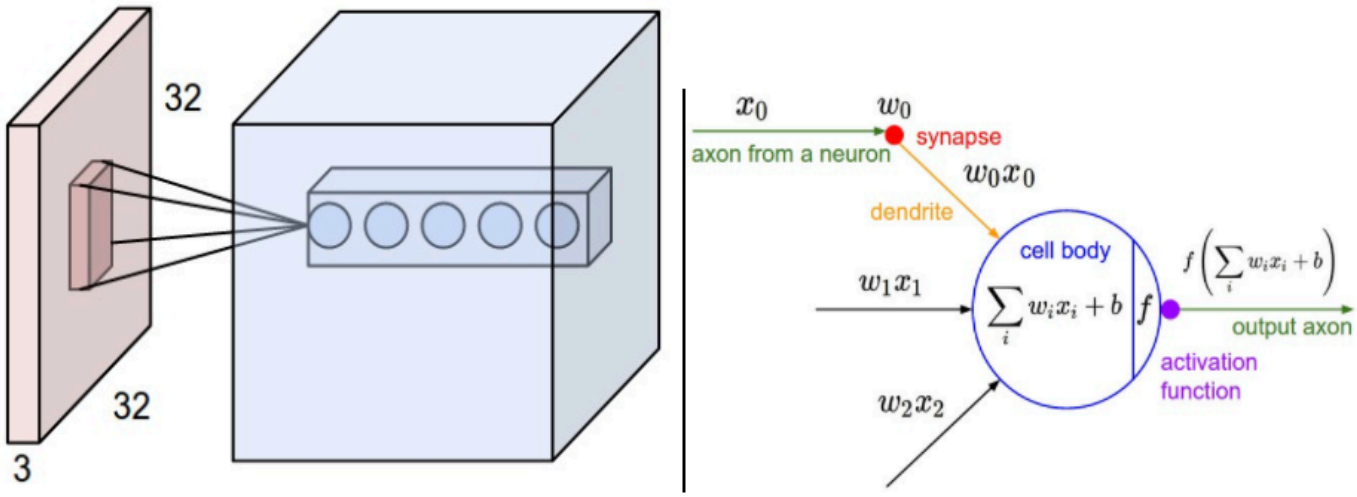


CONVOLUTIONAL LAYER:

Convolutional Layer is the main building block and one of the important layers of Convolutional Neural Network. A Convolutional layer uses a set of learnable filters. Typically filter sizes are 3*3 or 5*5 or 7*7. And they can be arranged in a 3-dimensional way along the depth. During the Forward Propagation of the, each filter is convoluted across the weight and height of the input image. That is they perform dot products between the filter and image pixel intensity values. We slide the filter along the value of input image width and height and they produce a 2-dimensional activation map.

Intuitively, the CNN learns the filters that activate that identifies some colors or patterns in the image. These filters might be of any type like averaging filters, texture detection filter, edge detection filters. As we proceed in applying these filters on the image, each filter convolution will provide a separate 2-dimenstional activation map. These maps are stacked along the depth of the image they form the output volume. An example of convolution is shown below.





The above figures show an example input image with a filter. It also shows how a sample convolution happens and the activation map is applied.

There are hyper parameters associated with this main block of the neural network. If tuned properly, they might lead to not over fitting and better testing results. They are listed below:

- Depth of the Convolutional Layer or Number of Filters:

  Input Convolutional Layer takes input from the raw image and goes through convolution as many times, the number of filters is available. As I mentioned earlier, these filters might detect different things like blobs, edges, textures, colors etc.
- Stride:

  Stride is the parameter which lets us know, how to move the filter along the image pixel intensity values. For example, if stride value is 2, while convolution is performed, one-pixel value is skipped. Convolution is performed once for every 2 pixels.
- Padding:

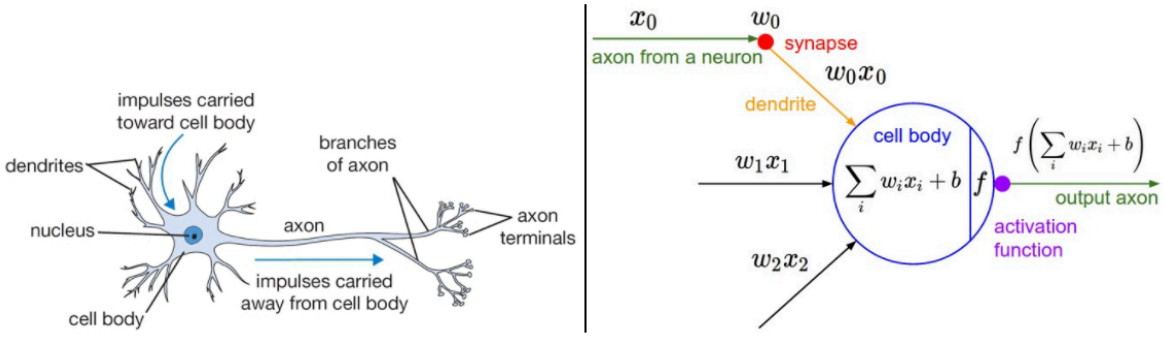
  Padding is essential to perform convolution for the boundary pixels. There are many ways to perform padding. Zero-padding, Pixel Replicating are few such ways. Again, the size of padding is a hyper parameter.

Thus, the Convolution Layer can be explained with dimensions as below:

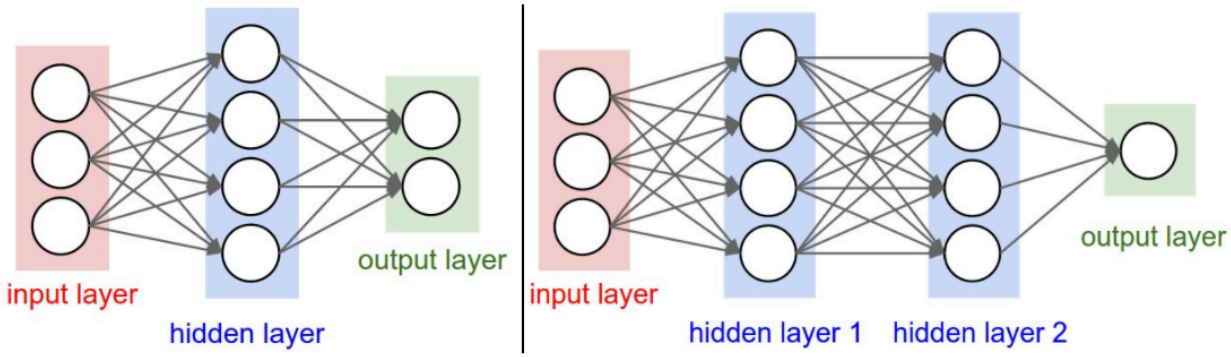
- Input Volume size: W1 * H1 * D1
- Hyper parameters are:
  - Number of filters K
  - Spatial extent F (like 3*3 or 5*5 etc)
  - Stride S
  - Amount of padding P
- Output Volume Size: W2 * H2 * D2 – width and height are computed by symmetry
  - W2 = (W1 – F + 2P)/S + 1
  - H2 = (H1 – F + 2P)/S + 1
  - D2 = K
- Parameter Sharing introduces F * F * D1, weights per filter, and a total of weights with dimension (F * F * D1) * K and K biases

FULLY CONNECTED LAYER:

A fully Connected Layer is the traditional Multi-Layer Perceptron. The Layer is termed as Fully Connected since every neuron in the previous layer is connected to every neuron in the next layer. These fully connected layers can be seen as a way of learning non-linear combinations of the features.

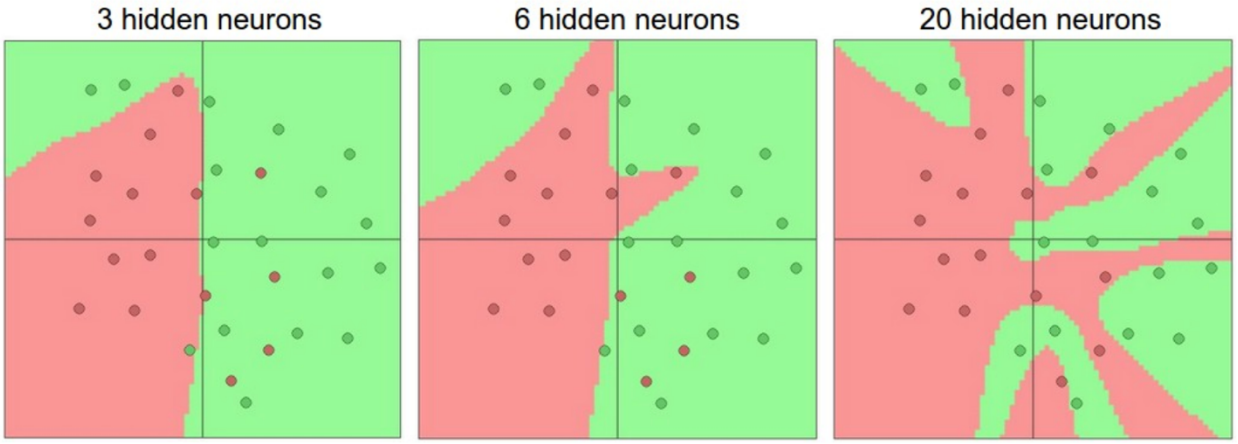


The comparison of human brain with the artificial neural network is shown in the previous page. A sample architecture of Fully Connected Neural Network with one hidden layer and two hidden layers are shown below. We can notice that each neuron from previous layer is connected to all the neurons of the present layer.



This layer, like Convolutional Layer has many hyper parameters. The main two is listed below:

- Number of hidden units in a layer
    - Choosing number of hidden units in a layer is important to prevent over fitting or under fitting of training data. This in turn is important since it might decrease the testing accuracy
- Activation Function
    - There can be many types of activation function. This is chosen for every layer. Explanation about various activation function are given in the next section. The final layer activation function is many a times chosen to be Softmax function since it gives the probability output.



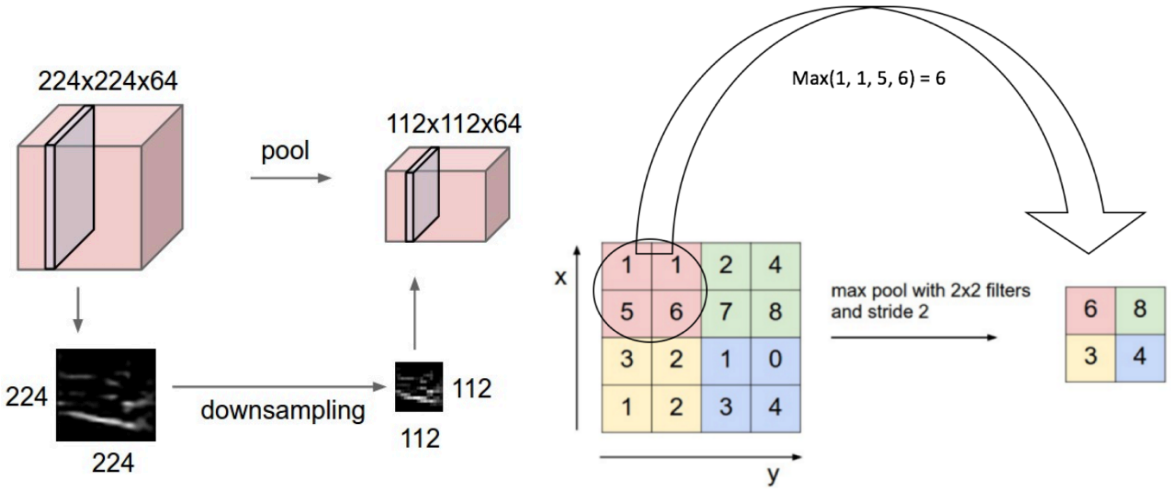
An example effect of number of hidden neurons in a layer is shown above. As we can see 3 – hidden neurons underfits the training data. 6 – hidden neurons seem to be optimal. And 20 – hidden neurons very much over fits the training data.

Hence this shows choosing optimal number of hidden neurons is as important as choosing any other main parameter in the Convolutional Neural Network like number of layers or type of layers.

MAX POOLING LAYER:

Spatial Pooling, also called as subsampling or downsampling reduces the dimentionality of the input. There are different types of Spatial Pooling. Mainly they can ben Max Pooling, Min Pooling or Average Pooling.

Max Pooling is done by defining a neighborhood and taking the maximum of the neighbor and is used as a replacement pixel intensity value. The below figure shows how Max Pooling is done on the neighbor of 2*2. Thus a 2*2 window size is used for Max Pooling.
(Source: http://cs231n.github.io/convolutional-networks/#pool)



Usually Max Pooling layers are added after a Convolutional Layer with some activation function like ReLU. Thus, it reduces the spatial size of the representation, thus reducing the parameters to be trained. After Pooling, the depth dimension remains unchanged. While other type of Pooling exists like Averaging, taking L2 Norm etc., they are not recently used since Max Pooling is considered to be out performing any other such Pooling operations

More generally, Pooling layer has the following mentioned dimensions:

- Input Volume Size: W1 * H1 * D1
- Hyper parameters are:
  - Spatial Extent F
  - Stride S
- Output Volume Size: W2 * H2 * D2
  - W2 = (W1 – F) / S + 1
  - H2 = (W1 – F) / S + 1
  - D2 = D2
- Introduces zero parameters since there is always fixed output given an input
- Also, it is known that zero padding is not common for Pooling layers, since they are not necessary

Though the hyper paramters can take different values, only two configurations are used in practice commonly. They are F=3, S=2 and F=2 and S=2

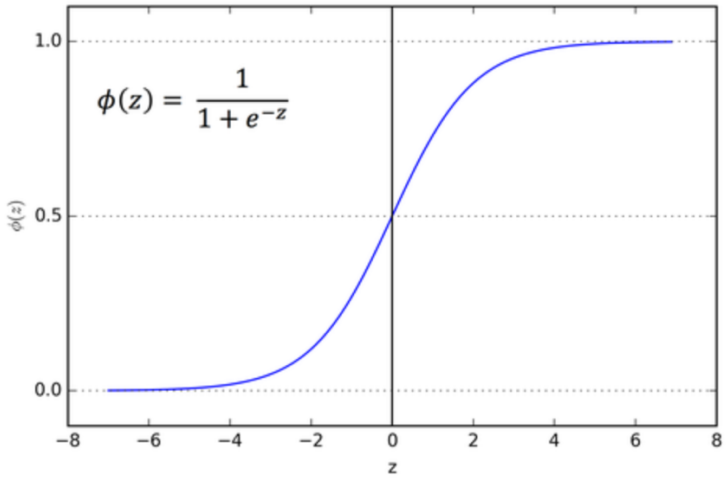
ACTIVATION FUNCTIONS:

Activation Function, also known as Transfer Function determines the output of the Neural Network Layers. There can be Linear or Non-Linear Activation Functions. The results may be between -1 to +1 or between 0 to 1, depending upon the function. Some activation functions are given below.
(Source: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6)

- **Sigmoid Function:**

The sigmoid function looks like a S – shaped one which gives output between 0 and 1. Therefore, the sigmoid function is mostly used when we need probability as output. The output values of the sigmoid function can be interpreted as probability of activation. The function is differentiable (meaning, we can find the slope of the sigmoid curve given any two points on the curve). This function is also monotonic. The SoftMax function is a more generalized version of sigmoid function, used for multi-class classification. The curve is give below:

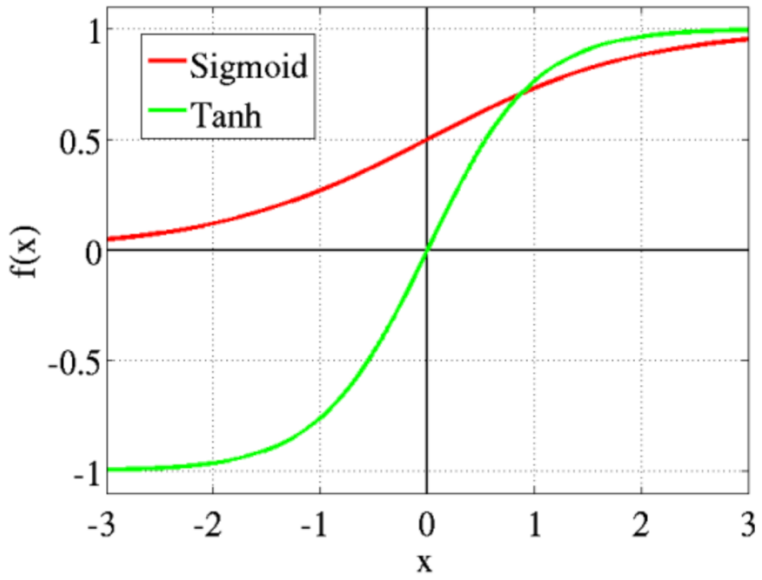
$$\sigma(x)=1 / (1+e^{-x})$$



- **Tanh Function:**

Tanh functions looks similar S-shaped function like sigmoid function, but outputs the value between -1 to +1. The function is also differentiable and monotonic. The main advantage of using Tanh function is that they negative inputs are mapped to very strongly negative values. The Tanh function is mainly used for classification between two classes. And both tanh and logistic sigmoid activation functions are used in Feed-Forward Networks.

$$\tanh(x)=2\,\sigma(2x) - 1$$

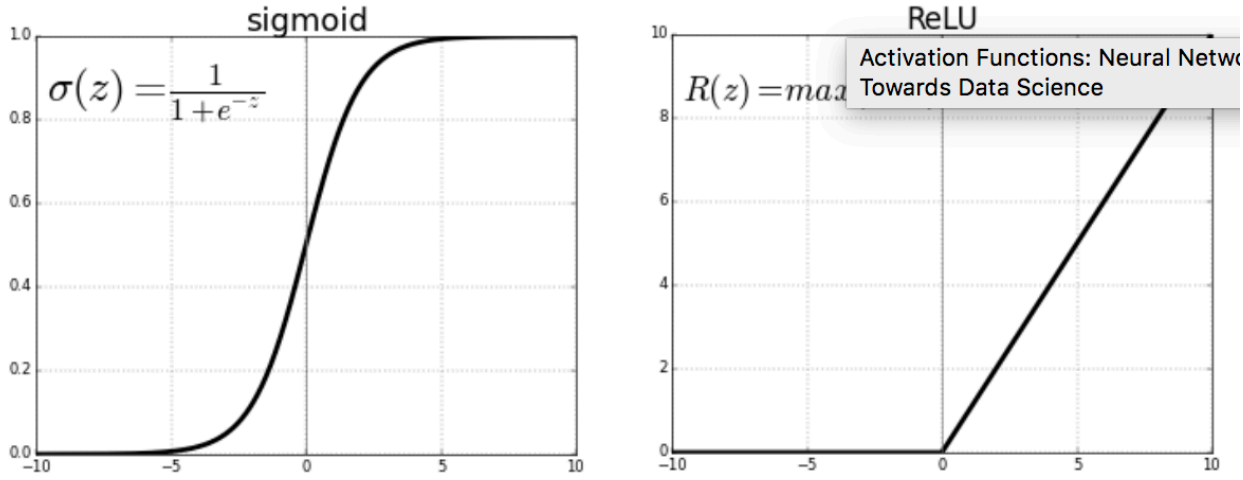


- **ReLU (Rectified Linear Unit) Function:**

ReLU is the most famous and widely used Activation Function. It performs the below mentioned function:

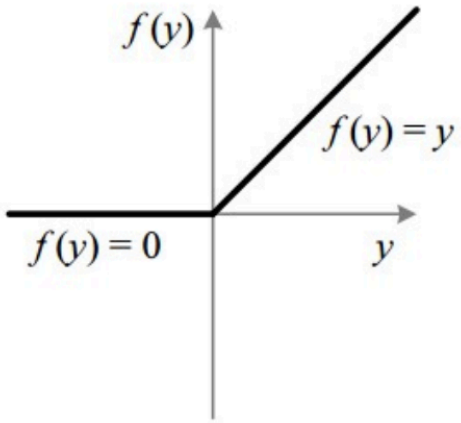
$$f(x)=\max(0,x)$$

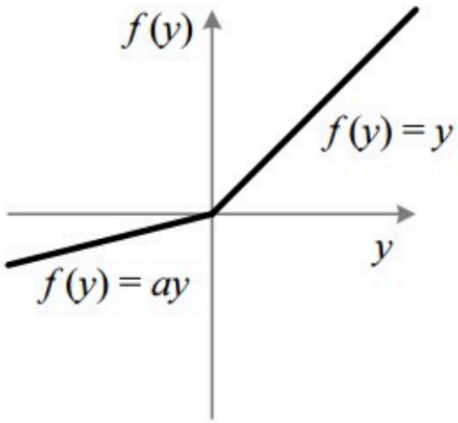
As it shows below, the activation function is simply thresholded at zero. This function is proved to greatly speed up the convergence process. Another advantage of ReLU function is that is not computationally expensive as Sigmoid or Tanh function. This can be implemented by simple thresholding. The function is differentiable and monotonic. Another differentiating factor is that the derivative of the function is also monotonic. The main disadvantage is that all negative values immediately are transferred to zero and hence model fitting ability decreases. The resulting graph will show that mapping of negative values are not proper.



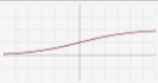
- **Leaky ReLU:**

I explained the disadvantage of the Rectified Linear Unit that the negative values are immediately thresholded to zero and this makes the model fitting not proper. To solve that problem, Leaky Rectified Linear Unit is used. This helps to increase the range of ReLU function and the range is between – infinity to + infinity. This function is also monotonic and their derivatives are also monotonic.





- **A Table of Activation Functions:**

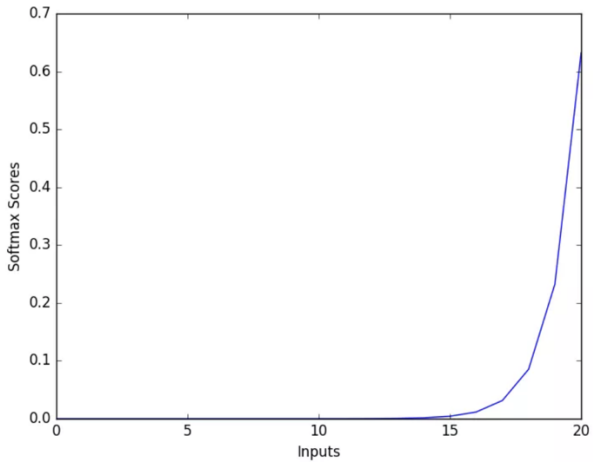
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

SOFTMAX FUNCTION:

Softmax function, also called as normalized exponential function, is a generalization of sigmoid or logistic function.

- Properties:
  - The function squashes a K-dimensional vector to arbitrary real values in the range of 0 and 1.
  - The sum of probability output values adds up to 1 since it is normalized.
- Usage:
  - Used in Multi-class classification using a model
  - This function is mainly used in the last layer of any Neural Network to push the previous layer inputs to a probability map for every class.
  - The predicted probability of j^{th} class, given a sample vector x and a weight vector w is given as follows:

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$



SOFTMAX FUNCTION	SIGMOID FUNCTION
Used for multi-class classification	Used for binary classification
The probability outputs sum will add to 1	The probability outputs sum need not be 1
Used in different layers of Neural Network. Mainly used in the last layer	Used as activation functions while building the Neural Networks
The high value will have the higher probability when compared to other values	The high value will have the high probability but not necessarily the higher probability

2. OVER FITTING AND TECHNIQUES TO REDUCE THEM:

- Any Machine Learning algorithm is supposed to be suffering from two main issues – Under-fitting and Over-fitting.
- The model is said to be underfitting if the accuracy of the validation set is higher than the accuracy of the training set. Thus, on the whole if the model performs bad, the model is said to be under-fitting.
- Alternatively, over-fitting happens when the model fits too well on the training set. Thus, it makes it hard to generalize to new test samples and the testing accuracy would be very less. The training accuracy will be very much higher than the validation accuracy on an over-fitted model.

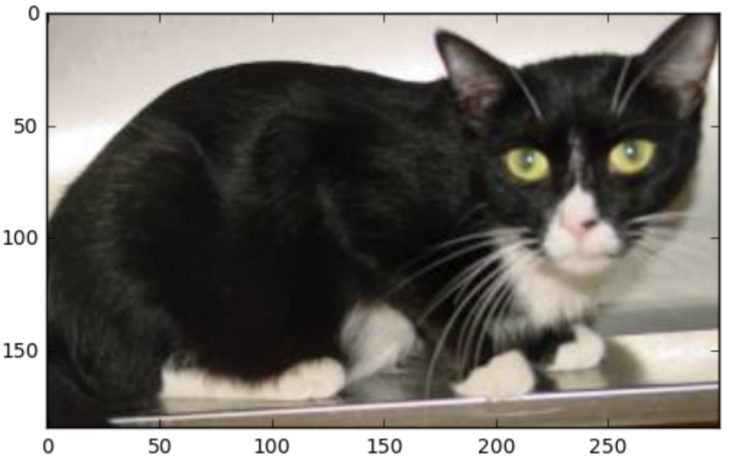
STEPS TO AVOID OVER-FITTING:

- **Add more data:**

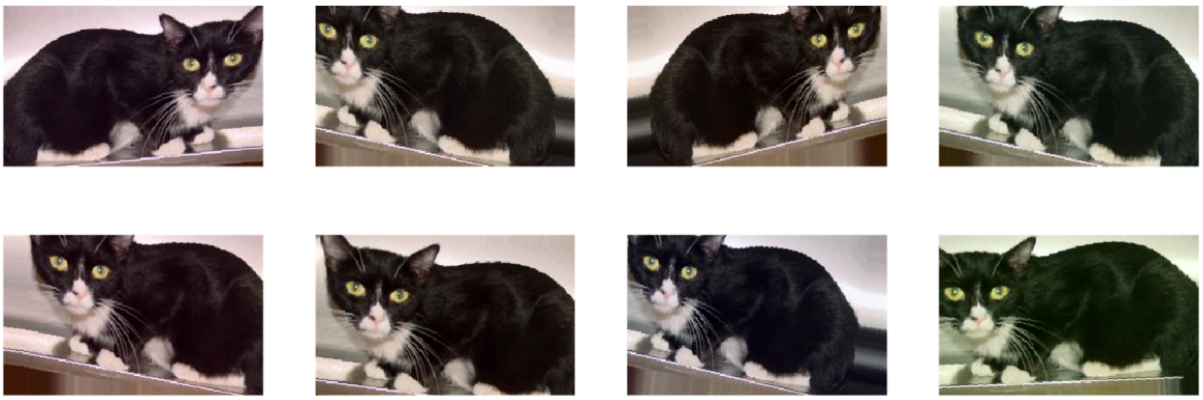
  As the step clearly says, we can collect more data. Collecting more data will bring many variations in the training set. Thus, reduces the chances of overfitting. On real time, collecting more data might not be a feasible solution which may prove to be very costly in terms of labor and money.

- **Use Data Augmentation:**

  Data Augmentation is one way of generalizing the given data and increase the count of training data by introducing different varieties of the existing data. Data Augmentation includes processes like randomly rotating the image, zooming in and out of the image, adding color filters to the image etc. These can be done only on the training set and not on the cross-validation set, since they ideally are considered to be the test set within the cross-validation loop. An example of different types of Data Augmentation is shown below.



