EE 569: Digital Image Processing
Homework #5

By
Brinal Chimanlal Bheda
USC ID - 8470655102
bbheda@usc.edu
Spring 2019

Issue date: 03/20/2019
Submission Date: 04/07/2019

Problem 1: CNN Training and its Application to the MNIST Dataset

(a) CNN Architecture and Training

I. Abstract and Motivation

Foundation of Deep Learning is in Artificial Neural Networks (ANN) which came in 1980-1990s. There was some high-level approximation theory, which covered one input layer, one hidden layer and one output layer. It was a 2 layer connection. Most of the times ANN was served as a classifier. So basically first had to extract features to serve as the input to the classifier. But there were some competitive solutions like Support Vector Machine and Random Forest used for classifier as their performance was better for many applications. So neural network became not popular but later in 2012 it become a boom. However, now it is on another form one called as Convolutional Neural Network (CNN), famous in image processing and other is Recurrent Neural Network (RNN), used in speech and language processing.

In CNN, directly put the input data say, noisy images or cat images inside the network as it does the feature extraction and the classification and gives the output say, denoised images or labels. The difference between the recent research of the CNN compared to the previous way of ANN is the features. ANN only classifier and needs the human to do feature extraction and input it, while CNN does both the feature extraction as well as the classification as a whole. We use the softwares like Tensorflow which is a low level and Keras build on Tensorflow which is a high level language.

In modern Machine Learning, we have input data and output data which can be in the form of classes or data/images. And we have the ML model, which mainly does two things – feature extraction and classification/regression. Both the input and output are high dimensional vectors. ML wants flexibility and hence each pixel in the image is one dimension. Suppose have an RGB image with the width and height as Nx and Ny, it is a high dimensional vector in Nx.Ny.3 dimensional space. Also, can change the resolution for the width and height of the output image/data. For the class labels output, consider a vector depending on the number of classes, say have K classes so it forms k-dimensional vector and each value in it corresponds to the probability of each class and carries value as 0 and 1 (in training) and some decimal value lying in between 0 and 1 (in testing). This is the result of from input high dimensional vector to output high dimensional vector mapping. If there is only one 1 and others are 0 then it is called one hot vector. It is unit vector in the K-D space.

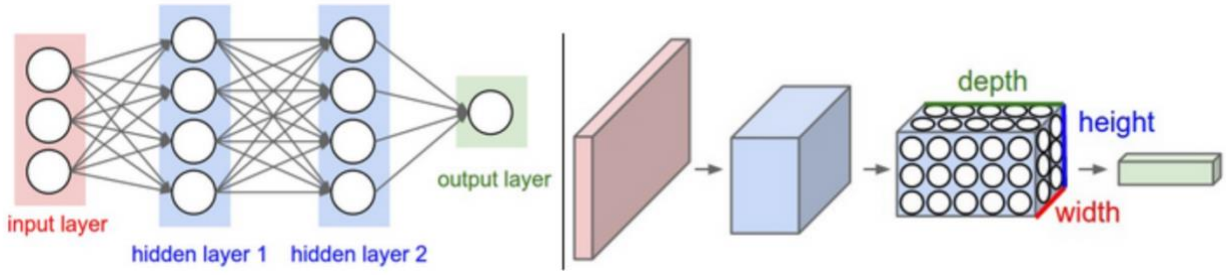
This is for the output in the training. For the testing this is different and we do not get exact 0 and 1 as solution, this is ideal and then we choose the maximum one, using softmax, which picks the highest probability between 0 and 1. For ML, all the multimedia data like speech, language, graph, point cloud convert to high dimensional vectors first and this process is called embedding. Training gives out input-output pair mapping and this method is called as supervised learning which includes the data and labels in the input. In unsupervised learning method, no labels are provided to the model.

Deep learning has very little inference power. It is smarter memory intensive system. It relies on the dense distribution in a small subspace in the input data space. If the testing data also goes to the small subspace area then the output would be good or else it would not. It works because of big data. Face recognition is one of the computer vision application examples which has been very successful. In CNN, we use joint spatial spectral representation (SSR). CNN finds application in image/video processing like restoration, denoising, segmentation and computer vision like object classification, object detection, object tracking, face recognition, 3D shape reconstruction, event detection.

CNN Basics:-

(Source: Stanford CS231n Course Instruction)

From ANN to CNN



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

The above diagram shows the transition from the ANN to CNN. We see that the height and width represent the spatial domain and the depth represents the spectral domain, also the spatial domain goes on decreasing in size and the spectral domain goes on increasing. Spatial falls down to only one and for the spectral, it will be the number of classes.