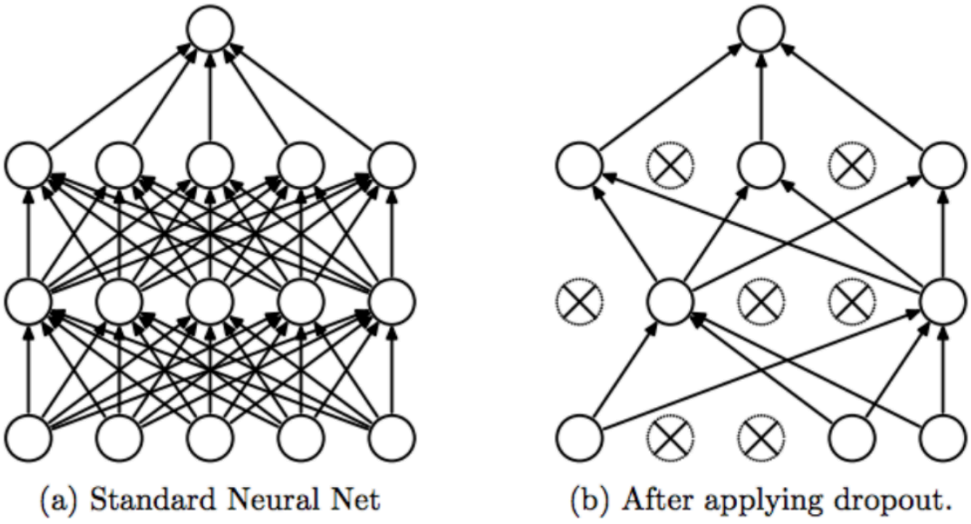
Original image



Augmented cats

- **Use architectures that generalize well:**
  This is supposedly a trial and error method, where we can try different architectures that can generalize well to get a higher cross-validation accuracy and testing accuracy

- **Add regularization**
  Regularization of training the model can be done in 3 main ways. They are:
  - Dropouts: Deletes a random sample of the activations, makes them zero while training the model. This can be added to any layer in the Convolutional Neural Network like Convolutional Layer, Fully Connected Layers etc. This can be done on a trial and error basis by starting with a low dropout value and increasing it to get good results.



Visualization of dropout

(Source: https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d)

- L1 Regularization – Also called as Lasso Regression (Least Absolute Shrinkage and Selection Operator) adds absolute value of magnitude of coefficient as penalty term to the loss function

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Cost function

- L2 Regularization – Also called as Ridge Regression adds squared magnitude of coefficient as penalty term to the loss function

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Cost function

(Source: https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c)

- **Reduce architecture complexity**

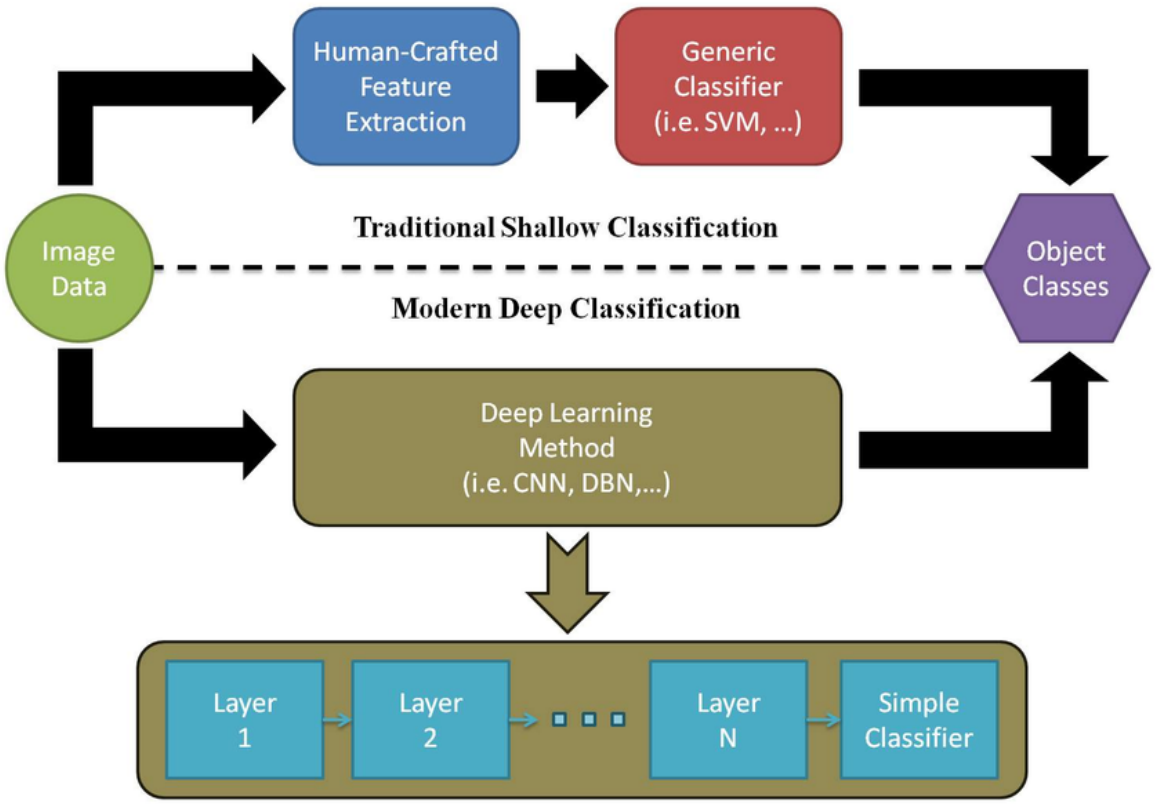
  Reducing network complexity is one form of regularization that can deal with overfitting

3. CNN WORKING BETTER THAN TRADITIONAL METHODS (BASED ON COMPUTER VISION)

This is an important question since the world is going towards Deep Nets to solve many real-world problems. More importantly Deep Nets, mainly Convolutional Neural Networks are used in many Computer Vision problems, replacing the Traditional methods to solve them. This has been accepted by many CV based researchers since, surprisingly Deep Nets yield accurate results than traditional ones.

Some main reasons that they Deep Nets are working very well are discussed in this section. First of all, the feature extraction done by traditional methods involve humans. But in Deep Nets like CNN, feature extraction itself is learnt by itself by a learning algorithm and adapts to the present errors to improve them. Deep Nets do not see a picture as the way a human sees them and automatic feature extraction by layers like Convolutional Layers enhance the method of Prediction. For example, in Image classification problem, the main feature extraction that humans would do for traditional methods would be like texture classification, edge detection, blob detection, color space identification etc. But in the case of CNN, the needed features are learnt by itself by the Convolutional Layers while the number of filters, stride etc., become hyper parameters.

The next reason I could see that CNN work much better than any other traditional algorithms is that they learn the non-linearity present in the features and train the model much better. Non-Linearity is one huge advantage present in the variations between classes. If they are learnt properly, any model can learn better. Thus, CNN takes advantage that the input are images and does Convolution in the first few layers. Convolution with many types of filters are an advantage that they learn many different types of variations present in the images. They get the ability to also learn the non-linearity along with linearity.



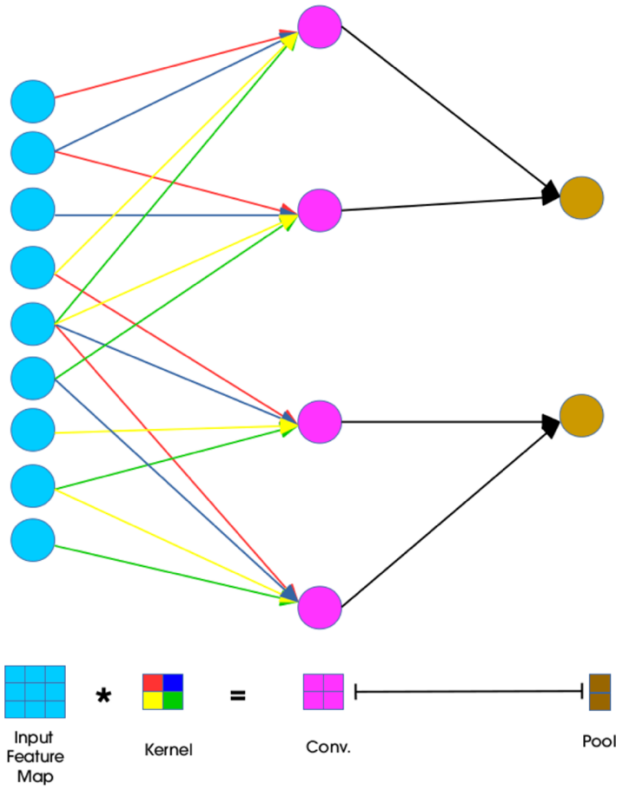
Next reason, mainly in the case of Image Classification the deep learning networks like CNN are very complex, deep in layers, hugely determined in terms of degrees of freedom. They have developed such that they are much more than multi-layer perceptrons. They have been proved to be one of the best methods to do Image classification. There have been many research papers on comparison of CNN/Deep Learning based Image Classification with traditional based methods. Most of the papers have proved that Deep learning is better than any other traditional high-level methods like ensemble learning or SVM or even Boosting techniques.

Thus, my take on the reasons mainly is, any deep learning network learns the Feature Extraction method too along with the weights for Classification. This gives us better automatic extracted features. And the layers of feature detection, number of filters that needed to be used, learning rates of every learning algorithm are hyper parameters. And there are many ways to reduce overfitting and underfitting. Deep Learning might look like an alchemy today, but eventually will learn to practice it as chemistry.

4. BACK PROPOGATION (BP) OPTIMIZATION TO TRAIN CNN

Convolutional Neural Networks are a well-known biologically inspired variation of Multilayer Perceptions. Neurons in CNN share weights unlike in MLPs, where each neuron has a separate weight. Thus, this sharing of weights has reduced the overall complexity of training the weights. These shared weights are able to perform convolutions on the data with the convolution filter being formed using the weights. This is what I mentioned in the earlier section that the filter weights are learnt by the CNN and hence the entire feature extraction itself is adaptive and improves after every epoch.

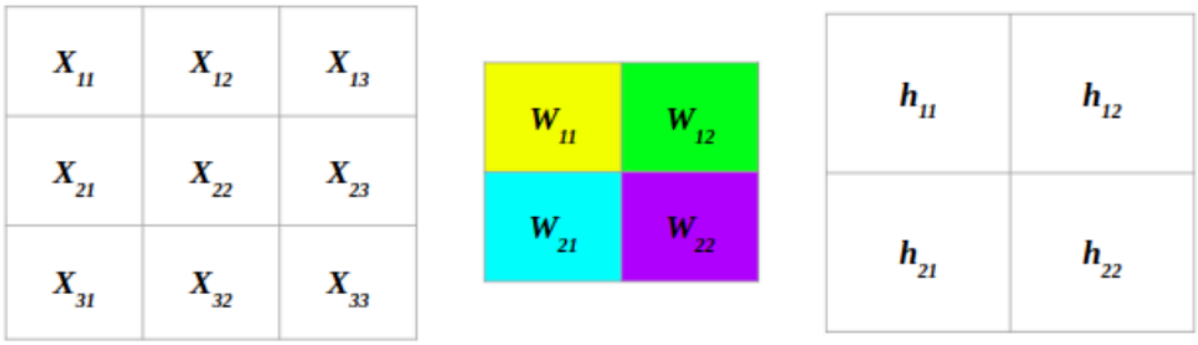
The famous Back Propagation algorithm for weights optimization performs both forward and backward propagation in every epoch. For example, in the convolution step, the kernel us flipped by 180 degrees and slid across the input feature map in steps of strides. As it is shown below the forward propagation gets values from the inputs and applied weights to perform convolution. (Source: https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199)



In the forward pass, the filters are flipped and performed for Convolution. If they are not flipped, it is cross-correlation. The Cross Correlation and Convolution is explained below.

$$(I \otimes K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n) K(m,n)$$

$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i-m, j-n)K(m,n)$$
$$= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n)K(-m,-n)$$



I have shown an example above. Left Matrix is the Image matrix. The middle one is the filter matrix with the weights as coefficients. The right matrix is the convoluted output. The method of convolution is shown below.
(Source: http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/)

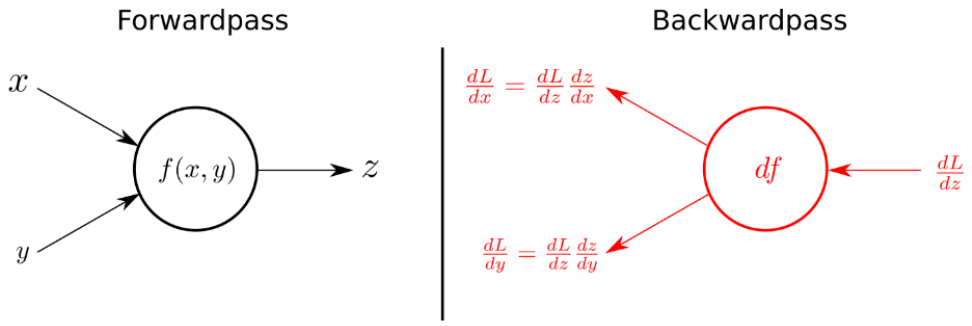
$$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$$

$$h_{12} = W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23}$$

$$h_{21} = W_{11}X_{21} + W_{12}X_{22} + W_{21}X_{31} + W_{22}X_{32}$$

$$h_{22} = W_{11}X_{22} + W_{12}X_{23} + W_{21}X_{32} + W_{22}X_{33}$$

The next part is Back Propagation where the gradients of the output is found. Back Propagation can be seen as a Forward Propagation from the output to input. The aim is to calculate the gradient in x and w vectors given the output of the Forward Propagation.



$$\partial h_{ij} \text{ represents } \frac{\partial L}{\partial h_{ij}}$$

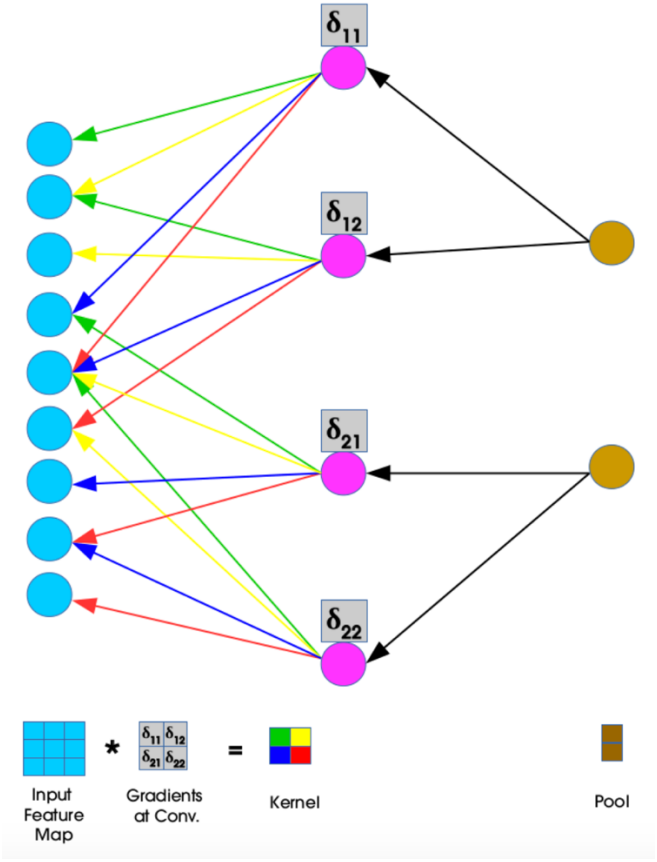
$$\partial w_{ij} \text{ represents } \frac{\partial L}{\partial w_{ij}}$$

$$\partial W_{11} = X_{11}\partial h_{11} + X_{12}\partial h_{12} + X_{21}\partial h_{21} + X_{22}\partial h_{22}$$

$$\partial W_{12} = X_{12}\partial h_{11} + X_{13}\partial h_{12} + X_{22}\partial h_{21} + X_{23}\partial h_{22}$$

$$\partial W_{21} = X_{21}\partial h_{11} + X_{22}\partial h_{12} + X_{31}\partial h_{21} + X_{32}\partial h_{22}$$

$$\partial W_{22} = X_{22}\partial h_{11} + X_{23}\partial h_{12} + X_{32}\partial h_{21} + X_{33}\partial h_{22}$$



The weight sharing is the important strategy in Convolutional Neural Networks. They significantly reduce the number of parameters that to be learned. Also, the gradient/derivatives of forward and backward propagations differ depending on what layer the propagation is going through. Thus, Forward Propagation uses flipped weight matrix as kernel and Backward Propagation uses gradient matrices as kernel.

B) TRAIN LENET-5 ON MNIST DATASET

➢ ABSTRACT AND MOTIVATION:

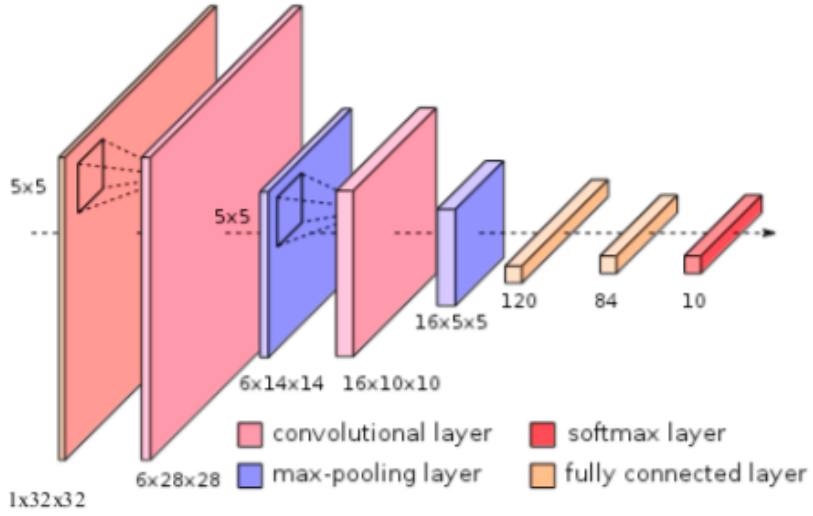
The purpose of this portion of the question is to implement and get hand on experience on CNN using any available opensource. I have used Python with Keras, which is built on top of Tensor Flow. The structure details of LeNet-5 to be used is given along with the question. I have explained the implementation details and various results in the upcoming sections. Another purpose of this question is also to improve the accuracy on Training MNIST dataset given the LeNet-5 CNN structure.

➢ APPROACH AND PROCEDURES:

MNIST (Modified National Institute of Standards and Technology) Dataset is a famous dataset consisting of handwritten digits commonly used in various Image Processing models. The dataset consists of 60,000 training samples and 10,000 testing samples. This dataset considered is a subset of larger handwritten digits dataset available from NIST. The digits are size-normalized and centered in the fixed-size image. Some examples from the dataset is shown below.

The given CNN Architecture for this part of the question is LeNet-5 Architecture. It has 5 layers on the total.

LAYER	LAYER NAME	#FILTERS	ACTIVATION	OTHER DETAILS
1	Convolutional Layer	6	Relu	Followed by Max Pooling layer
2	Convolutional Layer	16	Relu	Followed by Max Pooling layer
3	Fully Connected Layer	120	Relu	-
4	Fully Connected Layer	84	Relu	-
5	Fully Connected Layer	10	Softmax	Outputs probabilities of every class



Given this base architecture, I have changed many parameters like:

NETWORK PARAMETER	VALUES CHANGED
Activation Function	Linear, Sigmoid, Tanh, Relu and Leaky Relu
Dropouts	0.2 and 0.5
Optimizers	SGD, Adadelta, Adam, Adagrad and RMSProp
Learning Rate	0.01, 0.1 and 0.5
Decay	0.01, 0.1 and 0.5
Weight Initialization	Glorot Uniform and Random Uniform

- For each parameter, **100 epochs** were performed. On an average each epoch took 7 seconds in a Mac with 2.3 GHz Intel Core i5 configuration.
- Hence, I used Google Collab with GPU settings to run the entire parameter changing CNN.
- On an average each epoch took 3 seconds with GPU based Google Collab and hence 5 minutes to train the dataset for one set of parameters
- Thus, using different For Loops, the code was automated to run for different parameters and the CNN Class code is explained below
- To get the Testing accuracy after every epoch, the validation set was given as the testing set
- After each epoch, the history of Train accuracy, Train Loss, Validation Set accuracy and Validation Set Loss were saved
- The Training epochs vs Training accuracy and Testing accuracy were obtained and this was also automated

Algorithm Implemented (Python and Keras):

main() Function:

- Different lists were initialized with the above mentioned parameters called as activation_list, dropout_list, optimizer_list, lr_decay_list, weight_initialization_list
- Epochs is defined to be 100
- Five for loops were run to loop over all the parameters
- Object for LeNet_5_MNIST_Dataset() class is defined
- Call load_dataset() function for the object
- Call preprocess_dataset() function for the object
- Call fit_model() function for the object
- Save all the outputs from the returned model into a pandas dataframe

LeNet_5_MNIST_Dataset() class:

__init__() function:

- Define all the parameters for the class
- Image rows, columns, number of classes, activation function, dropout value for drop out layer, Optimizer, learning rate and decay for the optimizer, method of weight initialization, batch size and epochs to train are passed in the Class Initialization function from the outer main for loops.

load_dataset() function:

- Train and test dataset is loaded using mnist.load_data()

preprocess_dataset() function:

- Reshape the given MNIST dataset
- Convert pixel values to float and divide by 255 to normalize the values between 0 and 1
- Encode the class values by one hot encoding using keras.utils.to_categorical()

fit_model() function:

- Define the number of filters for every layer
- Define a model – Sequential
- Add a Convolutional layers with padding with the given filter size and activation unit
- Define Dropouts layers with parameter passed
- Define Max Pooling Layers post the Convolutional Layers with strides defined
- Add Fully connected layers with given activation units
- Add a Fully Connected layer in the end with Softmax activation function
- Define the optimizer and compile
- Define learning rates and decays as passed to the class
- Fit the model with the defined layers and parameters
- While fitting the model, give testing dataset as the validation set
- Also get the training and testing score after fitting the entire model
- Return the model and history of the epoch results

➢ EXPERIMENTAL RESULTS:

Iterations	Activation	Dropout	Optimizer	lr_deca	WeightInitialization	Train loss	TrainAccuracy	Test loss	TestAccuracy
0	Leaky ReLU	0.2	SGD	0.01	glorot_uniform	0.095464835	0.970716667	0.087204428	0.9708
1	Leaky ReLU	0.2	SGD	0.01	RandomUniform	0.398764454	0.8868	0.378419526	0.8943
2	Leaky ReLU	0.2	SGD	0.1	glorot_uniform	0.087513612	0.9734	0.079440736	0.9745
3	Leaky ReLU	0.2	SGD	0.1	RandomUniform	0.255068028	0.9272	0.241367356	0.9345
4	Leaky ReLU	0.2	SGD	0.5	glorot_uniform	0.082514242	0.97495	0.072520468	0.9772
5	Leaky ReLU	0.2	SGD	0.5	RandomUniform	0.154831852	0.954516667	0.145208922	0.9557
6	Leaky ReLU	0.2	Adadelta	0.01	glorot_uniform	2.032271061	0.551283333	2.022849717	0.5664
7	Leaky ReLU	0.2	Adadelta	0.01	RandomUniform	2.30221951	0.112366667	2.302205232	0.1135
8	Leaky ReLU	0.2	Adadelta	0.1	glorot_uniform	1.555623054	0.6747	1.548089216	0.6861
9	Leaky ReLU	0.2	Adadelta	0.1	RandomUniform	2.302313418	0.112366667	2.302291515	0.1135
10	Leaky ReLU	0.2	Adadelta	0.5	glorot_uniform	1.268183073	0.698966667	1.247811912	0.7064
11	Leaky ReLU	0.2	Adadelta	0.5	RandomUniform	2.302124203	0.112366667	2.302103021	0.1135
12	Leaky ReLU	0.2	Adam	0.01	glorot_uniform	0.004073944	0.999066667	0.024203964	0.9925
13	Leaky ReLU	0.2	Adam	0.01	RandomUniform	0.00491711	0.998916667	0.019719297	0.9936
14	Leaky ReLU	0.2	Adam	0.1	glorot_uniform	14.43509743	0.104416667	14.46115503	0.1028
15	Leaky ReLU	0.2	Adam	0.1	RandomUniform	14.54631226	0.097516667	14.54819266	0.0974
16	Leaky ReLU	0.2	Adam	0.5	glorot_uniform	14.51998605	0.09915	14.49177937	0.1009
17	Leaky ReLU	0.2	Adam	0.5	RandomUniform	14.52697056	0.098716667	14.53852184	0.098
18	Leaky ReLU	0.2	Adagrad	0.01	glorot_uniform	0.082265665	0.974533333	0.071857109	0.9759
19	Leaky ReLU	0.2	Adagrad	0.01	RandomUniform	0.121286957	0.963866667	0.111187467	0.9651
20	Leaky ReLU	0.2	Adagrad	0.1	glorot_uniform	14.51756834	0.0993	14.45470776	0.1032

21	Leaky ReLU	0.2	Adagrad	0.1	RandomUniform	14.47109451	0.102183333	14.49016753	0.101
22	Leaky ReLU	0.2	Adagrad	0.5	glorot_uniform	14.5283137	0.098633333	14.57398165	0.0958
23	Leaky ReLU	0.2	Adagrad	0.5	RandomUniform	14.5283137	0.098633333	14.57398165	0.0958
24	Leaky ReLU	0.2	RMSprop	0.01	glorot_uniform	0.004041373	0.9991	0.021815 58	0.9931
25	Leaky ReLU	0.2	RMSprop	0.01	RandomUniform	0.00521661	0.9987	0.021575079	0.993
26	Leaky ReLU	0.2	RMSprop	0.1	glorot_uniform	14.47109451	0.102183333	14.49016753	0.101
27	Leaky ReLU	0.2	RMSprop	0.1	RandomUniform	13.10680249	0.1868	13.07189988	0.1888
28	Leaky ReLU	0.2	RMSprop	0.5	glorot_uniform	14.51998605	0.09915	14.49177937	0.1009
29	Leaky ReLU	0.2	RMSprop	0.5	RandomUniform	14.54872997	0.097366667	14.53529824	0.0982
30	Leaky ReLU	0.5	SGD	0.01	glorot_uniform	0.148893769	0.954716667	0.137820547	0.9572
31	Leaky ReLU	0.5	SGD	0.01	RandomUniform	2.240129857	0.22895	2.239092486	0.2304
32	Leaky ReLU	0.5	SGD	0.1	glorot_uniform	0.113141112	0.967016667	0.103140454	0.9692
33	Leaky ReLU	0.5	SGD	0.1	RandomUniform	0.353895422	0.904583333	0.338793358	0.9124
34	Leaky ReLU	0.5	SGD	0.5	glorot_uniform	0.119640533	0.9646	0.105021489	0.967
35	Leaky ReLU	0.5	SGD	0.5	RandomUniform	0.274248019	0.91995	0.262743274	0.9237
36	Leaky ReLU	0.5	Adadelta	0.01	glorot_uniform	2.17862782	0.497833333	2.17464 2837	0.5073
37	Leaky ReLU	0.5	Adadelta	0.01	RandomUniform	2.302333723	0.112366667	2.302324855	0.1135
38	Leaky ReLU	0.5	Adadelta	0.1	glorot_uniform	2.035049904	0.596383333	2.029182696	0.6036
39	Leaky ReLU	0.5	Adadelta	0.1	RandomUniform	2.301993739	0.112366667	2.301979181	0.1135
40	Leaky ReLU	0.5	Adadelta	0.5	glorot_uniform	1.890059389	0.611866667	1.887629022	0.6126
41	Leaky ReLU	0.5	Adadelta	0.5	RandomUniform	2.302290357	0.112366667	2.302292525	0.1135
42	Leaky ReLU	0.5	Adam	0.01	glorot_uniform	0.019005347	0.9943	0.023402064	0.9922
43	Leaky ReLU	0.5	Adam	0.01	RandomUniform	0.021045933	0.993516667	0.026352141	0.991

44	Leaky ReLU	0.5	Adam	0.1	glorot_uniform	14.47109451	0.102183333	14.49016753	0.101
45	Leaky ReLU	0.5	Adam	0.1	RandomUniform	14.30695861	0.112366667	14.28869145	0.1135
46	Leaky ReLU	0.5	Adam	0.5	glorot_uniform	14.54872997	0.097366667	14.53529824	0.0982
47	Leaky ReLU	0.5	Adam	0.5	RandomUniform	14.51998605	0.09915	14.49177937	0.1009
48	Leaky ReLU	0.5	Adagrad	0.01	glorot_uniform	0.119199034	0.965316667	0.109431923	0.9644
49	Leaky ReLU	0.5	Adagrad	0.01	RandomUniform	0.199687242	0.93995	0.187957279	0.9426
50	Leaky ReLU	0.5	Adagrad	0.1	glorot_uniform	12.6346412	0.213716667	12.6432297	0.2126
51	Leaky ReLU	0.5	Adagrad	0.1	RandomUniform	14.66182524	0.09035	14.68036102	0.0892
52	Leaky ReLU	0.5	Adagrad	0.5	glorot_uniform	14.54872997	0.097366667	14.53529824	0.0982
53	Leaky ReLU	0.5	Adagrad	0.5	RandomUniform	14.659448	0.090466667	14.67874921	0.0893
54	Leaky ReLU	0.5	RMSprop	0.01	glorot_uniform	0.020891074	0.99375	0.026997171	0.9913
55	Leaky ReLU	0.5	RMSprop	0.01	RandomUniform	0.022677565	0.9929	0.028585653	0.9902
56	Leaky ReLU	0.5	RMSprop	0.1	glorot_uniform	14.5283137	0.098633333	14.5739165	0.0958
57	Leaky ReLU	0.5	RMSprop	0.1	RandomUniform	1.990191116	0.847283333	1.921220475	0.8563
58	Leaky ReLU	0.5	RMSprop	0.5	glorot_uniform	14.67471971	0.08955	14.69325549	0.0884
59	Leaky ReLU	0.5	RMSprop	0.5	RandomUniform	13.5738541	0.15785	13.5843385	0.1572
100	linear	0.2	SGD	0.01	glorot_uniform	0.107141199	0.967766667	0.100002509	0.9676
101	linear	0.2	SGD	0.01	RandomUniform	0.244069326	0.9279	0.233973935	0.9322
102	linear	0.2	SGD	0.1	glorot_uniform	0.085684157	0.974916667	0.07686539	0.9756
103	linear	0.2	SGD	0.1	RandomUniform	0.187122115	0.9456	0.175157424	0.9466
104	linear	0.2	SGD	0.5	glorot_uniform	0.087046067	0.974216667	0.07904486	0.9761
105	linear	0.2	SGD	0.5	RandomUniform	0.154455192	0.953783333	0.142453263	0.9558
106	linear	0.2	Adadelta	0.01	glorot_uniform	1.352772431	0.724183333	1.333008433	0.7435

107	linear	0.2	Adadelta	0.01	RandomUniform	2.302114596	0.112366667	2.302084087	0.1135
108	linear	0.2	Adadelta	0.1	glorot_uniform	0.932077346	0.7906	0.906141613	0.8011
109	linear	0.2	Adadelta	0.1	RandomUniform	2.301321253	0.112383333	2.301309648	0.1135
110	linear	0.2	Adadelta	0.5	glorot_uniform	0.795500358	0.815216667	0.763965332	0.8288
111	linear	0.2	Adadelta	0.5	RandomUniform	2.301523697	0.112366667	2.30148299	0.1135
112	linear	0.2	Adam	0.01	glorot_uniform	0.027019341	0.991883333	0.036261391	0.9883
113	linear	0.2	Adam	0.01	RandomUniform	0.028303074	0.991583333	0.034228271	0.9889
114	linear	0.2	Adam	0.1	glorot_uniform	14.30695861	0.112366667	14.28869145	0.1135
115	linear	0.2	Adam	0.1	RandomUniform	14.52697056	0.098716667	14.53852184	0.098
116	linear	0.2	Adam	0.5	glorot_uniform	14.43509743	0.104416667	14.46115503	0.1028
117	linear	0.2	Adam	0.5	RandomUniform	12.83725718	0.20355	12.84612215	0.203
118	linear	0.2	Adagrad	0.01	glorot_uniform	0.104530508	0.969283333	0.09871313	0.969
119	linear	0.2	Adagrad	0.01	RandomUniform	0.121289067	0.964083333	0.109681258	0.9664
120	linear	0.2	Adagrad	0.1	glorot_uniform	14.43509743	0.104416667	14.46115503	0.1028
121	linear	0.2	Adagrad	0.1	RandomUniform	14.51756834	0.0993	14.45470776	0.1032
122	linear	0.2	Adagrad	0.5	glorot_uniform	14.51756834	0.0993	14.45470776	0.1032
123	linear	0.2	Adagrad	0.5	RandomUniform	14.30695861	0.112366667	14.28869145	0.1135
124	linear	0.2	RMSprop	0.01	glorot_uniform	14.51998605	0.09915	14.49177937	0.1009
125	linear	0.2	RMSprop	0.01	RandomUniform	0.024988758	0.992666667	0.035597572	0.9876
126	linear	0.2	RMSprop	0.1	glorot_uniform	14.66182524	0.09035	14.68036102	0.0892
127	linear	0.2	RMSprop	0.1	RandomUniform	14.51998605	0.09915	14.49177937	0.1009
128	linear	0.2	RMSprop	0.5	glorot_uniform	14.51729971	0.099316667	14.45470776	0.1032
129	linear	0.2	RMSprop	0.5	RandomUniform	14.66182524	0.09035	14.68036102	0.0892

130	linear	0.5	SGD	0.01	glorot_uniform	0.140843052	0.958716667	0.128327456	0.9625
131	linear	0.5	SGD	0.01	RandomUniform	0.374399388	0.89205	0.356404312	0.8991
132	linear	0.5	SGD	0.1	glorot_uniform	0.137105136	0.959366667	0.123547868	0.9627
133	linear	0.5	SGD	0.1	RandomUniform	0.254932614	0.9257	0.241394283	0.9288
134	linear	0.5	SGD	0.5	glorot_uniform	14.30695861	0.112366667	14.28869145	0.1135
135	linear	0.5	SGD	0.5	RandomUniform	0.183198281	0.9472	0.171593314	0.9481
136	linear	0.5	Adadelta	0.01	glorot_uniform	1.749553343	0.597633333	1.735432279	0.6007
137	linear	0.5	Adadelta	0.01	RandomUniform	2.302168653	0.1124	2.302155042	0.1135
138	linear	0.5	Adadelta	0.1	glorot_uniform	1.414663051	0.763383333	1.397481738	0.7702
139	linear	0.5	Adadelta	0.1	RandomUniform	2.302133351	0.112366667	2.302120626	0.1135
140	linear	0.5	Adadelta	0.5	glorot_uniform	1.292535587	0.727883333	1.275021763	0.7419
141	linear	0.5	Adadelta	0.5	RandomUniform	2.301659082	0.112366667	2.301640929	0.1135
142	linear	0.5	Adam	0.01	glorot_uniform	0.069035814	0.979016667	0.062949222	0.9801
143	linear	0.5	Adam	0.01	RandomUniform	0.075800999	0.97745	0.068539694	0.9777
144	linear	0.5	Adam	0.1	glorot_uniform	14.54872997	0.097366667	14.53529824	0.0982
145	linear	0.5	Adam	0.1	RandomUniform	14.54631226	0.097516667	14.54819266	0.0974
146	linear	0.5	Adam	0.5	glorot_uniform	14.54631226	0.097516667	14.54819266	0.0974
147	linear	0.5	Adam	0.5	RandomUniform	14.47109451	0.102183333	14.49016753	0.101
148	linear	0.5	Adagrad	0.01	glorot_uniform	0.156375067	0.953666667	0.141673181	0.9565
149	linear	0.5	Adagrad	0.01	RandomUniform	0.180330274	0.947216667	0.16501004	0.9482
150	linear	0.5	Adagrad	0.1	glorot_uniform	14.66182524	0.09035	14.68036102	0.0892
151	linear	0.5	Adagrad	0.1	RandomUniform	14.30695861	0.112366667	14.28869145	0.1135
152	linear	0.5	Adagrad	0.5	glorot_uniform	14.30695861	0.112366667	14.28869145	0.1135

153	linear	0.5	Adagrad	0.5	RandomUniform	12.9447127	0.196783333	12.95572523	0.1962
154	linear	0.5	RMSprop	0.01	glorot_uniform	14.30695861	0.112366667	14.28869145	0.1135
155	linear	0.5	RMSprop	0.01	RandomUniform	0.072774177	0.9785	0.06795853	0.9777
156	linear	0.5	RMSprop	0.1	glorot_uniform	14.30695861	0.112366667	14.28869145	0.1135
157	linear	0.5	RMSprop	0.1	RandomUniform	14.51783698	0.099283333	14.45309595	0.1033
158	linear	0.5	RMSprop	0.5	glorot_uniform	14.47109451	0.102183333	14.49016753	0.101
159	linear	0.5	RMSprop	0.5	RandomUniform	14.43509743	0.104416667	14.46115503	0.1028
300	relu	0.2	SGD	0.01	glorot_uniform	0.090239444	0.972233333	0.08419138	0.9728
301	relu	0.2	SGD	0.01	RandomUniform	2.287950625	0.112383333	2.287439823	0.1135
302	relu	0.2	SGD	0.1	glorot_uniform	0.077784548	0.97635	0.071509414	0.977
303	relu	0.2	SGD	0.1	RandomUniform	0.358594448	0.89915	0.340638964	0.9059
304	relu	0.2	SGD	0.5	glorot_uniform	2.301159277	0.112366667	2.301021397	0.1135
305	relu	0.2	SGD	0.5	RandomUniform	0.211499602	0.93865	0.196982544	0.943
306	relu	0.2	Adadelta	0.01	glorot_uniform	2.056340389	0.50045	2.054264886	0.5027
307	relu	0.2	Adadelta	0.01	RandomUniform	2.302346898	0.112366667	2.302335107	0.1135
308	relu	0.2	Adadelta	0.1	glorot_uniform	1.938456338	0.6331	1.931747162	0.6521
309	relu	0.2	Adadelta	0.1	RandomUniform	2.302153585	0.112366667	2.302142062	0.1135
310	relu	0.2	Adadelta	0.5	glorot_uniform	1.941111758	0.48995	1.936883507	0.4873
311	relu	0.2	Adadelta	0.5	RandomUniform	2.302162798	0.112366667	2.302137801	0.1135
312	relu	0.2	Adam	0.01	glorot_uniform	0.003741685	0.999183333	0.021457492	0.9932
313	relu	0.2	Adam	0.01	RandomUniform	0.006532256	0.99845	0.019687754	0.9931
314	relu	0.2	Adam	0.1	glorot_uniform	2.30115928	0.112366667	2.301025502	0.1135
315	relu	0.2	Adam	0.1	RandomUniform	0.051319862	0.985983333	0.062059572	0.9811

316	relu	0.2	Adam	0.5	glorot_uniform	14.51756834	0.0993	14.45470776	0.1032
317	relu	0.2	Adam	0.5	RandomUniform	14.52697056	0.098716667	14.53852184	0.098
318	relu	0.2	Adagrad	0.01	glorot_uniform	0.079533717	0.976283333	0.068110429	0.9787
319	relu	0.2	Adagrad	0.01	RandomUniform	0.13478129	0.96035	0.127174629	0.961
320	relu	0.2	Adagrad	0.1	glorot_uniform	14.47109451	0.102183333	14.49016753	0.101
321	relu	0.2	Adagrad	0.1	RandomUniform	2.2082169	0.18825	2.204312809	0.1939
322	relu	0.2	Adagrad	0.5	glorot_uniform	14.52697056	0.098716667	14.53852184	0.098
323	relu	0.2	Adagrad	0.5	RandomUniform	14.54631226	0.097516667	14.54819266	0.0974
324	relu	0.2	RMSprop	0.01	glorot_uniform	0.003930898	0.99905	0.022094381	0.9922
325	relu	0.2	RMSprop	0.01	RandomUniform	0.008637019	0.997666667	0.020744773	0.9934
326	relu	0.2	RMSprop	0.1	glorot_uniform	14.52697056	0.098716667	14.53852184	0.098
327	relu	0.2	RMSprop	0.1	RandomUniform	14.51998605	0.09915	14.49177937	0.1009
328	relu	0.2	RMSprop	0.5	glorot_uniform	14.66182524	0.09035	14.68036102	0.0892
329	relu	0.2	RMSprop	0.5	RandomUniform	14.51998605	0.09915	14.49177937	0.1009
330	relu	0.5	SGD	0.01	glorot_uniform	0.148260248	0.959266667	0.136376337	0.9603
331	relu	0.5	SGD	0.01	RandomUniform	2.296252349	0.112366667	2.295992073	0.1135
332	relu	0.5	SGD	0.1	glorot_uniform	0.131137471	0.96045	0.120844972	0.9634
333	relu	0.5	SGD	0.1	RandomUniform	0.616361144	0.830783333	0.599156701	0.8373
334	relu	0.5	SGD	0.5	glorot_uniform	0.792135699	0.8138	0.769604556	0.8265
335	relu	0.5	SGD	0.5	RandomUniform	0.229158248	0.9378	0.218036325	0.9412
336	relu	0.5	Adadelta	0.01	glorot_uniform	2.247296929	0.358433333	2.247378036	0.355
337	relu	0.5	Adadelta	0.01	RandomUniform	2.30235811	0.112366667	2.302352671	0.1135
338	relu	0.5	Adadelta	0.1	glorot_uniform	2.164656144	0.447216667	2.163627447	0.4429

339	relu	0.5	Adadelta	0.1	RandomUniform	2.302265622	0.112366667	2.302258164	0.1135
340	relu	0.5	Adadelta	0.5	glorot_uniform	2.118480429	0.5792	2.114676093	0.5934
341	relu	0.5	Adadelta	0.5	RandomUniform	2.301847136	0.1177	2.301847843	0.1176
342	relu	0.5	Adam	0.01	glorot_uniform	0.025488105	0.992833333	0.029801478	0.9906
343	relu	0.5	Adam	0.01	RandomUniform	0.034433623	0.989783333	0.037612062	0.9871
344	relu	0.5	Adam	0.1	glorot_uniform	2.301159317	0.112366667	2.301016188	0.1135
345	relu	0.5	Adam	0.1	RandomUniform	2.301159401	0.112366667	2.301016633	0.1135
346	relu	0.5	Adam	0.5	glorot_uniform	14.54872997	0.097366667	14.53529824	0.0982
347	relu	0.5	Adam	0.5	RandomUniform	14.47109451	0.102183333	14.49016753	0.101
348	relu	0.5	Adagrad	0.01	glorot_uniform	0.112393773	0.968533333	0.103902296	0.9702
349	relu	0.5	Adagrad	0.01	RandomUniform	0.190326486	0.945933333	0.180617001	0.9476
350	relu	0.5	Adagrad	0.1	glorot_uniform	2.301160505	0.112366667	2.301035883	0.1135
351	relu	0.5	Adagrad	0.1	RandomUniform	0.295159954	0.92715	0.28086699	0.9318
352	relu	0.5	Adagrad	0.5	glorot_uniform	14.51756834	0.0993	14.45470776	0.1032
353	relu	0.5	Adagrad	0.5	RandomUniform	14.47109451	0.102183333	14.49016753	0.101
354	relu	0.5	RMSprop	0.01	glorot_uniform	0.023194224	0.9934	0.027684475	0.9919
355	relu	0.5	RMSprop	0.01	RandomUniform	0.036540137	0.989116667	0.040841876	0.9872
356	relu	0.5	RMSprop	0.1	glorot_uniform	14.52697056	0.098716667	14.53852184	0.098
357	relu	0.5	RMSprop	0.1	RandomUniform	14.52697056	0.098716667	14.53852184	0.098
358	relu	0.5	RMSprop	0.5	glorot_uniform	14.5283137	0.098633333	14.57398165	0.0958
359	relu	0.5	RMSprop	0.5	RandomUniform	14.51998605	0.09915	14.49177937	0.1009
400	tanh	0.2	SGD	0.01	glorot_uniform	0.130168535	0.961666667	0.120086782	0.9638
401	tanh	0.2	SGD	0.01	RandomUniform	0.376593573	0.8955	0.358040967	0.8995

402	tanh	0.2	SGD	0.1	glorot_uniform	0.107781914	0.968483333	0.101250448	0.9674
403	tanh	0.2	SGD	0.1	RandomUniform	0.244029196	0.931266667	0.227794141	0.9364
404	tanh	0.2	SGD	0.5	glorot_uniform	0.110320259	0.966566667	0.104012988	0.9666
405	tanh	0.2	SGD	0.5	RandomUniform	0.195228667	0.943566667	0.181384527	0.9469
406	tanh	0.2	Adadelta	0.01	glorot_uniform	1.426103692	0.701416667	1.407023287	0.7219
407	tanh	0.2	Adadelta	0.01	RandomUniform	2.302071558	0.138933333	2.302058526	0.1416
408	tanh	0.2	Adadelta	0.1	glorot_uniform	1.319528354	0.759083333	1.301983503	0.7688
409	tanh	0.2	Adadelta	0.1	RandomUniform	2.301653067	0.112366667	2.301620459	0.1135
410	tanh	0.2	Adadelta	0.5	glorot_uniform	1.131456195	0.764366667	1.113415995	0.7735
411	tanh	0.2	Adadelta	0.5	RandomUniform	2.301820925	0.112366667	2.301772076	0.1135
412	tanh	0.2	Adam	0.01	glorot_uniform	0.001865573	0.999716667	0.028873648	0.9913
413	tanh	0.2	Adam	0.01	RandomUniform	0.001738894	0.999766667	0.027259979	0.9906
414	tanh	0.2	Adam	0.1	glorot_uniform	0.271599115	0.9461	0.263864059	0.9472
415	tanh	0.2	Adam	0.1	RandomUniform	1.697135026	0.312416667	1.692399977	0.3122
416	tanh	0.2	Adam	0.5	glorot_uniform	7.673013232	0.104416667	7.643724861	0.1028
417	tanh	0.2	Adam	0.5	RandomUniform	8.486098086	0.344783333	8.456739764	0.3473
418	tanh	0.2	Adagrad	0.01	glorot_uniform	0.086899181	0.974283333	0.07944709	0.975
419	tanh	0.2	Adagrad	0.01	RandomUniform	0.103533891	0.9692	0.098539216	0.9697
420	tanh	0.2	Adagrad	0.1	glorot_uniform	0.253675265	0.928466667	0.247598986	0.93
421	tanh	0.2	Adagrad	0.1	RandomUniform	2.058179 4	0.28415	2.057977143	0.2881
422	tanh	0.2	Adagrad	0.5	glorot_uniform	11.68383392	0.09915	11.65579602	0.1009
423	tanh	0.2	Adagrad	0.5	RandomUniform	7.341461263	0.112366667	7.365277678	0.1135
424	tanh	0.2	RMSprop	0.01	glorot_uniform	0.001350198	0.99975	0.029994731	0.9909

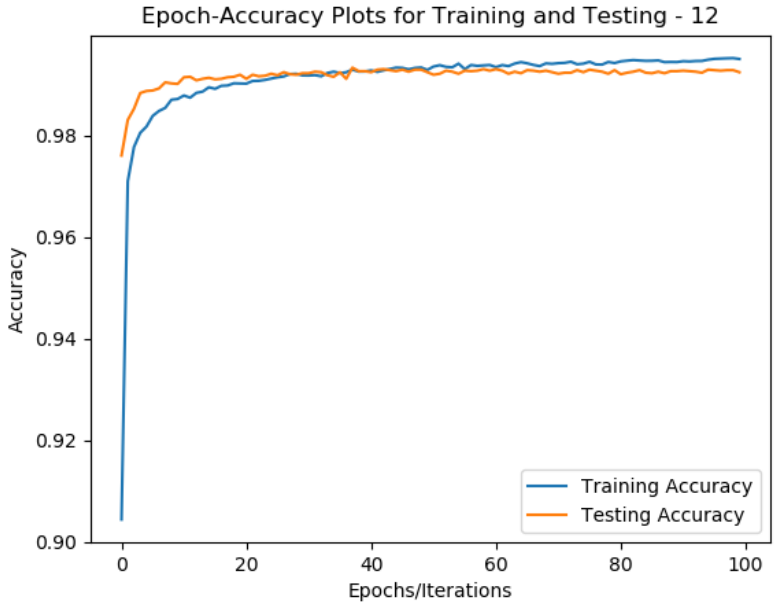
425	tanh	0.2	RMS prop	0.01	RandomUniform	0.001830033	0.999716667	0.028636257	0.9905
426	tanh	0.2	RMS prop	0.1	glorot_uniform	11.11983756	0.309416667	11.08657691	0.311
427	tanh	0.2	RMS prop	0.1	RandomUniform	1.981538015	0.840866667	1.998650567	0.8385
428	tanh	0.2	RMS prop	0.5	glorot_uniform	14.52697056	0.098716667	14.53852184	0.098
429	tanh	0.2	RMS prop	0.5	RandomUniform	14.52697056	0.098716667	14.53852184	0.098
430	tanh	0.5	SGD	0.01	glorot_uniform	0.159863721	0.951933333	0.147384556	0.9543
431	tanh	0.5	SGD	0.01	RandomUniform	0.483973139	0.86405	0.465458405	0.8713
432	tanh	0.5	SGD	0.1	glorot_uniform	0.170687714	0.948033333	0.15929122	0.9513
433	tanh	0.5	SGD	0.1	RandomUniform	0.281614246	0.91945	0.264154159	0.9242
434	tanh	0.5	SGD	0.5	glorot_uniform	0.148733075	0.95525	0.139405324	0.9563
435	tanh	0.5	SGD	0.5	RandomUniform	0.212541859	0.93915	0.195983104	0.942
436	tanh	0.5	Adadelta	0.01	glorot_uniform	1.9342463	0.584716667	1.929761799	0.5913
437	tanh	0.5	Adadelta	0.01	RandomUniform	2.302084998	0.113216667	2.302068808	0.1144
438	tanh	0.5	Adadelta	0.1	glorot_uniform	1.730499776	0.585583333	1.713906847	0.6024
439	tanh	0.5	Adadelta	0.1	RandomUniform	2.30158508	0.126716667	2.301559427	0.1274
440	tanh	0.5	Adadelta	0.5	glorot_uniform	1.568269483	0.703366667	1.552569995	0.7242
441	tanh	0.5	Adadelta	0.5	RandomUniform	2.301914838	0.112366667	2.301904066	0.1135
442	tanh	0.5	Adam	0.01	glorot_uniform	0.019042892	0.99425	0.031595931	0.9884
443	tanh	0.5	Adam	0.01	RandomUniform	0.018467066	0.99445	0.031416636	0.9891
444	tanh	0.5	Adam	0.1	glorot_uniform	1.744748629	0.335083333	1.728599952	0.3381
445	tanh	0.5	Adam	0.1	RandomUniform	1.841664925	0.356733333	1.839964135	0.3606
446	tanh	0.5	Adam	0.5	glorot_uniform	5.801034124	0.31225	5.741153795	0.3139
447	tanh	0.5	Adam	0.5	RandomUniform	6.142427787	0.17945	6.16281461	0.1778

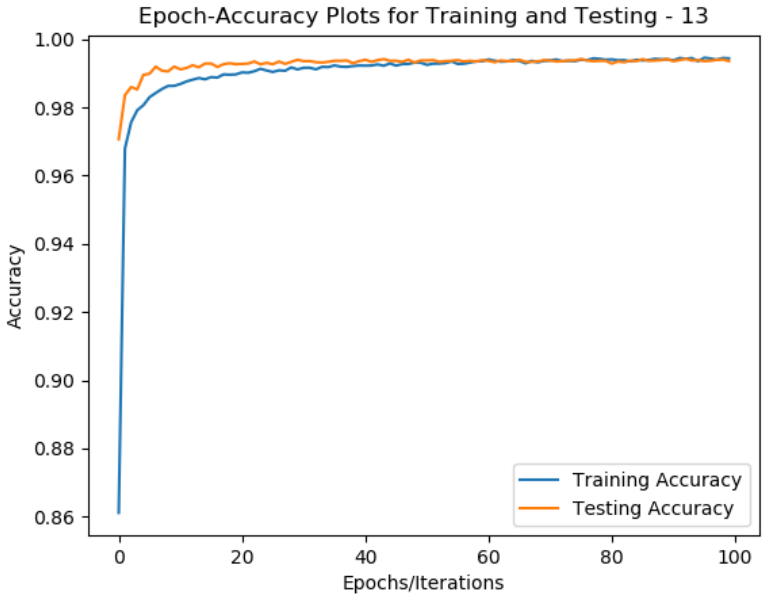
448	tanh	0.5	Adagrad	0.01	glorot_uniform	0.125013848	0.962816667	0.118112929	0.963
449	tanh	0.5	Adagrad	0.01	RandomUniform	0.145059302	0.95665	0.137155773	0.9581
450	tanh	0.5	Adagrad	0.1	glorot_uniform	0.839193935	0.79425	0.832899372	0.7933
451	tanh	0.5	Adagrad	0.1	RandomUniform	0.907739632	0.76705	0.903720916	0.7604
452	tanh	0.5	Adagrad	0.5	glorot_uniform	9.006154527	0.104416667	8.987987965	0.1028
453	tanh	0.5	Adagrad	0.5	RandomUniform	7.047359104	0.327366667	6.978148764	0.3298
454	tanh	0.5	RMSprop	0.01	glorot_uniform	0.018427967	0.9946	0.031867354	0.9902
455	tanh	0.5	RMSprop	0.01	RandomUniform	0.017848976	0.994616667	0.029414366	0.9898
456	tanh	0.5	RMSprop	0.1	glorot_uniform	0.156639859	0.953716667	0.14725676	0.9566
457	tanh	0.5	RMSprop	0.1	RandomUniform	0.246972851	0.927083333	0.250428533	0.9258
458	tanh	0.5	RMSprop	0.5	glorot_uniform	14.51756834	0.0993	14.45470776	0.1032
459	tanh	0.5	RMSprop	0.5	RandomUniform	14.51756834	0.0993	14.45470776	0.1032
500	sigmoid	0.2	SGD	0.01	glorot_uniform	2.301114494	0.112366667	2.300978508	0.1135
501	sigmoid	0.2	SGD	0.01	RandomUniform	2.301159283	0.112366667	2.301021317	0.1135
502	sigmoid	0.2	SGD	0.1	glorot_uniform	2.30064307	0.112366667	2.300497216	0.1135
503	sigmoid	0.2	SGD	0.1	RandomUniform	2.301159036	0.112366667	2.301016498	0.1135
504	sigmoid	0.2	SGD	0.5	glorot_uniform	2.300999998	0.112366667	2.300844822	0.1135
505	sigmoid	0.2	SGD	0.5	RandomUniform	2.301159595	0.112366667	2.301023611	0.1135
506	sigmoid	0.2	Adadelta	0.01	glorot_uniform	2.303031746	0.102183333	2.302590202	0.101
507	sigmoid	0.2	Adadelta	0.01	RandomUniform	2.30145738	0.112366667	2.301286048	0.1135
508	sigmoid	0.2	Adadelta	0.1	glorot_uniform	2.301385317	0.112366667	2.301219294	0.1135
509	sigmoid	0.2	Adadelta	0.1	RandomUniform	2.301401877	0.112366667	2.301229113	0.1135
510	sigmoid	0.2	Adadelta	0.5	glorot_uniform	2.304432721	0.0993	2.303787956	0.1032