CNN works so well than the traditional methods like SIFT, Bag of words followed by classification techniques like SVM or RF. This is because it can find features systematically and automatically. In the past used to separate the feature extraction and classification or regression. For the feature extraction required a lot of domain knowledge. These were called handcrafted features brought by human ad hoc trials. CNN made this process automatic. Then measure the cross entropy and modify and get better output for lesser error. This is done for all the training images, input taken output got and compared and made better. This process is called one epoch and do back propagation for each batch in it. Keep repeating the procedure for each epochs. For MNIST usually require 20-30 epochs.

For the MNIST dataset, there are 60,000 hand written digit images of size 32x32, which is grayscale, monochrome i.e. 8-bit. This is for training, put the images one by one and then do the back propagation one by one. Now say all the images in the training set are set once, then this is called one epoch. Basically there is random initialization of filter weights. Then do the back propagation to adjust these filter weights. Now do one epoch, other epoch and after several epochs say 20-25. Go on doing until the error does not reduce more. Should come to be flat curve and we can say it has reached convergence. Once this level is reached, the training is completed. And the filter weights got can be used for testing. There are 10,000 test images. For 10 different random initializations, we get 10 different results. This happens because this is a non-convex problem where there are multiple minimum so filter can go to any place. The performance curve should not only show the best result, but it should show the mean of all the results of the times of trainings done. For multiple initializations, write the mean and variance.

MNIST problem is pretty small and hence can run on the CPU as well. Thus the environment is set. First build the software package in the environment and then specify the network architectures and then call the back propagation BP. There are 3 parts, data, features and decision, all are high dimensional vectors and the information is changing. However, CNN is not so robust.

CNN Training:-

Training data is called the labelled data samples. Testing data is data samples with the unknown labels for performance evaluation. But also need to know the ground truth to evaluate. In real world the dataset is divided into 3 set – training data, validation data and testing data. Validation is like your own testing data but not the

final testing data which will be compared to find the model responses. It has the ground truth. The real test data has images without the ground truths.

- How to train a CNN using the training data set?
  - Specify a network architecture
  - Initialize the filter weights
  - Search for the optimal filter weights to minimize a cost function

For the first point talk about the number of layers, specifications of the layers, fitler size. Last point suggests to do the backward propagation and a common example is use the cross entropy cost function.

(Source: https://www.youtube.com/watch?v=aVId8KMsdUU)
(Source: https://www.youtube.com/watch?v=zpykfC4VnpM#t=19.618193)

- Define the cost function
- Initialization
  - Random initialization is often used
  - Could be tricky since poor initialization may lead to vanishing gradients (i.e. the weights can be changed)
- Minimizing the error (or the cost function) by backpropagation
  - Basically, the chain rule in calculus
  - Stochastic gradient descent (SGC)

- Standard gradient descent algorithm

It updates parameters θ of the objective J(θ) as,

$$\theta=\theta-\alpha\ \nabla\theta\ E[J(\theta)]$$

where the expectation in the above equation is approximated by evaluating the cost and gradient over the full training set.

- Stochastic Gradient Descent (SGD)

It simply does away with the expectation in the update and computes the gradient of the parameters using only a few training examples (called a minibatch). The new update is given by,

$$\theta=\theta-\alpha\ \nabla\theta\ J(\theta;x(i),y(i))$$

where (x(i),y(i)) are from the same minbatch and α is the learning rate . A typical minibatch size is 256.

In standard gradient descent method, θ is the parameter set and take the gradient and cost function of the parameters and then the expectation. SGD method is also pretty similar but for the expectation, cannot do for only one batch. So use for few training example i.e. average of few number of sample called mini-batch. One key parameter is alpha which is the learning rate. The learning rate if larger then tend to oscillate more, if small then to be stable, but too small takes long to converge. So choose it wisely.

- Choosing the proper learning rate and schedule (i.e. changing the value of the learning rate as learning progresses) can be difficult
  - One standard method is to use a small enough constant learning rate that gives stable convergence in the initial epoch (full pass through the training set) or two of training and then halve the value of the learning rate as convergence slows down
  - Another approach is to evaluate a held out set after each epoch and anneal the learning rate when the change in objective between epochs is below a small threshold. This tends to give good convergence to a local optima
- Training data shuffling
  - If the data is given in some meaningful order, this can bias the gradient and lead to poor convergence. Generally a good method to avoid this is to randomly shuffle the data prior to each epoch of training.

Shuffle around the training data. Do not use the same arrangement of the data. Also there is momentum, this is where if we go to a very shallow region then need to change the direction. Thus momentum will help to change the search directions.

- If the objective has the form of a long shallow ravine leading to the optimum and steep walls on the sides, standard SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. The standard SGD can lead to very slow convergence particularly after the initial steep gains.
- Momentum is one method for pushing the objective more quickly along the shallow ravine.

The momentum update is as follows:

$v=\gamma v+\alpha \nabla\theta J(\theta;x(i),y(i))$
$\theta=\theta-v$

MNIST is basically object recognition problem and object detection is when there are multiple objects in the image.

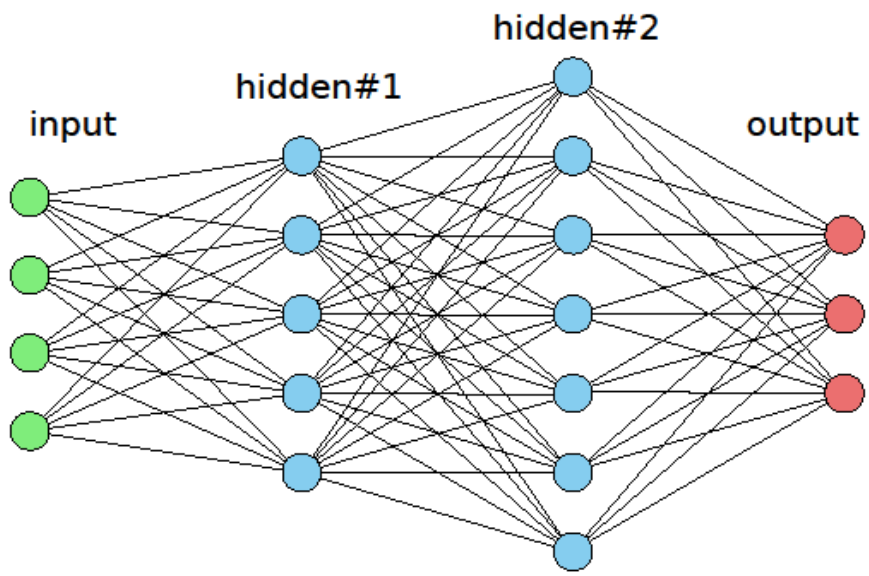
Why CNNs work so well? :-

From the figure above showing the convolution function and the non-linear activation, we see this is an affine system and input is x_i and the parameters are w_i and b. w_i is the weight of the correlation filter, b is bias and f is non-linear activation. For understanding the filter weights, (to find out how to interpret the filter weights). There are 3 angles or view-points:

- 1st viewpoint
  Parameters to optimize in large nonlinear networks, Backpropagation – SGD. Try to put a lot of input output pairs i.e. for an input image at the output there is a one hot-vector. Pixel images in pixel domain to labels in high dimension one hot vector. So find the pairs and then minimize the error. Randomly initialize the weight and put image in and get some output, may not be correct, there is some error, out it back in by back propagation. When try to propagate the error, try to reduce the error so called a steep descent, to lower the cost and called as gradient descent. This is the back-propagation algorithm. Also called as stochastic gradient descent algorithm. It is used for large non convex optimization problem. Non convex means have many local minimum. So it highly depends on the minimum. However, the final performance comes close to each other for different initial weights chosen. The local minimum gives the cost pretty low. All the local minimum points form a narrow band.
- 2nd viewpoint
  Consider here the matched filters. Design different kinds of patch, this patch is a certain pattern and can design some pattern. And view with the matched filters and try to find what is the frequently occurred patch. To do this can use k-means clustering. Say suppose the input is 5x5 patch, then cluster them into k clusters and then use the centroid of each cluster as the matched filter. Collect the input and try to cluster them into k clusters and then each centroid use as a filter. This is using k means based filter design. It is classified to find the best match.
- 3rd viewpoint
  Bases (or kernels) for a linear space, Subspace approximation.
  Do not consider independent filters. Consider them as working together. For the filter, have a kernel, eigen vector forms a space and consider to find the

space which gives the best approximation of the original signal. The subspace approximation actually suggests the PCA.

Multi-layer perceptron, supervised learning by backpropagation (BP)



All the nodes above are full connected. Input is n dimensional feature vector such that one feature per node. This is similar to ANN. Can also use SVM and RF. These are much better than LDA.