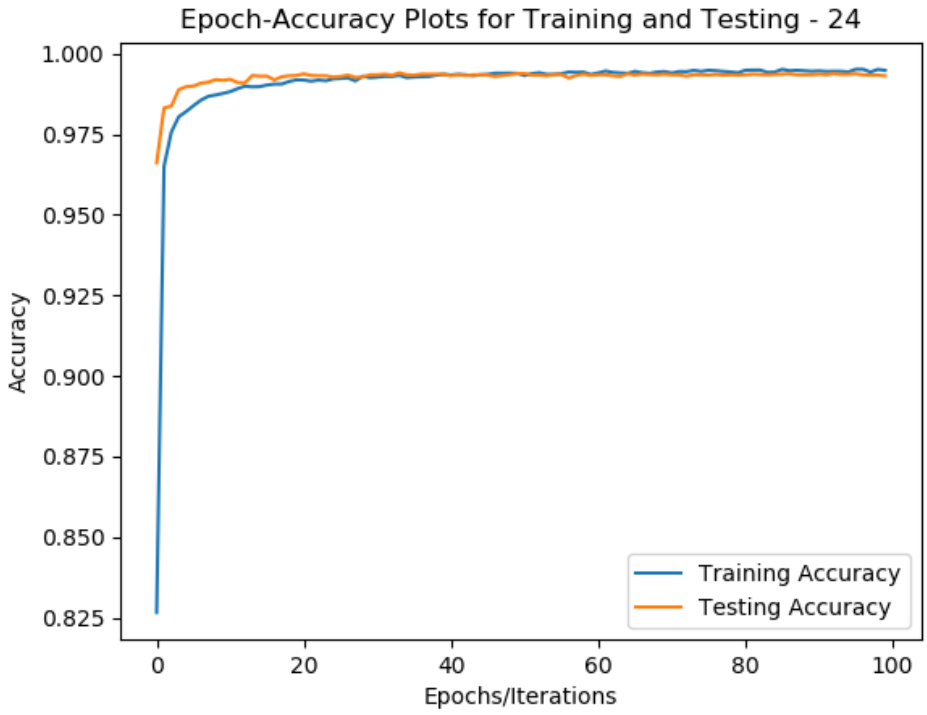
511	sigmoid	0.2	Adadelta	0.5	RandomUniform	2.301206636	0.112366667	2.301168602	0.1135
512	sigmoid	0.2	Adam	0.01	glorot_uniform	0.024876838	0.99245	0.032249454	0.989
513	sigmoid	0.2	Adam	0.01	RandomUniform	0.070064778	0.978333333	0.066788415	0.979
514	sigmoid	0.2	Adam	0.1	glorot_uniform	2.301162339	0.112366667	2.301037695	0.1135
515	sigmoid	0.2	Adam	0.1	RandomUniform	2.30115928	0.112366667	2.301019498	0.1135
516	sigmoid	0.2	Adam	0.5	glorot_uniform	7.519105436	0.112366667	7.509372476	0.1135
517	sigmoid	0.2	Adam	0.5	RandomUniform	10.01020438	0.112366667	10.03312419	0.1135
518	sigmoid	0.2	Adagrad	0.01	glorot_uniform	0.624418852	0.841433333	0.608142038	0.8508
519	sigmoid	0.2	Adagrad	0.01	RandomUniform	0.977321755	0.7206	0.970942721	0.7271
520	sigmoid	0.2	Adagrad	0.1	glorot_uniform	2.301159174	0.112366667	2.301019002	0.1135
521	sigmoid	0.2	Adagrad	0.1	RandomUniform	2.301159201	0.112366667	2.301018766	0.1135
522	sigmoid	0.2	Adagrad	0.5	glorot_uniform	10.13793555	0.112366667	10.15225623	0.1135
523	sigmoid	0.2	Adagrad	0.5	RandomUniform	7.317171433	0.112366667	7.313805725	0.1135
524	sigmoid	0.2	RMSprop	0.01	glorot_uniform	0.027429652	0.991783333	0.034662462	0.9889
525	sigmoid	0.2	RMSprop	0.01	RandomUniform	0.045795473	0.985583333	0.048342751	0.9842
526	sigmoid	0.2	RMSprop	0.1	glorot_uniform	10.31354939	0.098633333	10.34183413	0.0958
527	sigmoid	0.2	RMSprop	0.1	RandomUniform	8.645947203	0.112366667	8.69897358	0.1135
528	sigmoid	0.2	RMSprop	0.5	glorot_uniform	14.66182524	0.09035	14.68036102	0.0892
529	sigmoid	0.2	RMSprop	0.5	RandomUniform	14.51998605	0.09915	14.49177937	0.1009
530	sigmoid	0.5	SGD	0.01	glorot_uniform	2.300907399	0.112366667	2.300768943	0.1135
531	sigmoid	0.5	SGD	0.01	RandomUniform	2.301159	0.112366667	2.301018966	0.1135
532	sigmoid	0.5	SGD	0.1	glorot_uniform	2.300457263	0.112366667	2.300285709	0.1135
533	sigmoid	0.5	SGD	0.1	RandomUniform	2.301159196	0.112366667	2.301019705	0.1135

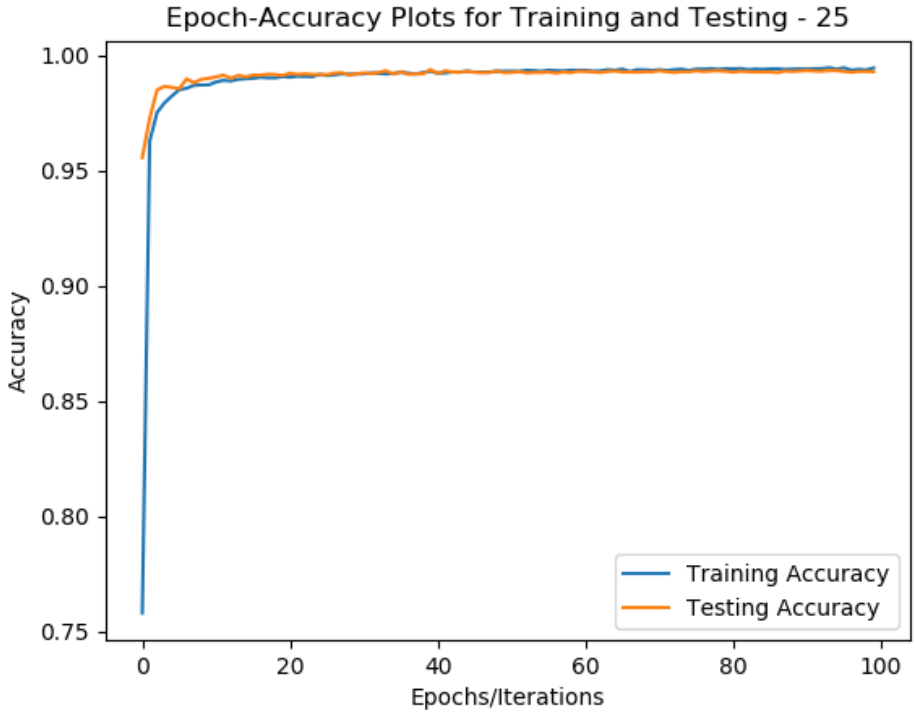
534	sigmoid	0.5	SGD	0.5	glorot_uniform	2.301033226	0.112366667	2.300886987	0.1135
535	sigmoid	0.5	SGD	0.5	RandomUniform	2.301158539	0.112366667	2.301016982	0.1135
536	sigmoid	0.5	Adadelta	0.01	glorot_uniform	2.304171795	0.112366667	2.303697122	0.1135
537	sigmoid	0.5	Adadelta	0.01	RandomUniform	2.301519164	0.112366667	2.30132785	0.1135
538	sigmoid	0.5	Adadelta	0.1	glorot_uniform	2.301916848	0.112366667	2.302237468	0.1135
539	sigmoid	0.5	Adadelta	0.1	RandomUniform	2.301220162	0.112366667	2.301040484	0.1135
540	sigmoid	0.5	Adadelta	0.5	glorot_uniform	2.301052254	0.112366667	2.300955186	0.1135
541	sigmoid	0.5	Adadelta	0.5	RandomUniform	2.301222616	0.112366667	2.301056211	0.1135
542	sigmoid	0.5	Adam	0.01	glorot_uniform	0.062300799	0.9801	0.062897673	0.9788
543	sigmoid	0.5	Adam	0.01	RandomUniform	0.070262664	0.977566667	0.069881996	0.977
544	sigmoid	0.5	Adam	0.1	glorot_uniform	2.301162441	0.112366667	2.301000632	0.1135
545	sigmoid	0.5	Adam	0.1	RandomUniform	2.301159376	0.112366667	2.301023241	0.1135
546	sigmoid	0.5	Adam	0.5	glorot_uniform	8.890092299	0.112366667	8.876787029	0.1135
547	sigmoid	0.5	Adam	0.5	RandomUniform	8.897645321	0.112366667	8.950790775	0.1135
548	sigmoid	0.5	Adagrad	0.01	glorot_uniform	0.843589346	0.78075	0.826165966	0.7904
549	sigmoid	0.5	Adagrad	0.01	RandomUniform	1.015351321	0.7034	1.00807056	0.7092
550	sigmoid	0.5	Adagrad	0.1	glorot_uniform	2.301159214	0.112366667	2.301019527	0.1135
551	sigmoid	0.5	Adagrad	0.1	RandomUniform	2.30115926	0.112366667	2.301012165	0.1135
552	sigmoid	0.5	Adagrad	0.5	glorot_uniform	10.05277523	0.112366667	10.04930186	0.1135
553	sigmoid	0.5	Adagrad	0.5	RandomUniform	10.09188884	0.112366667	10.06991371	0.1135
554	sigmoid	0.5	RMSprop	0.01	glorot_uniform	0.053367954	0.982716667	0.052046805	0.9825
555	sigmoid	0.5	RMSprop	0.01	RandomUniform	0.080810653	0.974116667	0.078419216	0.9739
556	sigmoid	0.5	RMSprop	0.1	glorot_uniform	8.544036606	0.2074	8.543151575	0.2127

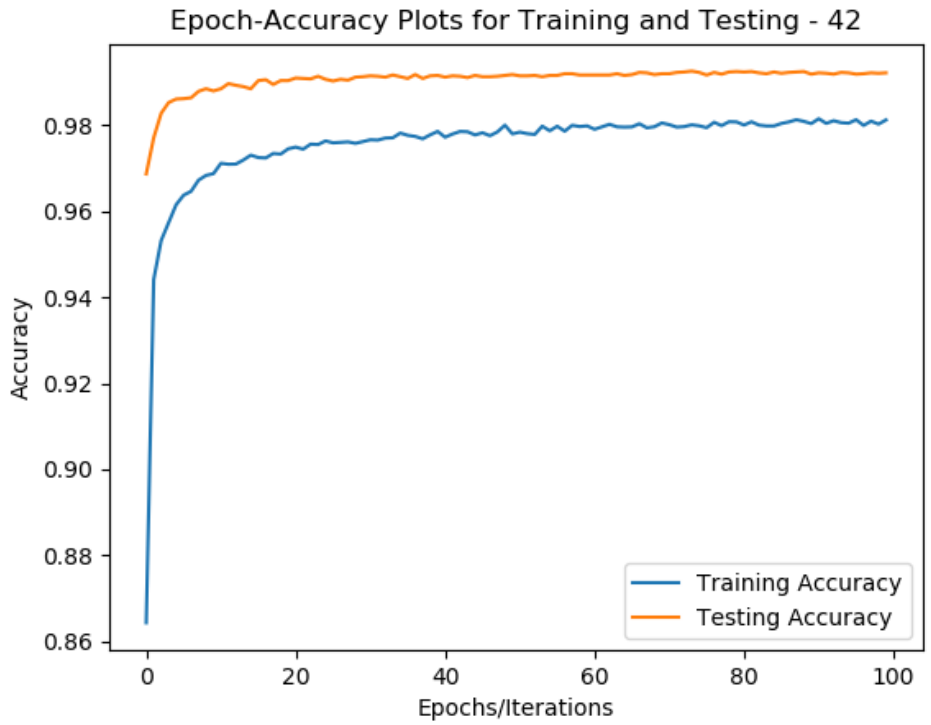
557	sigmoid	0.5	RMSprop	0.1	RandomUniform	8.700626477	0.112366667	8.696150442	0.1135
558	sigmoid	0.5	RMSprop	0.5	glorot_uniform	11.61923643	0.102183333	11.69028396	0.101
559	sigmoid	0.5	RMSprop	0.5	RandomUniform	14.30695861	0.112366667	14.28869145	0.1135

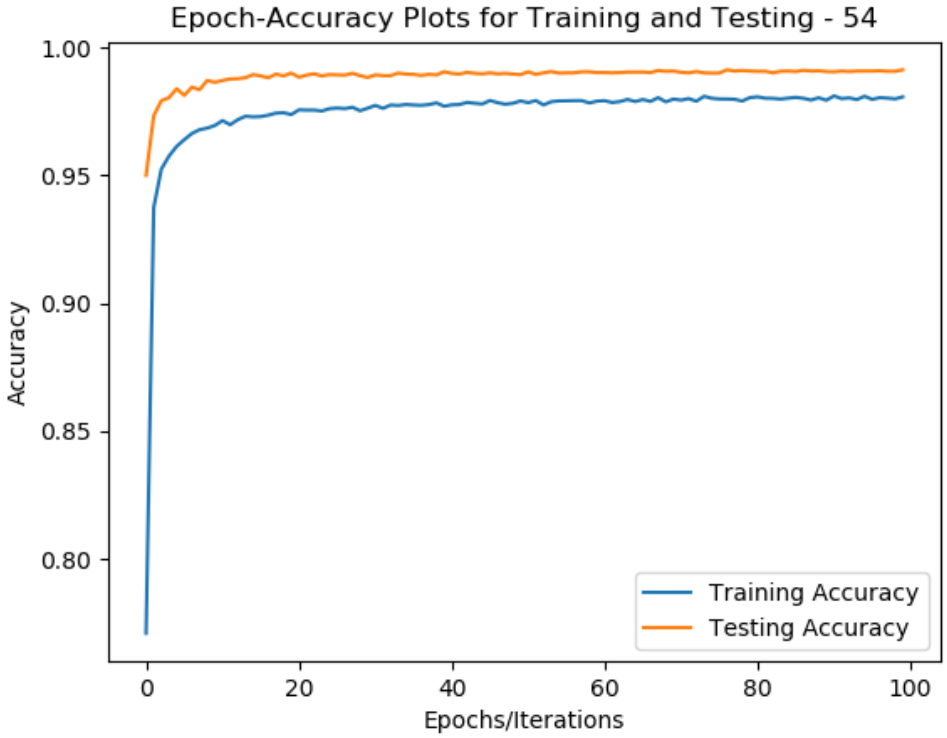


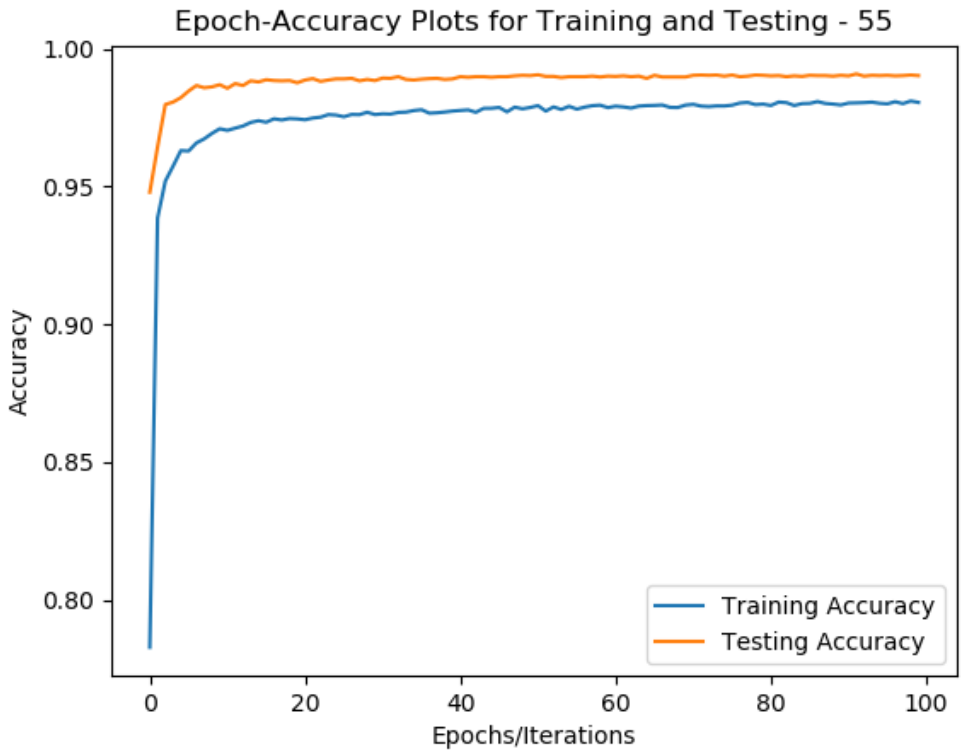


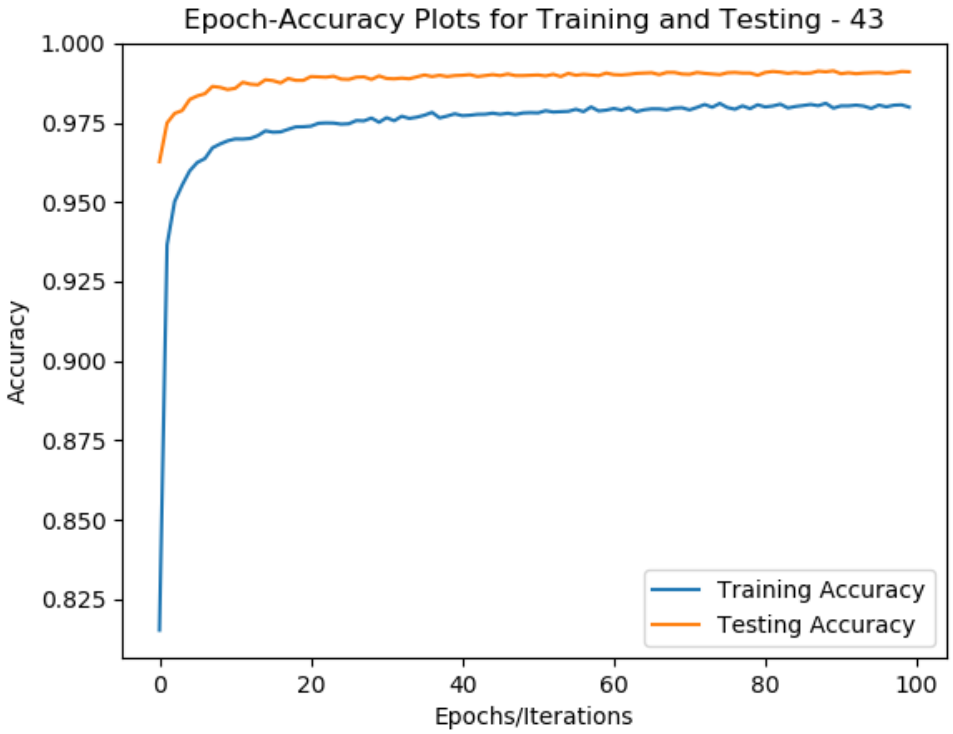

Epoch-Accuracy Plots for Training and Testing - 24
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
Accuracy
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy

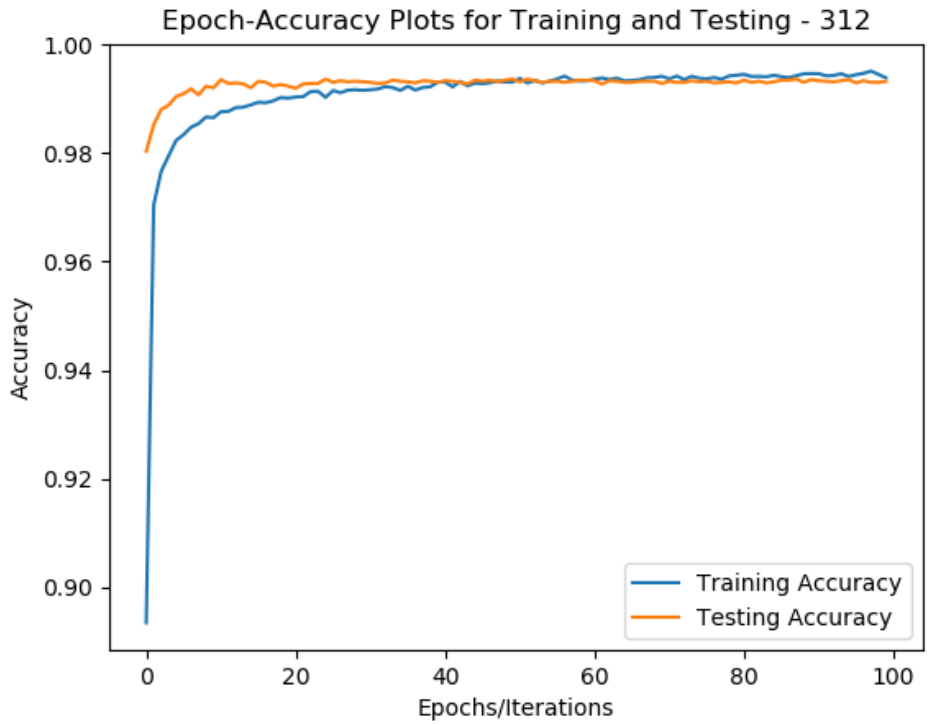

Epoch-Accuracy Plots for Training and Testing - 25
1.00
0.95
0.90
0.85
0.80
0.75
Accuracy
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy

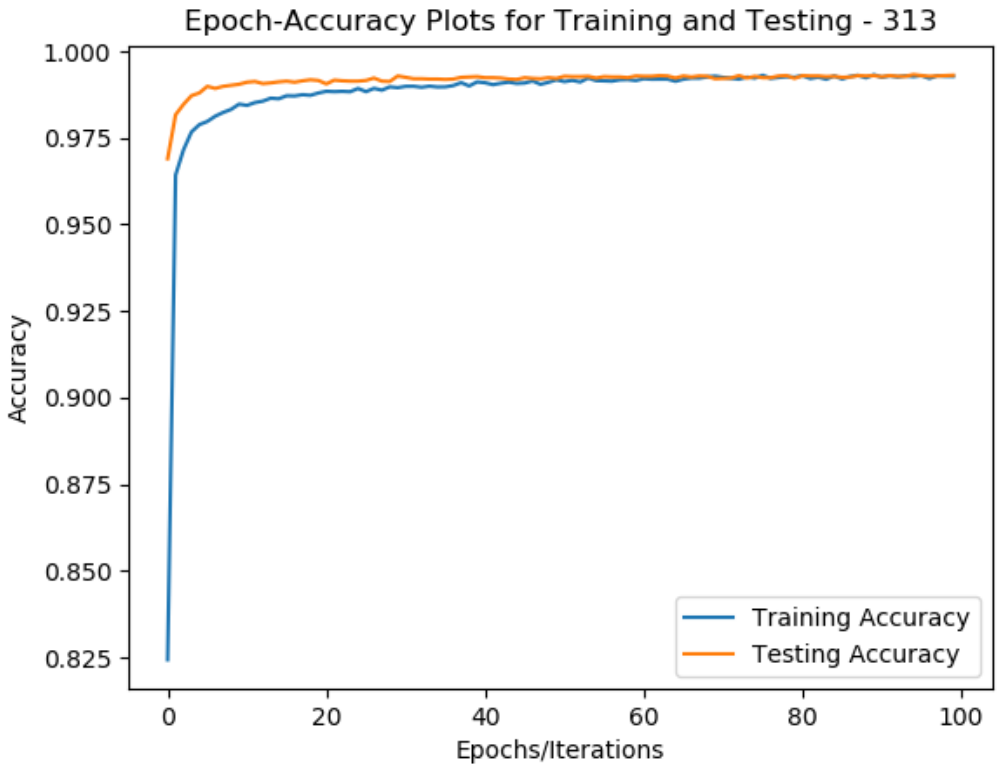

Epoch-Accuracy Plots for Training and Testing - 42
Accuracy
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy

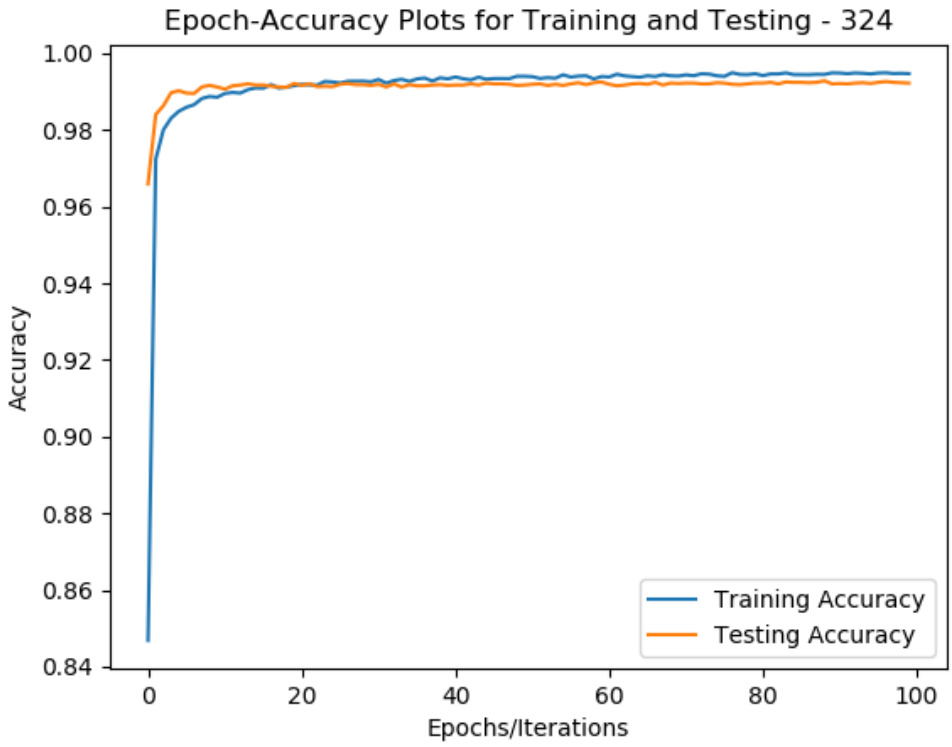

Epoch-Accuracy Plots for Training and Testing - 54
Accuracy
1.00
0.95
0.90
0.85
0.80
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy

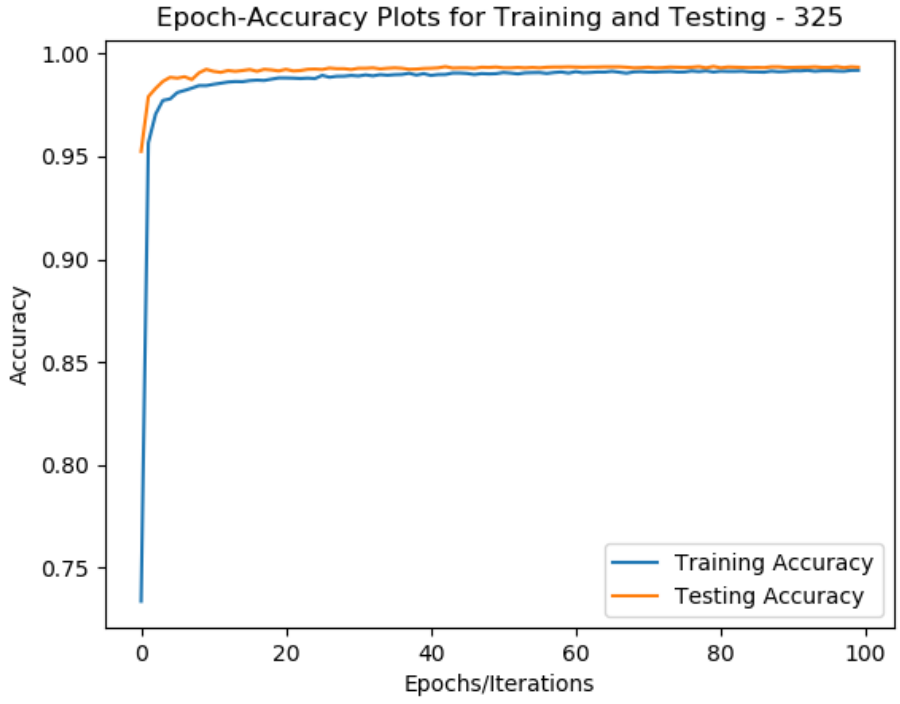

Epoch-Accuracy Plots for Training and Testing - 55
1.00
0.95
0.90
0.85
0.80
Accuracy
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy

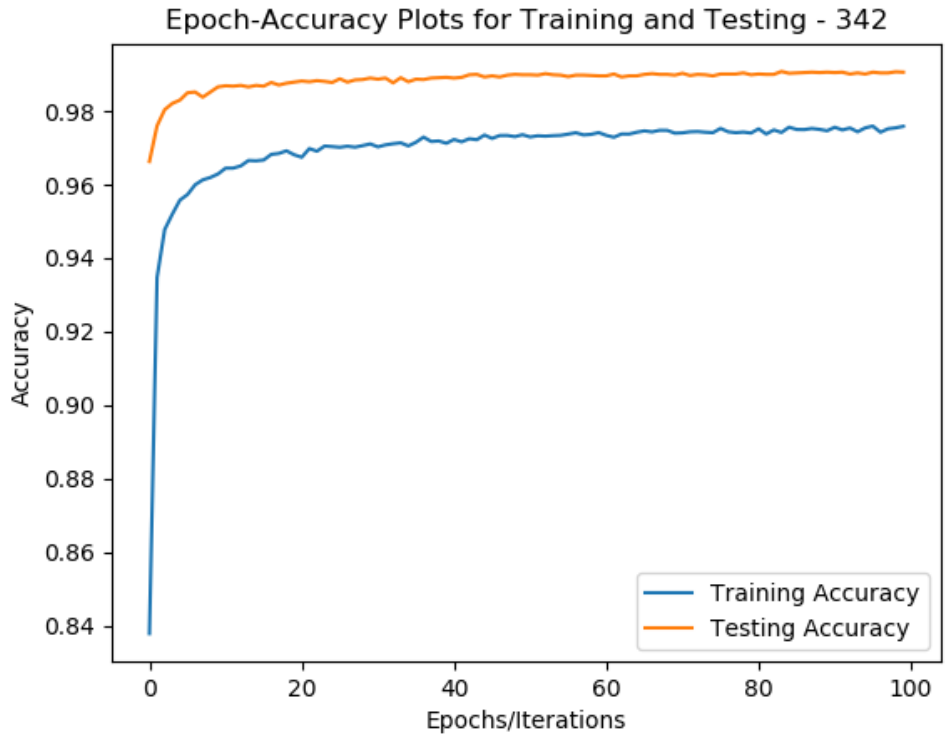

Epoch-Accuracy Plots for Training and Testing - 43
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
Accuracy
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy

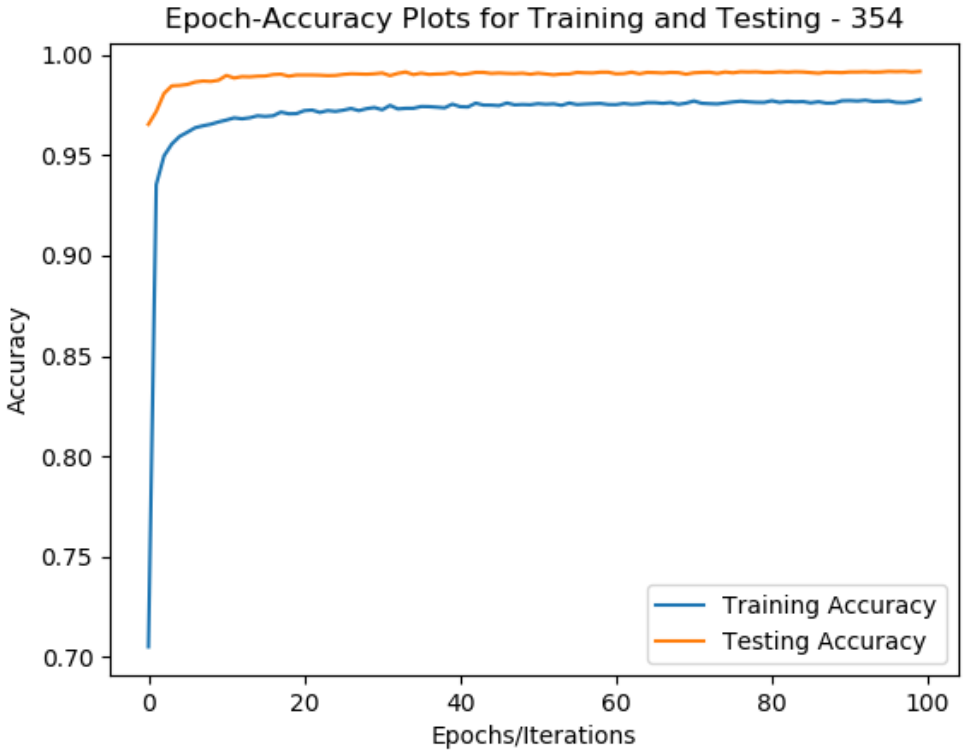

Epoch-Accuracy Plots for Training and Testing - 312
1.00
0.98
0.96
0.94
0.92
0.90
Accuracy
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy

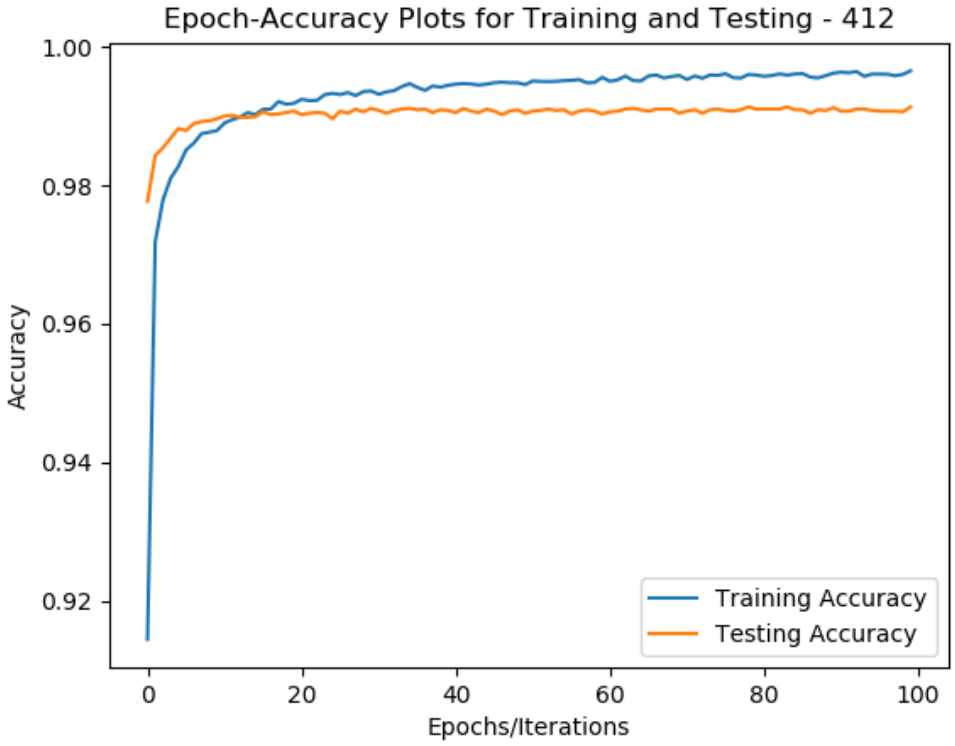

Epoch-Accuracy Plots for Training and Testing - 313
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
Accuracy
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy

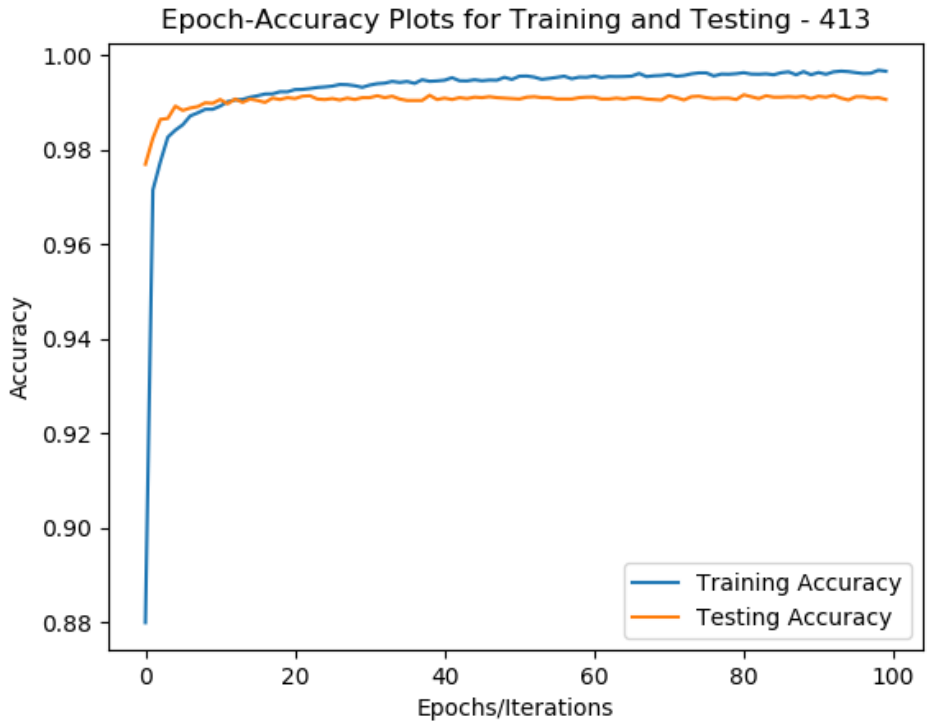

Epoch-Accuracy Plots for Training and Testing - 324
Accuracy
Epochs/Iterations
Training Accuracy
Testing Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0
20
40
60
80
100

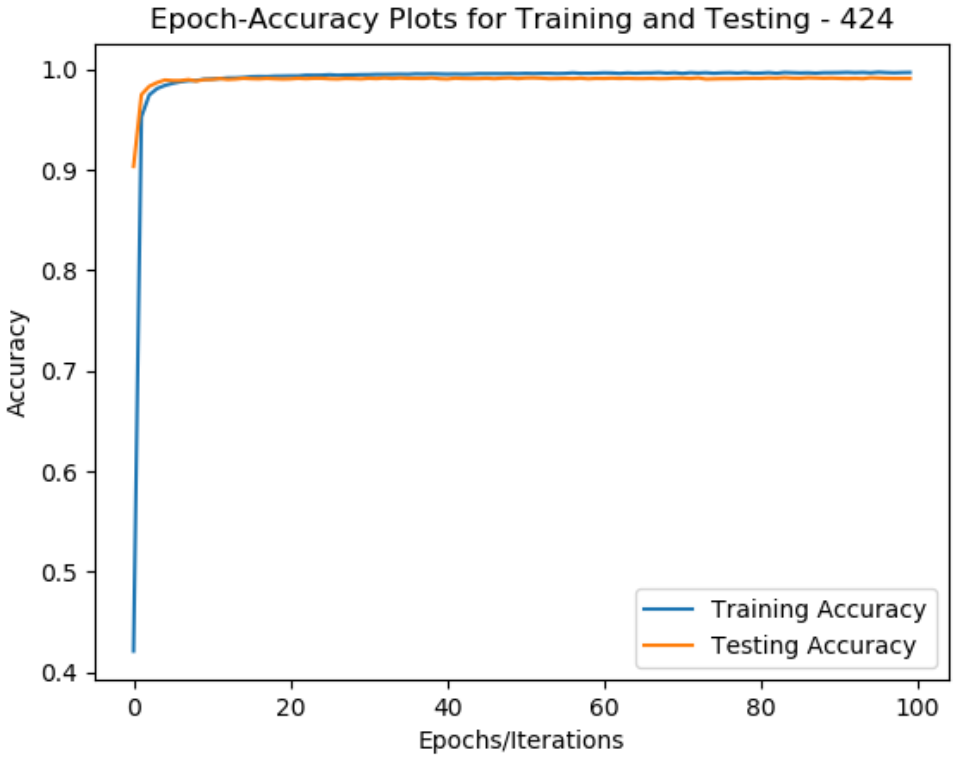

Epoch-Accuracy Plots for Training and Testing - 325
Accuracy
Epochs/Iterations
Training Accuracy
Testing Accuracy
1.00
0.95
0.90
0.85
0.80
0.75
0
20
40
60
80
100

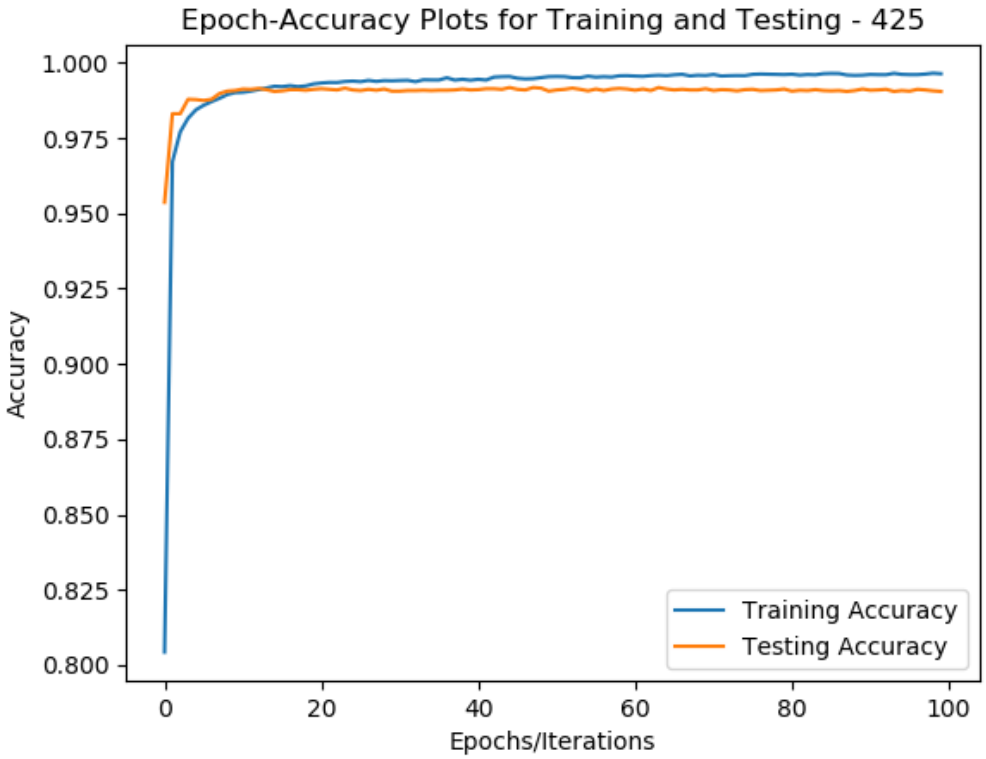

Epoch-Accuracy Plots for Training and Testing - 342
Accuracy
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy


Epoch-Accuracy Plots for Training and Testing - 354
Accuracy
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy


Epoch-Accuracy Plots for Training and Testing - 412
1.00
0.98
0.96
0.94
0.92
Accuracy
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy


Epoch-Accuracy Plots for Training and Testing - 413
1.00
0.98
0.96
0.94
0.92
0.90
0.88
Accuracy
0
20
40
60
80
100
Epochs/Iterations
Training Accuracy
Testing Accuracy


Epoch-Accuracy Plots for Training and Testing - 424
Accuracy
Epochs/Iterations
Training Accuracy
Testing Accuracy
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
20
40
60
80
100


Epoch-Accuracy Plots for Training and Testing - 425
Accuracy
Epochs/Iterations
Training Accuracy
Testing Accuracy
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0
20
40
60
80
100

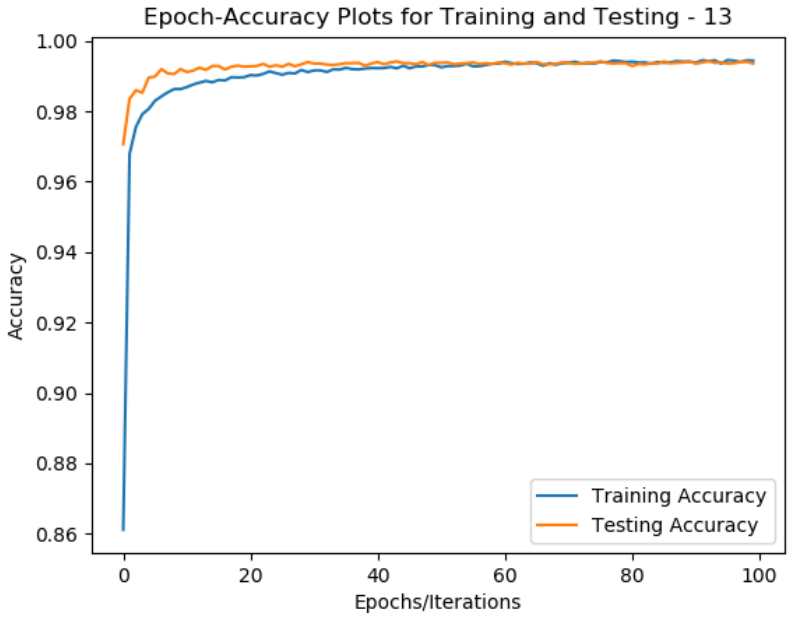
➢ DISCUSSION:

Thus, I have given the table of different parameters passed to run CNN and the Accuracy scores for Training and Testing data. I had also provided plots for the runs where the testing accuracy was more than 99%. Out of these experiments, some conclusions could be made on the LeNet-5 Architecture for MNIST Classification. They are listed below:

- **Leaky Relu, Relu and Tanh** activation functions work best for MNIST handwritten digits recognition. These are non-linear activation functions and perform better than Sigmoid or Linear functions
- Different dropout values work better in different cases. No conclusion could be made from the dropout values while looking into the performances
- While considering different Optimizers, **RMSProp and Adam** optimizers work the best. None of the other optimizers gave accuracy greater than 99%
- Learning Rate and Decay values work the best when they are set to lower values. The best performance is captured only when the learning rate is **as low as 0.01**
- Weight initialization isn't of much importance since both **Glorot Uniform** and **Random Uniform** weight initialization gives more than 99% accuracy
-

Best Performing parameters and outputs is given below: (Iteration number – 13)

- Activation Function: Leaky Relu
- Dropout value: 0.2
- Optimizer: Adam
- Learning Rate and Decay: 0.01
- Weight Initialization: Random Uniform
- Training Loss and Accuracy: 0.491% and 99.89%
- Testing Loss and Accuracy: 1.97% and 99.36%



C) IMPROVE THE LENET-5 FOR MNIST DATABASE

➢ ABSTRACT AND MOTIVATION:

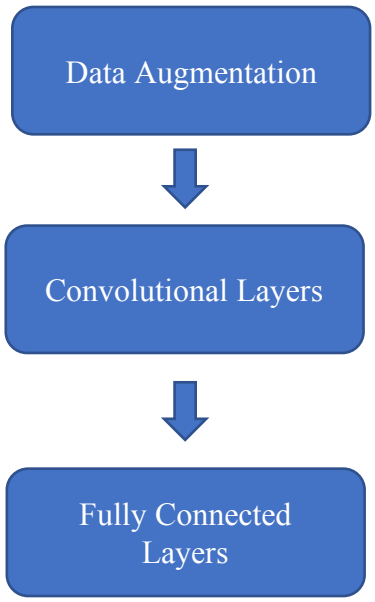
The purpose of this part of the question is to explore different architectures of CNN. I have tried five different architectures for improving the previous section's output. The best accuracy that was achieved by changing the parameters was 99.36% from the base LENET-5 given. This section I have tried various architectures and explained below. I have also improved the accuracy from baseline that I achieved before. The change of architecture also involves work in reducing over-fitting of the model. I believe that most of the above results have very good training accuracy but lesser testing accuracy since the model was overfitting. I have addressed this issue while building up different architecture below.

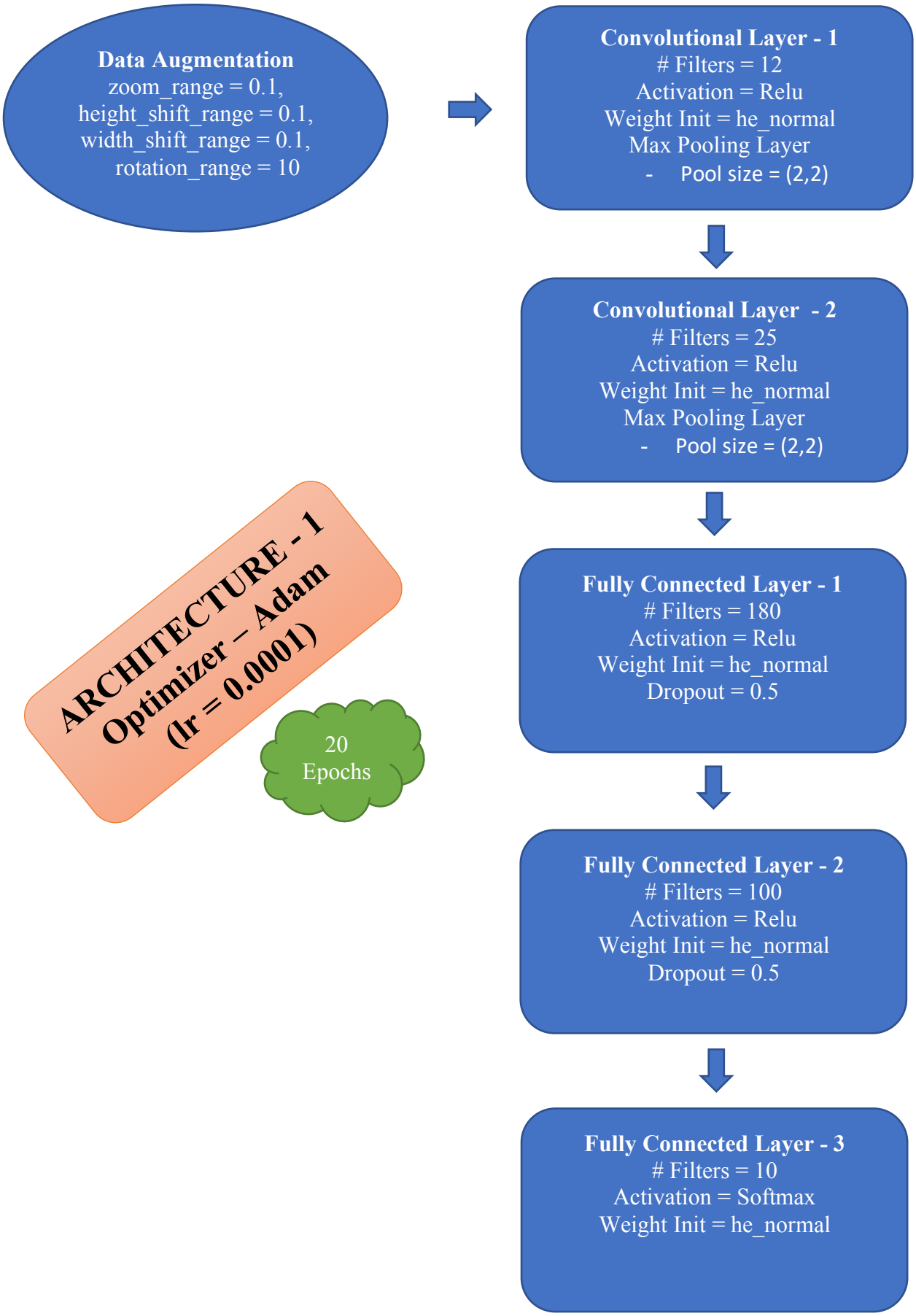
➢ THEORETICAL ANSWERS:

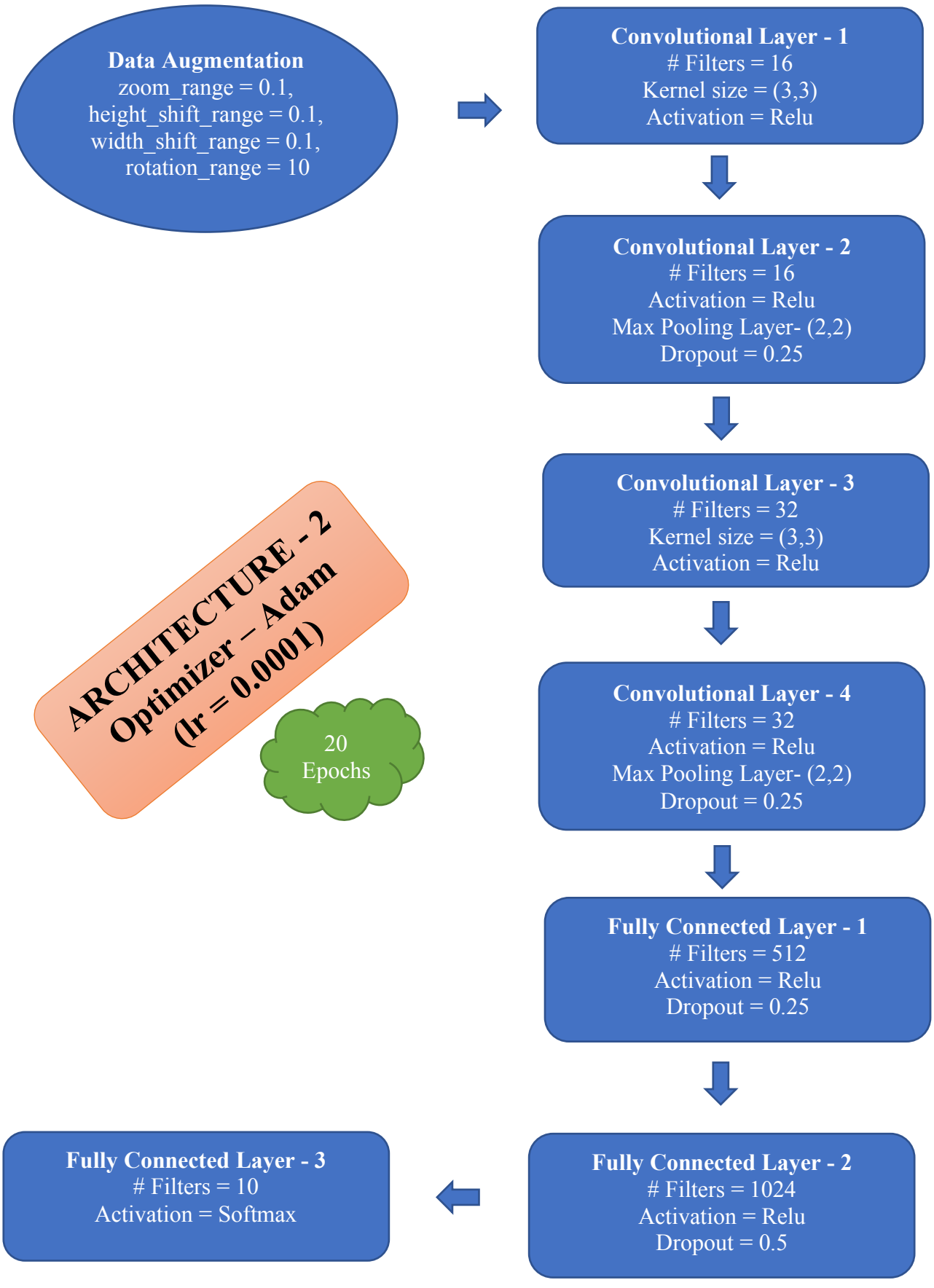
The following 5 different sets of Architectures of CNN were used. I have described every architecture with a block diagram. The important changes from the previous LeNet-5 have been made. Some of them are:

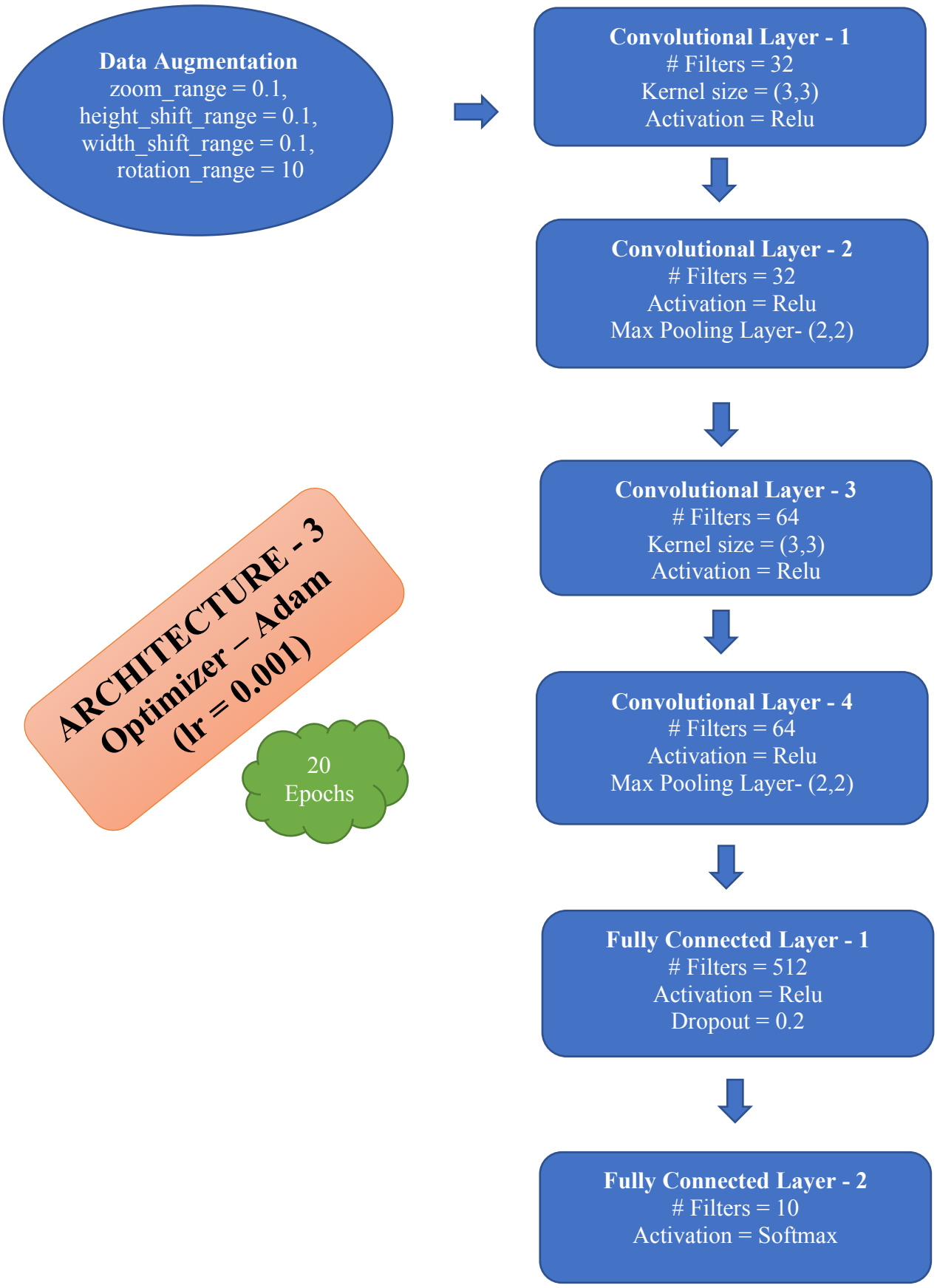
- Data Augmentation
- Add more layers
- Change the number of filters
- Change Filter Size
- Change Optimizer
- Change Weight Initialization

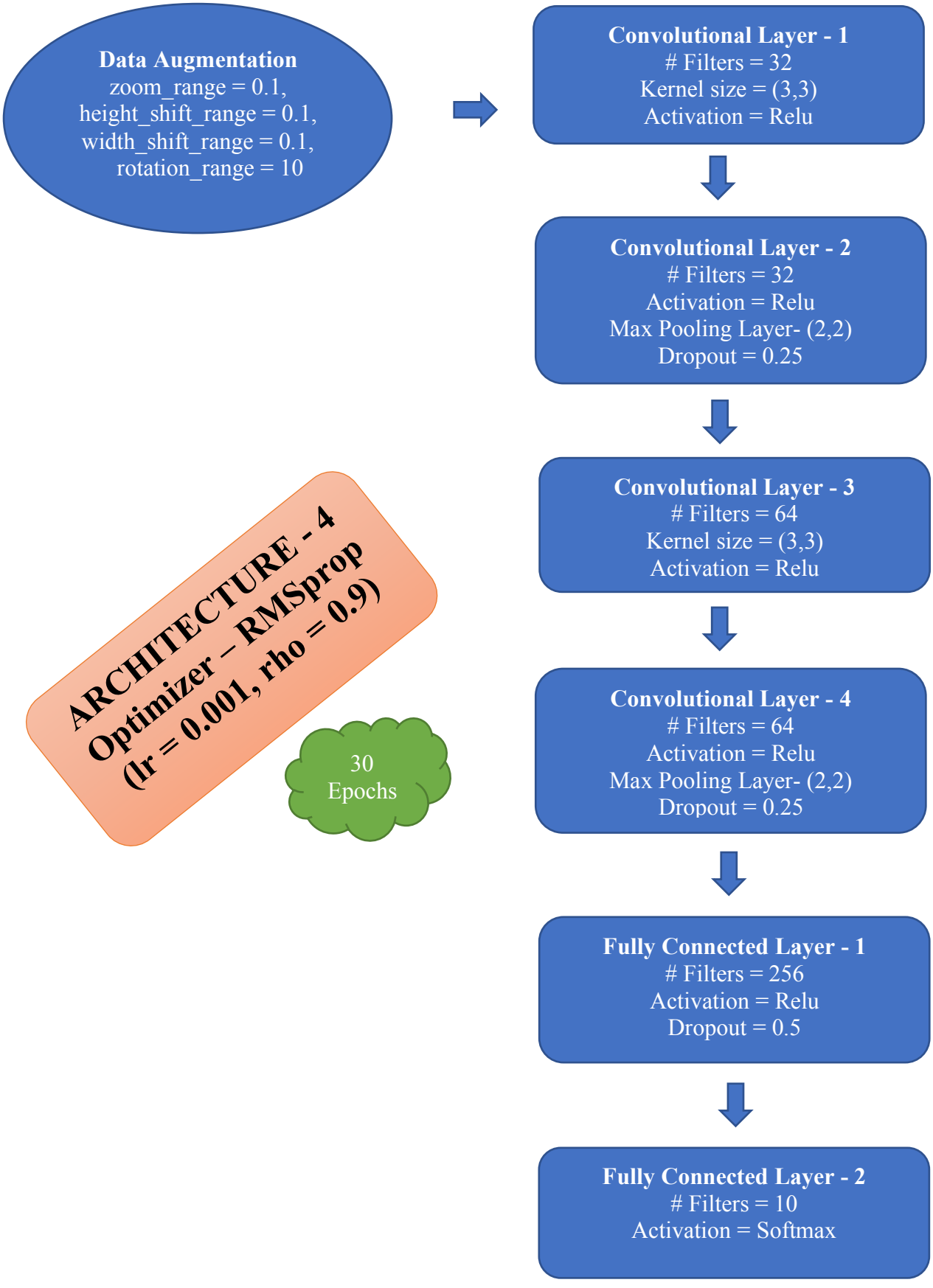
The general architecture is given below. The separate architectures are given in the following pages. The Results are discussed in the Experiments section.

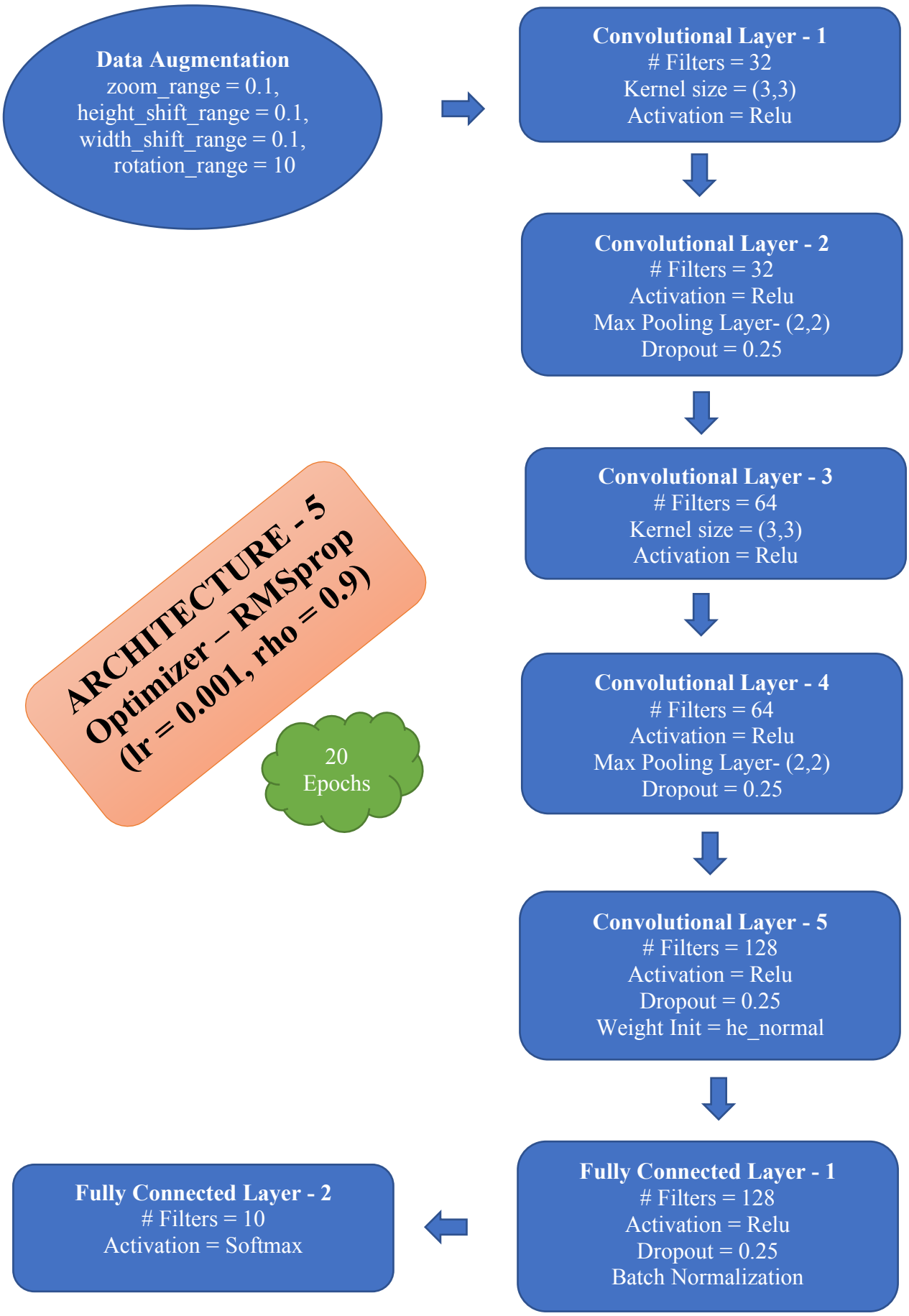



Data Augmentation
zoom_range = 0.1,
height_shift_range = 0.1,
width_shift_range = 0.1,
rotation_range = 10
Convolutional Layer - 1
# Filters = 12
Activation = Relu
Weight Init = he_normal
Max Pooling Layer
- Pool size = (2,2)
Convolutional Layer - 2
# Filters = 25
Activation = Relu
Weight Init = he_normal
Max Pooling Layer
- Pool size = (2,2)
Fully Connected Layer - 1
# Filters = 180
Activation = Relu
Weight Init = he_normal
Dropout = 0.5
Fully Connected Layer - 2
# Filters = 100
Activation = Relu
Weight Init = he_normal
Dropout = 0.5
Fully Connected Layer - 3
# Filters = 10
Activation = Softmax
Weight Init = he_normal
ARCHITECTURE - 1
Optimizer – Adam
(lr = 0.0001)
20
Epochs


Data Augmentation
zoom_range = 0.1,
height_shift_range = 0.1,
width_shift_range = 0.1,
rotation_range = 10
Convolutional Layer - 1
Filters = 16
Kernel size = (3,3)
Activation = Relu
Convolutional Layer - 2
# Filters = 16
Activation = Relu
Max Pooling Layer- (2,2)
Dropout = 0.25
Convolutional Layer - 3
Filters = 32
Kernel size = (3,3)
Activation = Relu
Convolutional Layer - 4
# Filters = 32
Activation = Relu
Max Pooling Layer- (2,2)
Dropout = 0.25
Fully Connected Layer - 1
# Filters = 512
Activation = Relu
Dropout = 0.25
Fully Connected Layer - 2
# Filters = 1024
Activation = Relu
Dropout = 0.5
Fully Connected Layer - 3
# Filters = 10
Activation = Softmax
ARCHITECTURE - 2
Optimizer – Adam
(lr = 0.0001)
20
Epochs


Data Augmentation
zoom_range = 0.1,
height_shift_range = 0.1,
width_shift_range = 0.1,
rotation_range = 10
Convolutional Layer - 1
Filters = 32
Kernel size = (3,3)
Activation = Relu
Convolutional Layer - 2
# Filters = 32
Activation = Relu
Max Pooling Layer- (2,2)
Convolutional Layer - 3
Filters = 64
Kernel size = (3,3)
Activation = Relu
Convolutional Layer - 4
# Filters = 64
Activation = Relu
Max Pooling Layer- (2,2)
Fully Connected Layer - 1
# Filters = 512
Activation = Relu
Dropout = 0.2
Fully Connected Layer - 2
# Filters = 10
Activation = Softmax
ARCHITECTURE - 3
Optimizer – Adam
(lr = 0.001)
20
Epochs

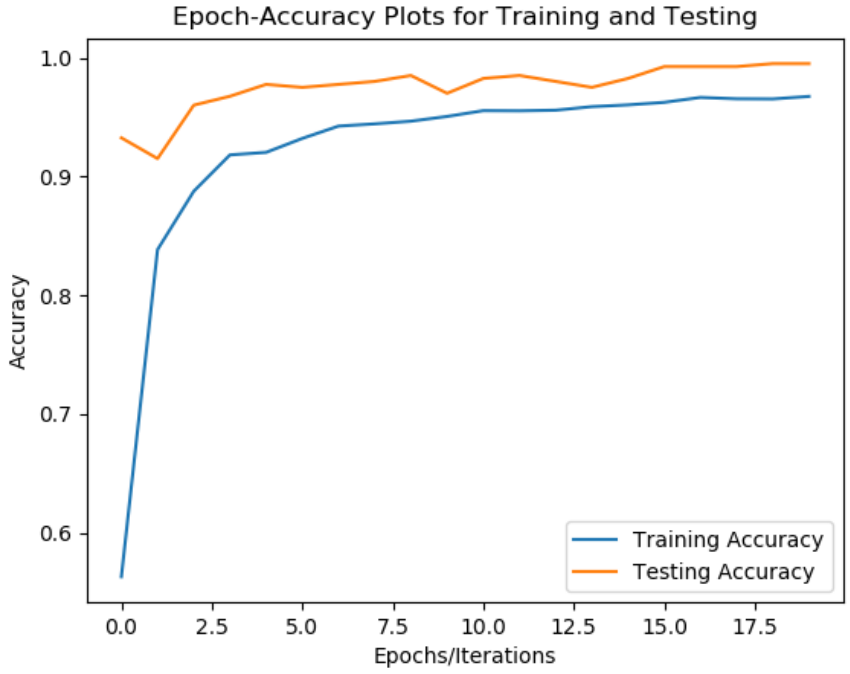

Data Augmentation
zoom_range = 0.1,
height_shift_range = 0.1,
width_shift_range = 0.1,
rotation_range = 10
Convolutional Layer - 1
Filters = 32
Kernel size = (3,3)
Activation = Relu
Convolutional Layer - 2
# Filters = 32
Activation = Relu
Max Pooling Layer- (2,2)
Dropout = 0.25
Convolutional Layer - 3
Filters = 64
Kernel size = (3,3)
Activation = Relu
Convolutional Layer - 4
# Filters = 64
Activation = Relu
Max Pooling Layer- (2,2)
Dropout = 0.25
Fully Connected Layer - 1
# Filters = 256
Activation = Relu
Dropout = 0.5
Fully Connected Layer - 2
# Filters = 10
Activation = Softmax
ARCHITECTURE - 4
Optimizer – RMSprop
(lr = 0.001, rho = 0.9)
30
Epochs


Data Augmentation
zoom_range = 0.1,
height_shift_range = 0.1,
width_shift_range = 0.1,
rotation_range = 10
Convolutional Layer - 1
Filters = 32
Kernel size = (3,3)
Activation = Relu
Convolutional Layer - 2
# Filters = 32
Activation = Relu
Max Pooling Layer- (2,2)
Dropout = 0.25
Convolutional Layer - 3
Filters = 64
Kernel size = (3,3)
Activation = Relu
Convolutional Layer - 4
# Filters = 64
Activation = Relu
Max Pooling Layer- (2,2)
Dropout = 0.25
Convolutional Layer - 5
# Filters = 128
Activation = Relu
Dropout = 0.25
Weight Init = he_normal
Fully Connected Layer - 1
# Filters = 128
Activation = Relu
Dropout = 0.25
Batch Normalization
Fully Connected Layer - 2
# Filters = 10
Activation = Softmax
ARCHITECTURE - 5
Optimizer – RMSprop
(lr = 0.001, rho = 0.9)
20
Epochs

➢ EXPERIMENTAL RESULTS:

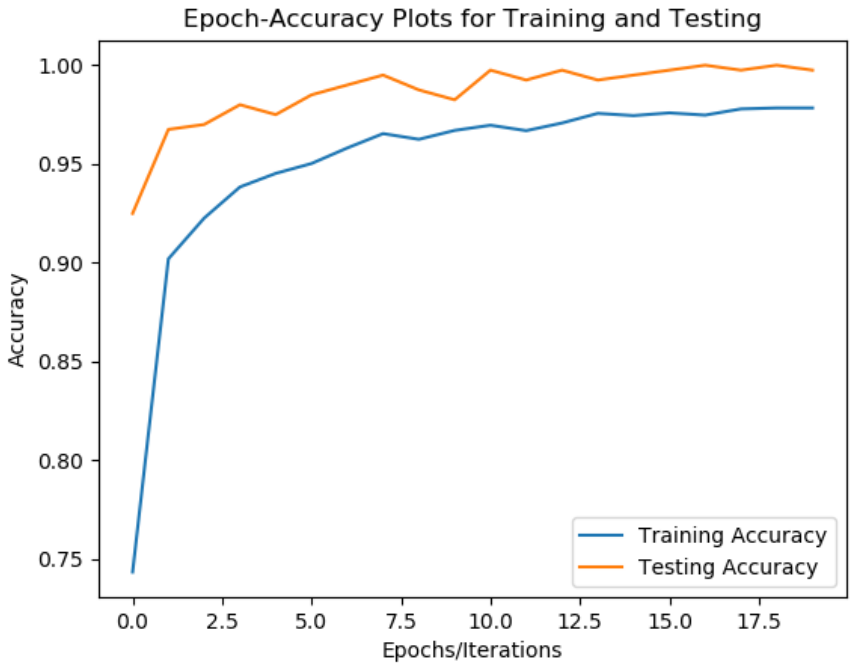
ARCHITECTURE - 1

Epochs	Train Accuracy	Train loss	Test Accuracy	Test loss
0	0.562875	1.2807457	0.9325	0.22982187
1	0.838375	0.52722347	0.915	0.2472382
2	0.8875	0.37744482	0.96	0.11531266
3	0.918125	0.29437286	0.9675	0.08701284
4	0.92025	0.28498097	0.9775	0.08051597
5	0.932	0.24660275	0.975	0.06603081
6	0.942375	0.21825111	0.9775	0.06170074
7	0.94425	0.19622805	0.98	0.04328371
8	0.9465	0.18554613	0.985	0.0515831
9	0.9505	0.17232651	0.97	0.0830431
10	0.955375	0.1679916	0.9825	0.04944265
11	0.95525	0.16904487	0.985	0.0385575
12	0.95575	0.15913911	0.98	0.04928782
13	0.95875	0.1525489	0.975	0.05160531
14	0.96025	0.14211272	0.9825	0.04169566
15	0.962375	0.13510999	0.9925	0.02075591
16	0.9665	0.12813955	0.9925	0.02087256
17	0.965375	0.12782279	0.9925	0.01806396
18	0.96525	0.12515394	0.995	0.02104935
19	0.967375	0.12801638	0.995	0.02379666



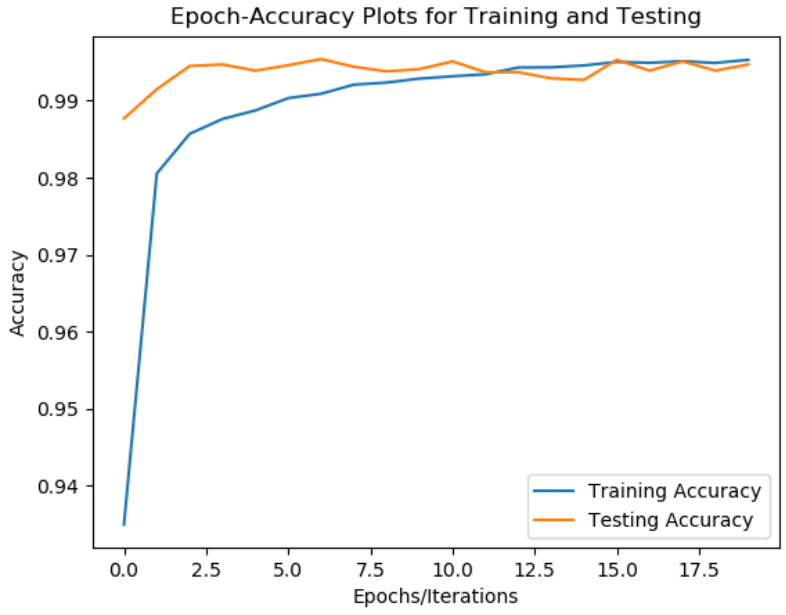
ARCHITECTURE - 2

Epochs	Train Accuracy	Train loss	Test Accuracy	Test loss
0	0.7435	0.82331044	0.925	0.25408816
1	0.902	0.31994825	0.9675	0.09124625
2	0.922625	0.25919591	0.97	0.08177656
3	0.938375	0.20619873	0.98	0.05435642
4	0.94525	0.18184782	0.975	0.05139378
5	0.95025	0.170402	0.985	0.04417474
6	0.958125	0.14903849	0.99	0.03285609
7	0.965375	0.11466673	0.995	0.01698359
8	0.9625	0.12767414	0.9875	0.03411251
9	0.967	0.11711363	0.9825	0.03883691
10	0.969625	0.10563293	0.9975	0.01574267
11	0.966875	0.10996208	0.9925	0.01777324
12	0.97075	0.10077072	0.9975	0.01278832
13	0.975625	0.08703085	0.9925	0.01447107
14	0.9745	0.08560757	0.995	0.01013167
15	0.975875	0.08496218	0.9975	0.00989108
16	0.97475	0.08316443	1	0.00641639
17	0.977875	0.07631549	0.9975	0.00940191
18	0.978375	0.07878401	1	0.00529383
19	0.978375	0.06825292	0.9975	0.00547188



ARCHITECTURE - 3

Epochs	Train Accuracy	Train loss	Test Accuracy	Test loss
0	0.93498065	0.20422795	0.98768029	0.03363816
1	0.98054592	0.0627229	0.99148638	0.02418862
2	0.98568473	0.04718298	0.99449119	0.01840032
3	0.98762013	0.03917961	0.99469151	0.0173949
4	0.9887213	0.0348562	0.99389022	0.01810353
5	0.99032301	0.03164946	0.99459135	0.01753368
6	0.99089028	0.02922846	0.99539263	0.01496447
7	0.99207488	0.02680324	0.99439103	0.01647678
8	0.99234183	0.02467362	0.99379006	0.01899158
9	0.99285905	0.02286008	0.99409054	0.01934323
10	0.99315937	0.02182211	0.99509215	0.02069395
11	0.99340964	0.02061808	0.9936899	0.02085938
12	0.99429391	0.01900568	0.9936899	0.02325151
13	0.99432728	0.01850609	0.99288862	0.02370145
14	0.99456086	0.01786141	0.9926883	0.02675337
15	0.99502803	0.01738284	0.99529247	0.01474805
16	0.99491124	0.01682942	0.99389022	0.02112335
17	0.99511145	0.01487415	0.99509215	0.01679414
18	0.99489455	0.01687408	0.99389022	0.01791229
19	0.99529498	0.01588184	0.99469151	0.02011922



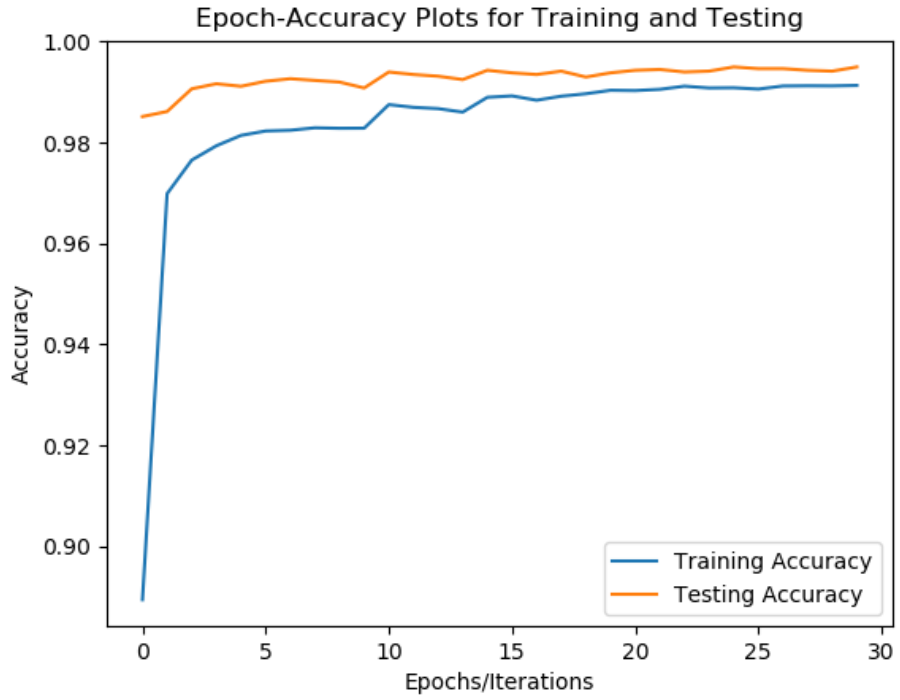
ARCHITECTURE - 4

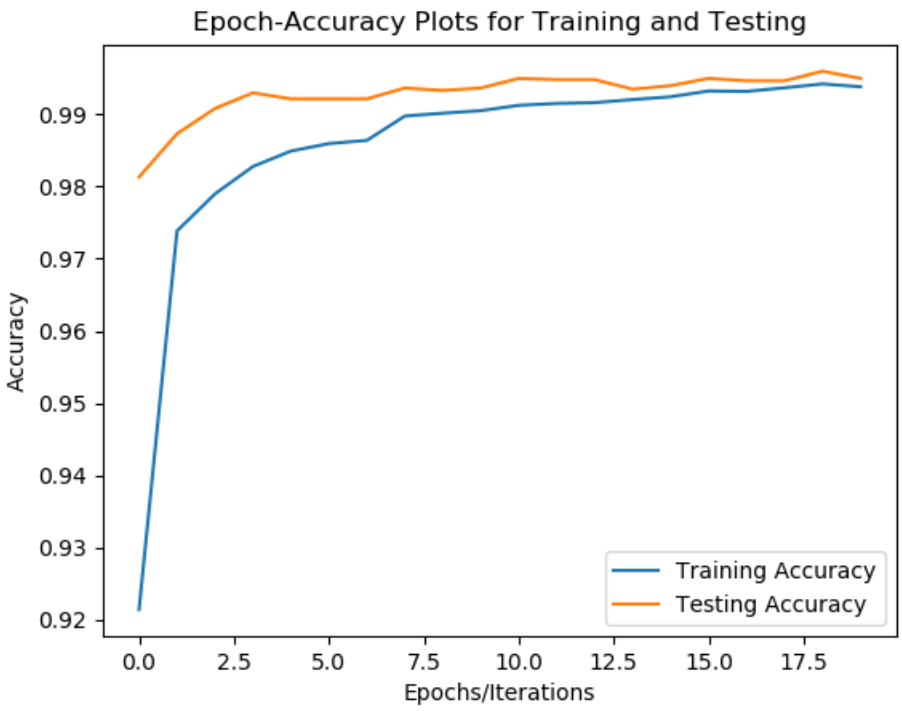
Epochs	Train Accuracy	Train loss	Test Accuracy	Test loss	Epochs
0	0.88941647	0.34554644	0.001	0.98516666	0.04336728
1	0.96987794	0.1031963	0.001	0.98616666	0.04036113
2	0.9765367	0.07786082	0.001	0.99066666	0.03690475
3	0.97937455	0.06890426	0.001	0.99166666	0.03331491
4	0.98143339	0.06428761	0.001	0.99116666	0.02856541
5	0.98230514	0.06182854	0.001	0.99216666	0.03391289
6	0.98245353	0.06299773	0.001	0.99266666	0.02978144
7	0.98293578	0.06249887	0.001	0.99233333	0.02679122
8	0.98284304	0.06352232	0.001	0.992	0.03125018
9	0.98286159	0.06331204	0.001	0.99083333	0.03817346
10	0.9875357	0.04652774	0.0005	0.994	0.02366194
11	0.98699781	0.0494646	0.0005	0.9935	0.02275992
12	0.98673813	0.04844474	0.0005	0.99316666	0.02249418
13	0.98605185	0.05011234	0.0005	0.9925	0.02916226
14	0.98898245	0.04012774	0.00025	0.99433333	0.02228283
15	0.98926067	0.03818279	0.00025	0.99383333	0.02160743
16	0.98840746	0.04005367	0.00025	0.9935	0.02703567
17	0.98920503	0.03962215	0.00025	0.99416666	0.02488797
18	0.98968727	0.03758382	0.000125	0.993	0.0222714
19	0.99037355	0.03455557	0.000125	0.99383333	0.02154872
20	0.99031791	0.03286288	0.000125	0.99433333	0.02061648
21	0.99055903	0.03357272	6.25E-05	0.9945	0.02056437
22	0.99118967	0.03046222	6.25E-05	0.994	0.02145301
23	0.99083725	0.03183482	6.25E-05	0.99416666	0.0203633
24	0.99087435	0.03111897	6.25E-05	0.995	0.01918547
25	0.99061468	0.03268595	6.25E-05	0.99466666	0.02196231
26	0.99122676	0.03111242	6.25E-05	0.99466666	0.02082662
27	0.99126386	0.03143463	6.25E-05	0.99433333	0.02344021
28	0.99124531	0.03134122	3.13E-05	0.99416666	0.02130853
29	0.99133805	0.03196559	3.13E-05	0.995	0.01999932

ARCHITECTURE - 5

Epochs	Train Accuracy	Train loss	Test Accuracy	Test loss	Epochs
0	0.92138831	0.25021377	0.001	0.98133333	0.05904751
1	0.97385791	0.08629916	0.001	0.98733333	0.04840163
2	0.97899362	0.06700407	0.001	0.99083333	0.02925738
3	0.98279442	0.05496856	0.001	0.993	0.02681591
4	0.98492658	0.04990134	0.001	0.99216667	0.02765443
5	0.98596485	0.04712424	0.001	0.99216667	0.02276857
6	0.98640982	0.04294838	0.001	0.99216667	0.0237421
7	0.98978419	0.03412648	0.0005	0.99366667	0.02397687
8	0.99017354	0.03278586	0.0005	0.99333333	0.02077568
9	0.99052581	0.03257487	0.0005	0.99366667	0.01954298
10	0.99126743	0.02797431	0.0005	0.995	0.01504051
11	0.99152699	0.02886335	0.0005	0.99483333	0.01801919
12	0.99163824	0.02663329	0.0005	0.99483333	0.02174577
13	0.99206467	0.027549	0.0005	0.9935	0.01923859
14	0.99245402	0.02478634	0.00025	0.994	0.0196623
15	0.99325126	0.02227541	0.00025	0.995	0.01687662
16	0.99319564	0.02231253	0.00025	0.99466667	0.0177709
17	0.99369623	0.02070635	0.000125	0.99466667	0.01692588
18	0.99423391	0.01879618	0.000125	0.996	0.01619474
19	0.99384456	0.02071652	0.000125	0.995	0.01734916

BEST PERFORMING RESULTS – ARCHITECTURE 4 AND 5 (> 99.6%)





ARCHITECTURE	TEST ACCURACY (%)
LeNet-5 Best from Section B	99.36
1	98.68
2	99.32
3	99.47
4	99.61
5	99.69

- **DISCUSSION:**

Thus, I varied different parameters and created different architectures, changing the baseline LeNet-5 architecture. Some observations are given below:

- Architecture 4 and 5 gives best performance having Testing Accuracy greater than 99.6%
- Adding more layers actually increases the performances of the Training and Testing MNIST Digits classification
- Increasing the filter size and number of filters also increases the performance. But we should be careful in increasing the values, since it might start overfitting
- RMSprop and Adam Optimization give good performances than any other optimizers
- Overfitting reduces the performance of the model. Hence techniques like Data Augmentation, adding Dropouts after some layers avoids overfitting
- Good architectures don't overfit the training data and the model is trained best even with lesser number of epochs
- The best performing models, have epoch-testing accuracy curve higher than epoch-training accuracy curve. This shows that the model is not over-fitting. If the model is over-fitting, the epoch-training accuracy curve would be on the top than the testing curve
- With good architectures, we could increase the performance from 99.36% to 99.69%

PROBLEM 2
SAAK TRANSFORM AND ITS APPLICATION TO THE MNIST DATASET

A) COMPARISON BETWEEN SAAK TRANSFORM AND CNN ARCHITECTURE

➢ ABSTRACT AND MOTIVATION:

This part of the question is to understand SAAK (Subspace Approximation with Augmented Kernels) Transform. This understanding would make us find differences between CNN and SAAK Transform. Convolutional Neural Network offer the state-of-art performance in hand-written digits recognition nowadays. I explained in previous section, how LeNet-5 architecture is formed with 2 Convolutional Layers and 3 Fully Connected Layers. I also explained how CNNs efficiency can be increased by changing various parameters like number of layers, number of filters in a layer, filter size etc. In this section of the question, I have explained Saak Transformation proposal for hand-written digits recognition. I also have explained the main differences between Saak Transformation and CNN architecture.

➢ THEORETICAL ANSWERS:

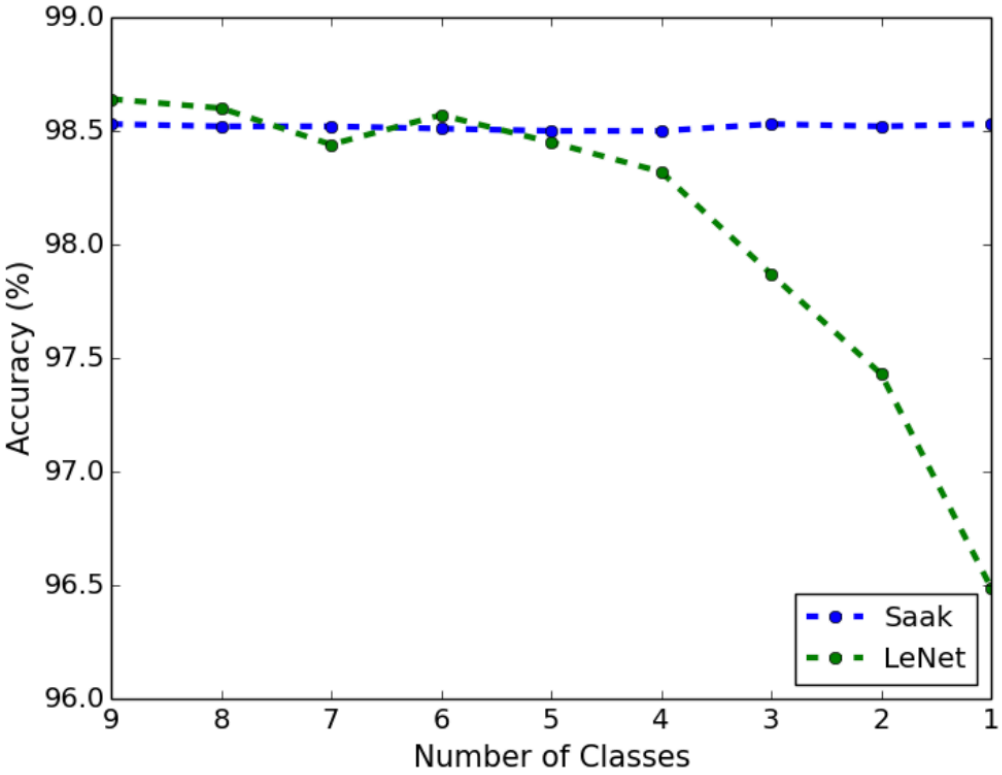
Two main ingredients for Saak Transformation proposed by Prof Kuo and Chen are:

- Subspace Approximation
- Kernel Augmentation

On a generic note, the Saak Transform has both forward and inverse transforms which has an advantage that the model can be used for both Image Analysis (like Classification) as well as data generation. On the other hand, both the methods have Convolution with activation function as Relu in the initial layers. Even though, they have these operations in common, there are some distinguishable factors. They are explained below.

End-to-end Optimization versus Modular Design:

- Saak Transformation technique is considered to be a Modular Design whereas, CNNs are considered to be an end-to-end optimization technique
- Due to nature of the technique, where the weights are forward and backward propagated to get the optimized weights, CNNs suffer from a variety of disadvantages. They are listed below:
  - Robustness
  - Scalability while training for different class numbers
  - Portability when there are different datasets
- Most of the disadvantages stated above for CNN are because, each network is trained for a specific dataset. And only a few parameters work unexplainable well for a dataset and generalization becomes weaker. Thus, when considering Robustness or Scalability for different datasets or different number of classes, CNNs performance decreases
- On the other hand, Saak Transform has modularized Feature Extraction and Classification like traditional methods which helps in better generalization. This in turn helps the model to be robust and scale well for different number of classes
- The reason for better Saak Transform is also reportedly by using of dimensionality reduction techniques like PCA



The above results show that Saak Transform looks good constant performance over different number of classed. CNNs performance is different for different classes. Thus, Scalability is a problem as discussed above with CNN.

Generative Model versus Inverse Transform:

- For all purposes we can assume that there are no Inverse Transform studied for CNN except for inverse RECOS. But there are CNN based approaches to generate images which involves the following steps
  - Generative Adversarial Network (GAN) – train a generative network and a discriminative network
  - Variational AutoEncoder (VAE) – train a encoder network and a decoder network
- But this is not so complicated in Saak Transform based Image generation since there is a forward and inverse method inherent. Given such a method, for any ideal arbitrary input, the forward and inverse multi-stage Saak Transform results in an identity operator.
- Even in non-ideal conditions, if there is one high resolution image, we can model forward and inverse transforms for both low-resolution image and its high-resolution image.
- This can be looked as forming a bridge between different resolution images to transform one to another
- This kind of inverse transform applications may pave way for future research and its various applications

Theoretical Foundation:

- The main reason behind some researches not supporting the results of CNN is that, it lacks complete theory on what is happening at every layers.
- There are various intuitions that the researchers have come up with. They include:
  - Multilayer Perceptrons approach
  - Scattering Networks
  - Tensor Analysis
  - Relevance propagation
  - Taylor Decomposition
  - The multi-layer convolutional sparse coding
  - Over-parameterized shallow neural network optimization
- Despite the above-mentioned methods were proposed on to understand what is mathematically going behind every layer, as the network size grows and as the complexity increases, the mathematics behind the process and model becomes intractable
- Conversely, Saak Transform is complete linear algebra and statistics and each and every step is recognizable and easy to understand the mathematics backend model

Filter Weight Determination, Feature Selection and Others:

CNN	SAAK TRANSFORM
Automatic filter weight selection scheme by optimizing an objective function. This is done iteratively	Kernel weights are found using KLT stage by stage. This is also done automatically, and they do not come under hand-crafted features
Data labels are needed for back propagation to determine the loss from previous epoch to propagate it and adapt the weights	Data labels are only needed in the decision stage (i.e.) during Classification stage and not in the feature extraction stage
CNN comes under 'deep' learning because the weights are learned in such a way. These are called iteratively optimized features	To obtain Saak Features, the labels are used in a different way. For obtaining Multi-Stage Saak Coefficients, coefficients of one classes is computed to build features of the same class
CNN conducts Convolutional Operations on overlapping blocks/cuboids. Also adapts Pooling methods to reduce the dimension to avoid over-fitting	Saak Transform is applied on non—overlapping blocks. Thus, no special technique is explicitly needed to reduce the spatial dimension. It is automatically taken care of
Horizontal and Vertical dimensions of CNN typically take odd numbers like 3*3, 5*5 etc.	The spatial dimension of features from Saak Transform do not have any restrictions on odd or even numbers
Parameter settings for CNN is like prior art as a convention (i.e.) they are mostly on trial and error basis	The Parameter settings for Saak Transform can be modulated using proper theoretical knowledge and support

B) APPLICATION OF SAAK TRANSFORM TO MNIST DATASET

➢ ABSTRACT AND MOTIVATION:

This part of the question is for us to get hands on implementation on Saak Transform and see how well it performs on MNIST Dataset. In the theory section I have explained the Saak theory and in the experimental results section I have given a table of results that has accuracy and performance. This section will hopefully help me understand the mathematics behind the Saak Transform since it is not a complete black box kind of methodology like CNN.

➢ APPROACH AND PROCEDURES:

Saak Transform has the following main steps as explained in the base paper by Prof Kuo: (Source: https://arxiv.org/abs/1710.04176)

- Optimal Linear Subspace Approximation is built
  - This is done using the second order statistics of input vectors
  - The Karhunen-Loeve Transform (KLT) is used to get the second order statistics in turn to approximate the subspace
- Negative of each transformed kernel is obtained and augmented with them
- As it is most cases proved that ReLU activation function performs the best to capture non-linearity in the patterns, Saak Transform also uses ReLU for transformation of output

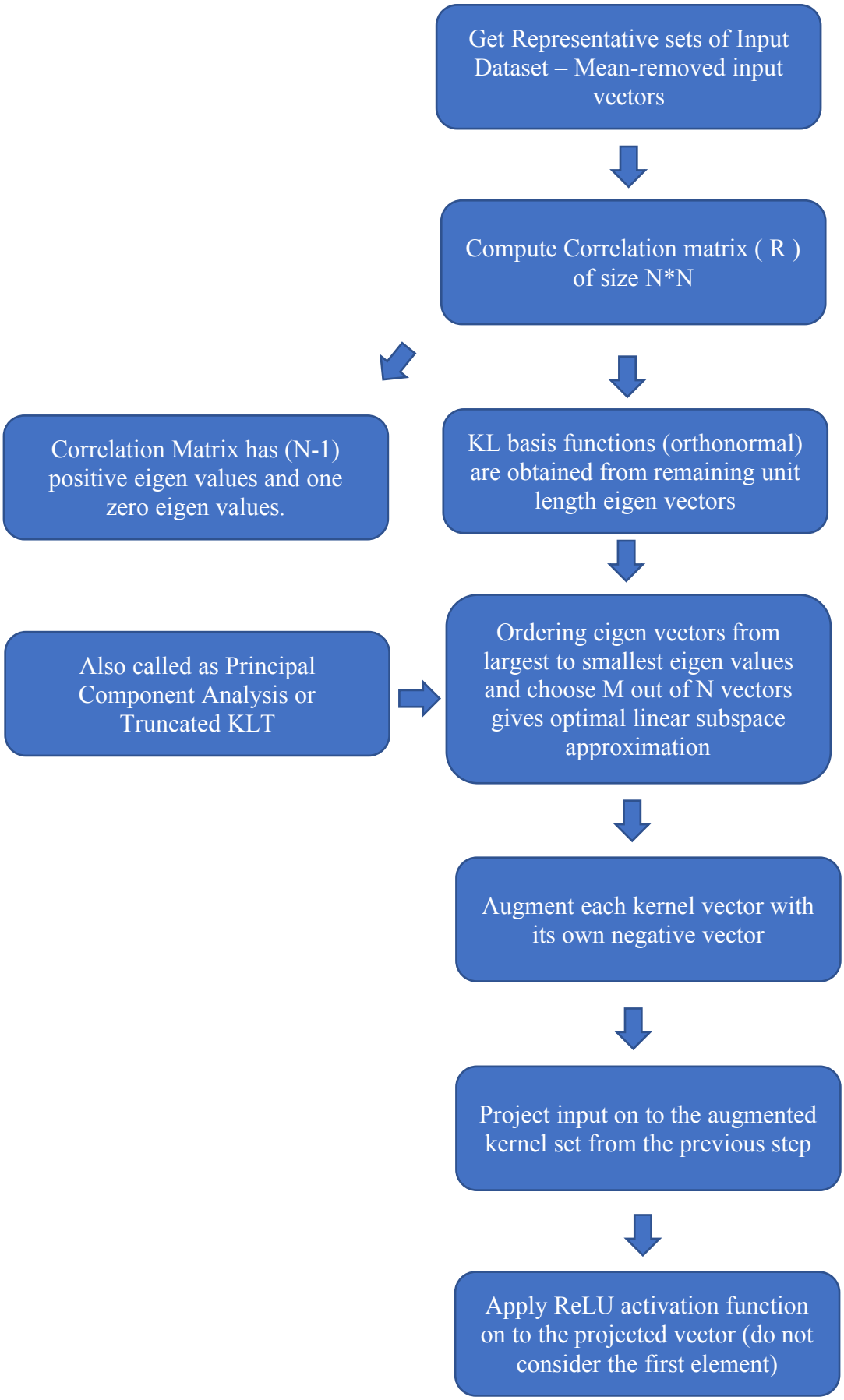
With this initial understanding of overview of Saak Transform, a flow chart has been provided in the next page to understand the entire process clearly. Saak Transform is an extension of forward/inverse RECOS transform. Saak Transform was proposed in an attempt to avoid the limitations of RECOS transform. The limitations are stated as below.

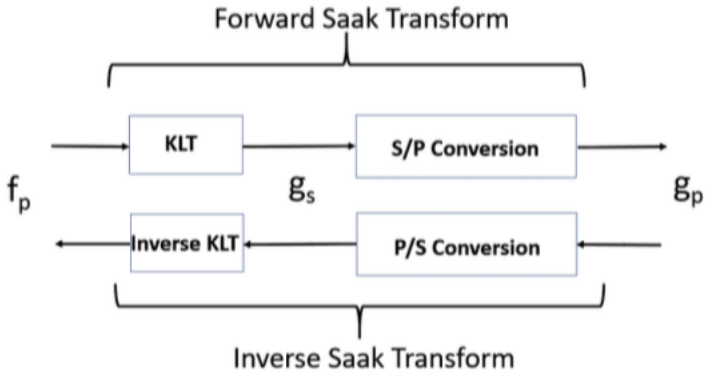
- The anchor vectors from RECOS transform are not orthonormal transfer functions and it is hard to facilitate forward and inverse transforms.
- RECOS transform cannot alter approximation loss to be arbitrarily small, if the architecture is fixed. There is a restriction after fixing the architecture that they cannot be made less than certain value
- RECOS transformation has a rectification operation which makes the signal representation less efficient. It is always needed to recover the representation ability of these rectified anchor vectors

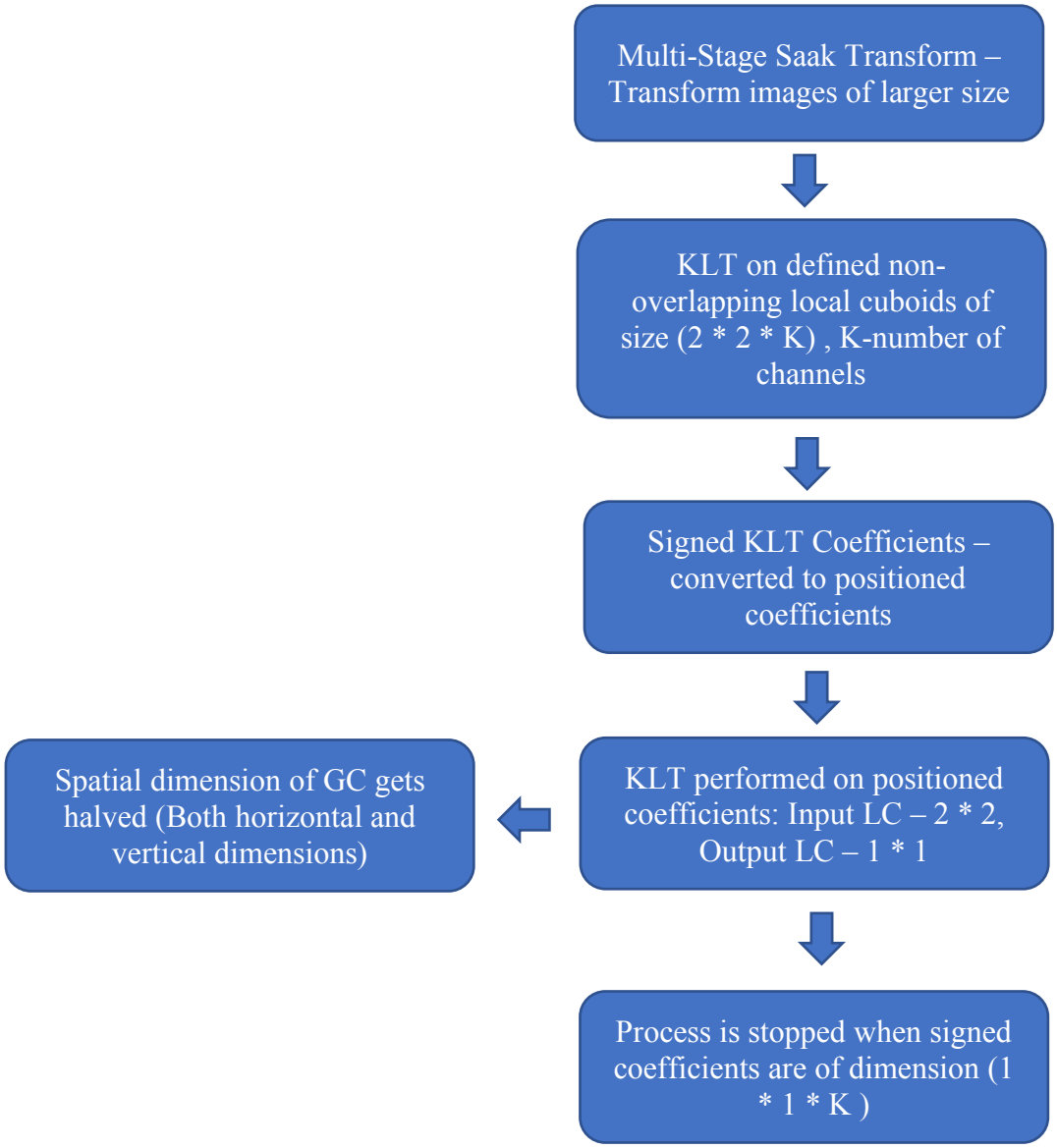
To solve these limitations, SAAK Transform has come up with two different methods:

- To overcome the first two limitations of RECOS transformation, SAAK transformation gets orthonormal kernel functions from KLT Transformation.
- To overcome the third limitation, Augmentation of the negative vectors of the kernel is used thus recovering the representation ability of the transformation kernel vectors

The forward Saak Transform is explained in the flow chart below. Inverse Saak Transform is to get the desired input to give to inverse KLT by using Position-to-sign format conversion. Details of Multistage Saak transformation is given after the flow chart.


Get Representative sets of Input Dataset – Mean-removed input vectors
Compute Correlation matrix (R) of size N*N
KL basis functions (orthonormal) are obtained from remaining unit length eigen vectors
Correlation Matrix has (N-1) positive eigen values and one zero eigen values.
Ordering eigen vectors from largest to smallest eigen values and choose M out of N vectors gives optimal linear subspace approximation
Also called as Principal Component Analysis or Truncated KLT
Augment each kernel vector with its own negative vector
Project input on to the augmented kernel set from the previous step
Apply ReLU activation function on to the projected vector (do not consider the first element)

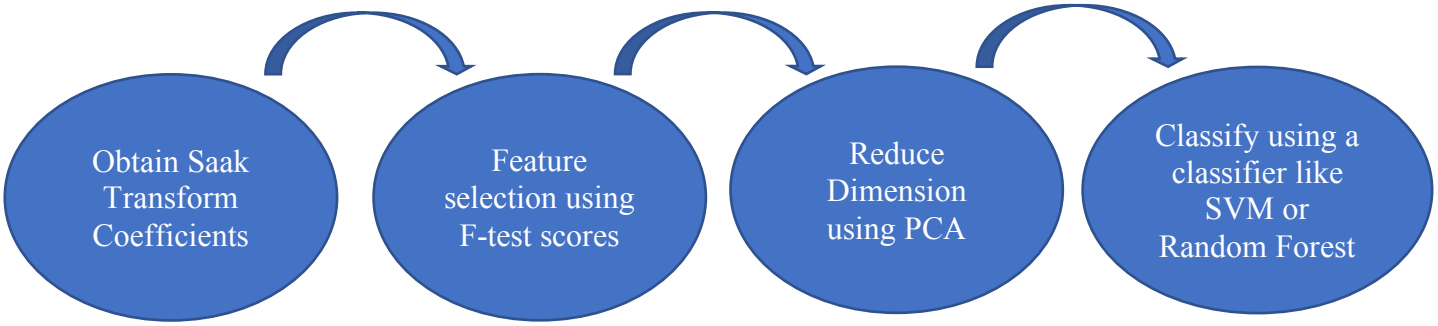




Multi-stage Saak Transform to get signed coefficients

Thus, using the above-mentioned process, Saak Transform for a given set of input would be calculated. This would give anchor vectors that recede on direction of maximum variance.

- From the obtained Saak Transform Coefficient Feature vectors, feature selection can be made by using F-test statistic of the ANOVA (ANalysis Of VAriance)
- F score = Between-Group variability (BGV) / Within-Group Variability (WGB)
- The features can be selected based on their F-test scores, like we can select features that have higher scores
- Once the features are selected, even after that the dimensions may be high. So PCA or any other dimensionality reduction techniques can be followed to reduce the dimension
- Once the dimensionality has been reduced, they can be used as input features for any classifier like SVM or Random Forest.



Implementation Details (Using Github code already present with some changes made):

- For Implementation of Saak Transform, I have used online code available as open source from this link - https://github.com/morningsyj/Saak-tensorflow
- Certain modifications were made to adapt the code to the needed question.
- In util.py file I made the following change in keeping the number of components

Stage	Components to keep
0	3
1	4
2	7
3	6
4	8

- I wrote functions to calculate F-test score using f_classif
- I wrote another function to define and transform data using Sklearn PCA model. I have tried on different PCA components like 32, 64 and 128
- I also wrote functions to call SVM or Random Forest Classifier to be called in the end. This classifier models are then used to predict labels and get accuracy values.

➢ EXPERIMENTAL RESULTS:

Experiment No	PCA Components	Classifier Used	Validation Size	Train Accuracy (%)	Test Accuracy (%)
1	32	SVM	5K	99.236	98.34
2	64	SVM	5K	98.899	98.31
3	128	SVM	5K	98.092	97.78
4	32	Random Forest	5K	99.870	93.02
5	64	Random Forest	5K	99.899	92.35
6	128	Random Forest	5K	99.898	89.93

➢ DISCUSSION:

From the above Saak Transformation results and CNN results from the previous question, certain comparisons could be made. From those comparison, some of the observations are given below:

- The table shows the results which had best test accuracy from different sections of this project

Experiment	Training Accuracy (%)	Testing Accuracy (%)
Question 1-b (Best LeNet-5)	99.89	99.36
Question 1-c (Best CNN Architecture)	99.38	99.69
Question 2-b (Best Saak Transform)	99.23	98.30

- From the above results, I can see from Testing Accuracy, that Saak Transform is very closely performing to CNN.
- This is a good start for Saak Transform since even though this method of proposal is in initial stages of research, I could see it is closely performing to CNN, where CNN is assumed to be the future of computer vision.
- Previous question results for different hyper parameter settings, CNN accuracy for the testing data was as low as 10%. Hence CNN seems to be very sensitive on parameter tuning and right parameters have to be chosen to get good accuracy
- But this is not the case with Saak transform, since for different hyper parameters, the Saak Transform Testing accuracy does not go less than 89%. This gives a hint that Saak Transform based results are more stable than CNN based results
- Some of the advantages of Saak Transform over CNN I could see after experimenting on MNIST dataset are listed below:
    - Saak is computationally faster and way better than CNN
    - Unlike CNN, the results of Saak are stable and not very sensitive to hyper parameters like PCA components and the type of classifier
    - Saak can give better results on even lesser training data and scales well to the size of training data

C) ERROR ANALYSIS

➢ ABSTRACT AND MOTIVATION:

This section of the question is again another open-ended question where I tried to analyze the outputs of CNN and Saak by individually checking the errors of both the methods on the MNIST dataset. I have analyzed and reported below on different errors from CNN and Saak. I have also analyzed errors that is common for both CNN and Saak Transform. I could come up with some conclusions on why there are common errors and why some images are correctly classified in one method and not in another. Also, I have tried to come up with improvement methods that can be of future research to improve the Image Classification problem in general.

➢ THEORETICAL ANSWERS:

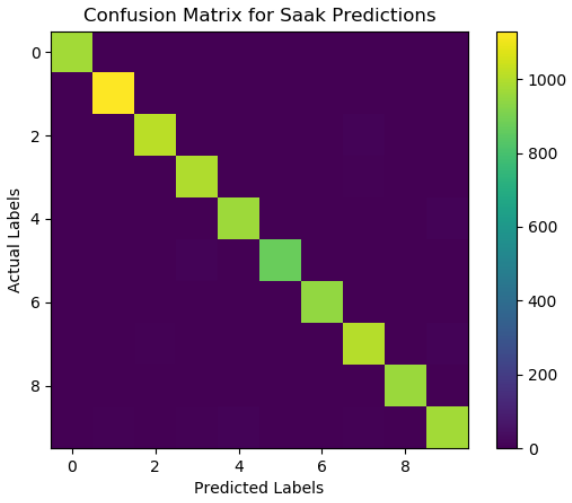
Error Analysis:

I wrote a code that took the predicted labels of the MNIST dataset from the **best performing CNN and Saak Transform model**. Using the Predicted Labels and the Actual labels, the code performed three types of analysis:

- Images Misclassified by CNN but Correctly Classified by Saak Transform
- Images Misclassified by Saak Transform but Correctly Classified by CNN
- Images Misclassified by both CNN and Saak Transform

```
------------------
Confusion Matrix for SAAK Prediction:
[[ 973    0    1    0    0    1    2    1    2    0]
 [   0 1130    2    0    0    1    0    0    1    1]
 [   4    0 1013    1    1    0    0    9    3    1]
 [   0    0    2  995    0    3    0    6    3    1]
 [   1    0    1    0  963    0    3    0    1   13]
 [   2    0    0    9    1  873    2    1    2    2]
 [   3    2    0    1    2    2  946    0    2    0]
 [   0    4    7    0    0    0    0 1006    1   10]
 [   2    0    1    3    2    2    1    2  959    2]
 [   3    6    0    7    9    2    1    5    2  974]]

Confusion Matrix for CNN Prediction:
[[ 978    0    0    0    0    1    0    1    0    0]
 [   0 1131    0    1    0    0    1    2    0    0]
 [   0    0 1028    1    0    0    0    2    1    0]
 [   0    0    0 1009    0    1    0    0    0    0]
 [   0    0    0    0  977    0    0    0    1    4]
 [   0    0    0    6    0  885    1    0    0    0]
 [   2    1    0    1    0    0  951    0    3    0]
 [   0    2    1    0    0    0    0 1025    0    0]
 [   0    0    1    1    1    0    0    0  971    0]
 [   0    0    0    0    1    0    0    1    0 1007]]
```

```
In [12]: runfile('/Users/abinaya/USC/Studies/DIP/Homeworks/hw-4/hw_4_answers/hw_4_prob_2_c/compare_results.py', wdir='/Users/abinaya/USC/Studies/DIP/Homeworks/hw-4/
hw_4_answers/hw_4_prob_2_c')
SAAK Accuracy Score (%):  98.32
CNN Accuracy Score (%):  99.62

--------------------
Count of CNN Misclassified:  38
Count of CNN Misclassified that SAAK Predicted Correctly:  20
About 52.6315789474% of CNN Misclassified, SAAK predicted correctly
Image Labels of this category:  [ 413  675  717  939 1404 1439 1460 1879 2036 2448 2455 2598 3535 3763
 4861 5655 6559 8280 9643 9694]

--------------------
Count of SAAK Misclassified:  168
Count of SAAK Misclassified that CNN Predicted Correctly:  150
About 89.2857142857% of SAAK Misclassified, CNN predicted correctly
Image Labels of this category:  [ 152  248  321  322  341  382  446  448  449  450  496  583  629  690
  708  721  741  883  925  948  952  960 1045 1063 1113 1195 1227 1233
 1243 1248 1261 1300 1320 1329 1501 1523 1531 1550 1554 1582 1682 1710
 1755 1791 2045 2071 2099 2110 2136 2190 2294 2300 2372 2388 2407 2415
 2423 2489 2608 2649 2772 2811 2864 2897 2928 2940 2954 2996 3061 3074
 3118 3334 3370 3476 3491 3504 3521 3559 3560 3598 3619 3768 3777 3797
 3809 3812 3854 3903 3942 3969 3986 4066 4076 4079 4164 4225 4249 4290
 4307 4370 4498 4501 4640 4741 4808 4824 4880 4881 4887 5458 5601 5643
 5735 5836 5937 5956 5973 5974 5983 5998 6036 6167 6506 6561 6572 6599
 6652 7217 7435 8095 8326 8340 8409 9010 9016 9025 9588 9635 9665 9680
 9699 9746 9750 9769 9771 9793 9809 9840 9923 9945]

--------------------
Count that Both CNN and SAAK Misclassified:  18
Percentage of the MNIST DATASET that both Misclassified: 0.18%
Image Labels of this category:  [ 660 1015 1040 1394 1902 2131 2183 2463 2655 3423 4177 4202 5938 6577
 6598 6626 8528 9730]
```



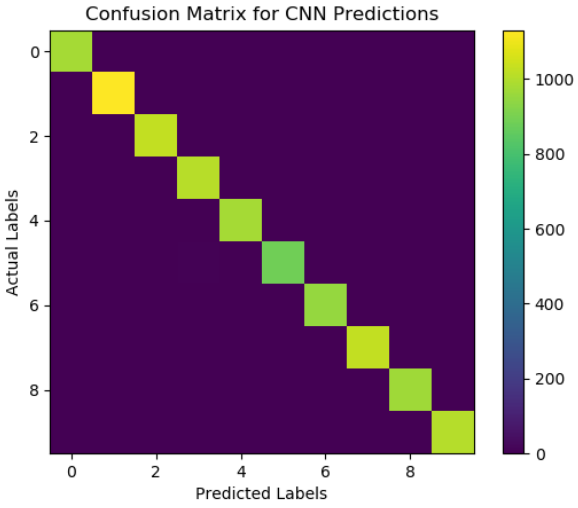
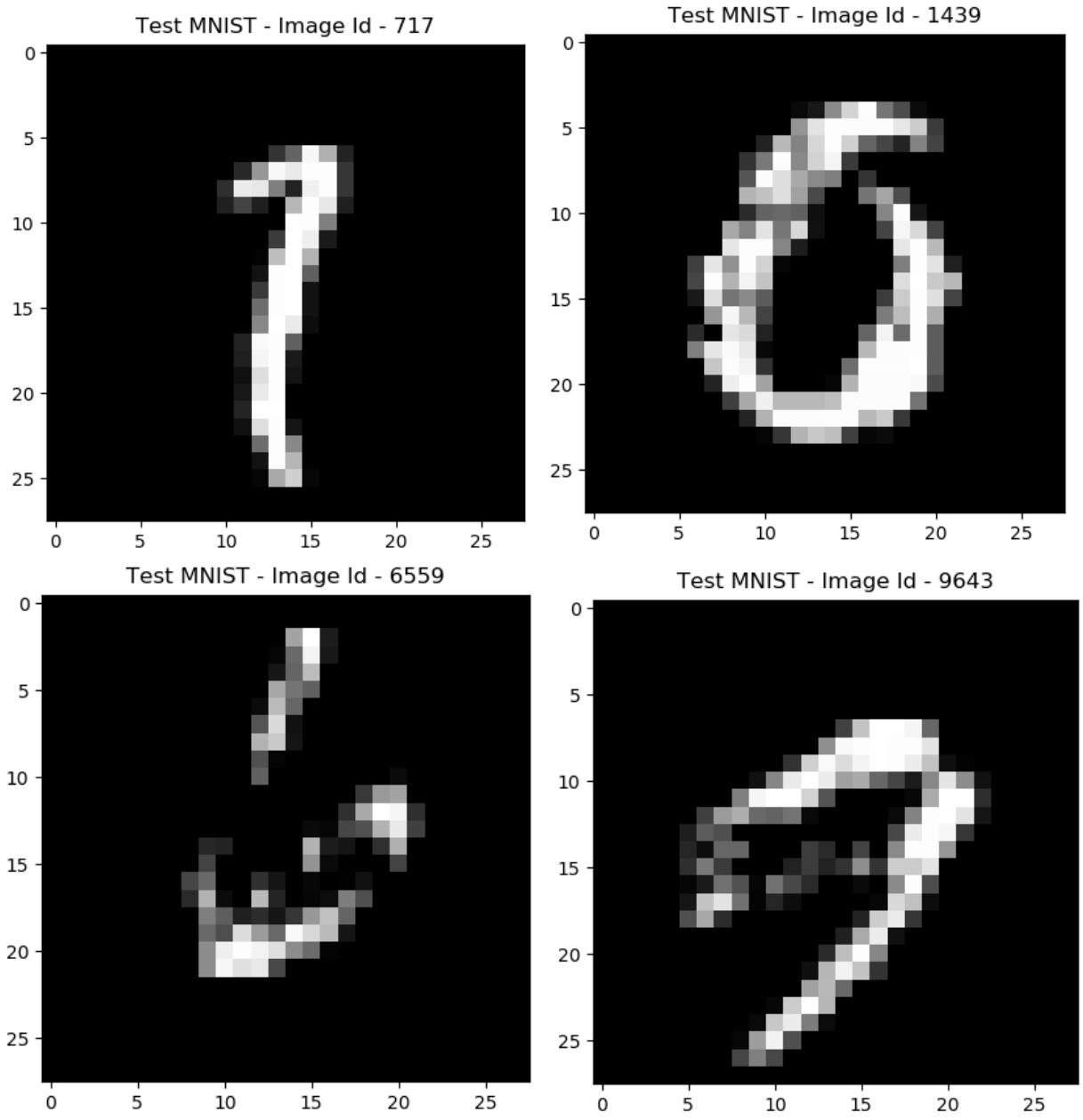


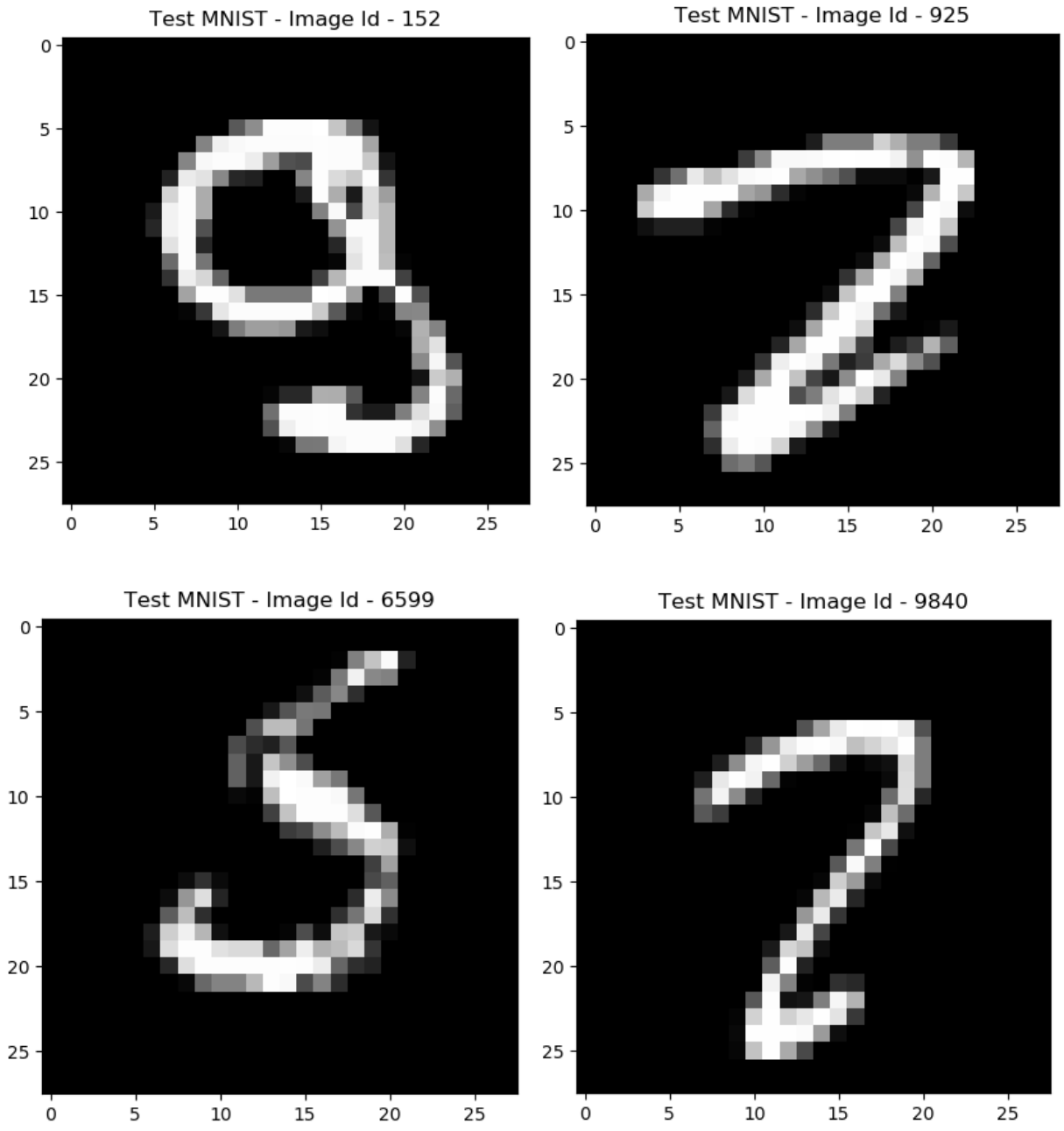
		Image Labels
CNN Misclassified – 38	CNN Misclassified, SAAK Predicted Correctly – 20 / 38 = **52.63%**	413 675 717 939 1404 1439 1460 1879 2036 2448 2455 2598 3535 3763 4861 5655 6559 8280 9643 9694
SAAK Misclassified – 168	SAAK Misclassified, CNN Predicted Correctly – 150 / 168 = **89.28%**	152 248 321 322 341 382 446 448 449 450 496 583 629 690 708 721 741 883 925 948 952 960 1045 1063 1113 1195 1227 1233 1243 1248 1261 1300 1320 1329 1501 1523 1531 1550 1554 1582 1682 1710 1755 1791 2045 2071 2099 2110 2136 2190 2294 2300 2372 2388 2407 2415 2423 2489 2608 2649 2772 2811 2864 2897 2928 2940 2954 2996 3061 3074 3118 3334 3370 3476 3491 3504 3521 3559 3560 3598 3619 3768 3777 3797 3809 3812 3854 3903 3942 3969 3986 4066 4076 4079 4164 4225 4249 4290 4307 4370 4498 4501 4640 4741 4808 4824 4880 4881 4887 5458 5601 5643 5735 5836 5937 5956 5973 5974 5983 5998 6036 6167 6506 6561 6572 6599 6652 7217 7435 8095 8326 8340 8409 9010 9016 9025 9588 9635 9665 9680 9699 9746 9750 9769 9771 9793 9809 9840 9923 9945
Both CNN and SAAK Misclassified – 18 (**0.18%** of test data)		660 1015 1040 1394 1902 2131 2183 2463 2655 3423 4177 4202 5938 6577 6598 6626 8528 9730

CASE 1 STUDY: - CNN Misclassified and SAAK Correctly Classifies:



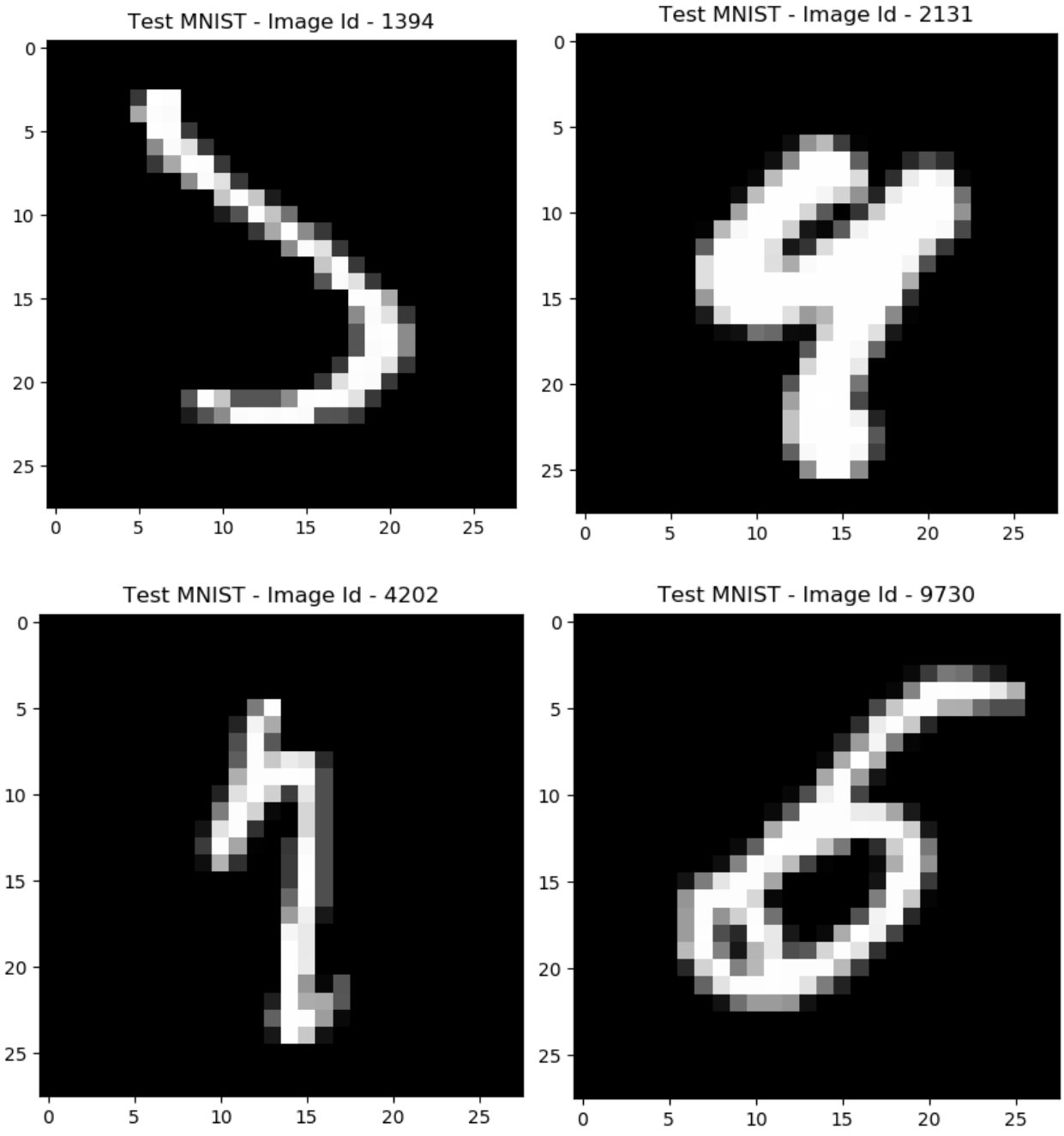
- The above images provided are examples of cases where the CNN misclassifies and Saak Transform exactly predicts correctly
- The above images look very distorted. Even for human eyes, it is very difficult to identify what the digit it is.
- Saak does a good job in identifying even the distorted images correctly while CNN obviously could not recognize since the features selected were bad. The training algorithm couldn't come up with good features.
- Saak Transform computes image features from using orthonormal kernels and it really helps in predict the features correctly

CASE 2 STUDY: - SAAK Misclassified and CNN Correctly Classifies:



- The above images provided are examples of cases where the SAAK misclassifies and CNN Transform exactly predicts correctly
- But if you closely look into the images, they are ambiguous. First image can be 8 or 9, second image can be 2 or 7, third image can be 3 or 9, fourth image can be 2 or 7
- So, I believe though SAAK does badly in such cases, it still predicts the closely resembling digit. Which can be acceptable. If these digits are human labeled, the actual labels itself might be wrong and the Saak Transform output may be right.
- These kinds of samples bring down the Saak Accuracy, but it should be acceptable since they are ambiguous to human eyes and there can be a debate on what is the actual output

CASE 3 STUDY: - Misclassified by both SAAK and CNN:



- The above images provided are examples of cases where the SAAK and CNN Transform both misclassifies
- For this case the images are again distorted like the previous case – 1. If these images are mis classified, I think that is okay and we don't have to worry too much
- May be to improve the accuracy we can preprocess it and reduce the distortion in the images. This could improve the performance and may be SAAK and CNN could predict the labels correctly
- Another way is to recheck the Actual labels if they were labelled by humans.

WAYS TO IMPROVE CNN OR SAAK TRANSFORM TECHNIQUES:

- As I showed different cases of misclassification happened, the main thing I noticed is that most of the mis-classified images are distorted. Reduction of distortion in the images could improve the performance of both the methods. This can be achieved by many ways of preprocessing. Better preprocessing techniques before using them in these models could improve the accuracy for sure. Some traditional techniques are given below:
    - Image Scaling
    - Image Normalization
    - Dimensionality Reduction
    - Per Channel Mean Subtraction
    - Data Augmentation

  From these techniques, we have actually used them in both the methods. And we have seen that they don't improve the performance very much. Hence, we can come up with different other ways of preprocessing that actually affects the performance improvement.
- To elaborate on Data Augmentation, we can come up with different versions of the image data we have like rotating the images, scaling the images etc. This can avoid the need of collecting more data and we can increase the volume of data with the already existing one.
- Another idea to improve the accuracy is to remove outliers in the images. For example, in the training data, we can consider images that are 7 labelled, and do some statistics based on spatial location. And if images which looks like 2 and are labelled as 7 can be considered as outliers and be removed from the data itself
- Since Saak Transform uses very less number of training data, this could potentially be the problem to reduce the accuracy. So, we can improve Saak Transform using more data and multi-scaling it.
- Saak Transform uses F-test score to select the important features from the available feature space. We can use some other advanced methods to select feature like Tukey's Honestly Significant Different Test (Tukey-Kramer Test), Newman-Keuls Test, Bonferroni's Test, Scheffe's Test, Dunn's Test etc.
- Maybe it is right time to recheck the Actual labels of the dataset. May be some images are labelled wrongly and we can manually relabel them as what to human maximally sees the digit as
- We can combine Saak Transformation model and CNN model and frame another Ensemble model which might improve the performance still more. Since these are the two techniques that lie on the top level for MNIST prediction, we can come up with combination of methods to get even more better output.
    - One way I could think of the combination as we can get the Saak Transform Coefficients and train few Fully Connected Layers (basically a Neural Network) to get best results instead of using SVM or Random Forest