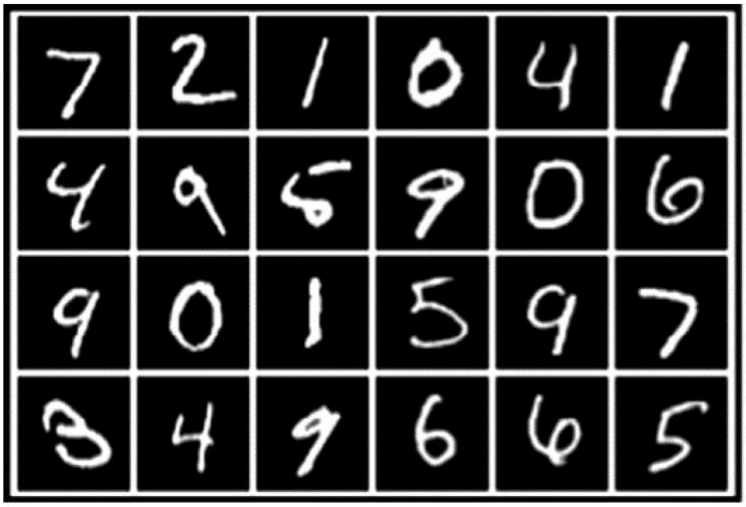
All the learned knowledge is put on the filter weights. All training/learning results are summarized in filter weights. Filter weights play a critical role in understanding CNN. Each convolutional or fully connected layer defines a transform matrix.

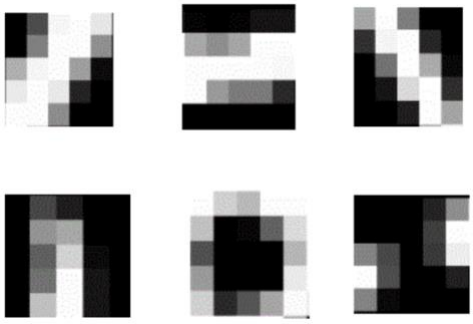
Multi-layer Approach

- There are multiple transform matrices
- Learning **A's** from a class of signals and their decision labels (**d**)
- Feed an instance of **x** into the network for its decision **d**
- Need a nonlinear activation between layers

For convolution as a matched filter. A convolution operation can be viewed as the inner product to two vectors. Interpreted as “correlation”. Filter Weights are fixed in the test stage, called anchor vectors.

Say for the MNIST dataset, take the following 6 representative patterns.





All above are the 5x5 patches and choose the above different patterns and then scan through the whole input for this and this will give some information about the pattern. The above 6 are got through the k means. Collect a lot of these 5x5 and do the k means clustering. And get the k clusters and the filter. Now use these filters to scan through the image and find the response.

Overfitting:
There is a performance gap between the training and testing. Usually the graph of epoch vs error is in a decreasing fashion. For more epochs, it goes lower and lower, this is for training. When using for the testing data, it almost follows same as training. However, at a certain point the testing curve becomes either flat or starts going up. So the model seems good for training data but for testing data does not

appear good. This means that the training data is used too much, so that the system is not general anymore. The model is too specialized such that it does not fit the testing data. So this is called overfitting. This occurs because the parameter number is too high, much greater than the number of training samples. Thus if the parameter of the model is too big, it has the potential or risk to overfitting.
The way to avoid overfitting is using dropout. This means when updating the weights, don't update all the weights. Update some weights and some weights assume that let the connections off for the back propagation. So when doing BP, drop out or eliminate some nodes. Dropout says to kill some nodes. This doesn't matter to eliminate some nodes because on average, so update the weight correspondingly. Next iteration change for other node locations and update the link. At one moment, say drop off 50% of the link so basically reduce the parameters by half. So whenever fitting actually fitting the model with half size. So making the parameter part smaller. But the weight is more like averaged along the time. So once if no update, next time it is updated. Each time only update 50% of them. This is random choose the weight update.

CNNs work much better than other traditional methods in many computer vision problems mainly because it also does the feature extraction part. CNN faces architecture as well as parameter issues.

For part (b), show multiple initializations. Different initializations give different results. There is training accuracy and testing accuracy. In training accuracy, have initialization and then test the performance wrt the training data. And then the model is applied to calculate the testing accuracy. The training takes barely half an hour so can try multiple initializations. Take the mean and variance of the final result of accuracy percentage. 10 different models each training and testing and then compare the accuracies obtained.

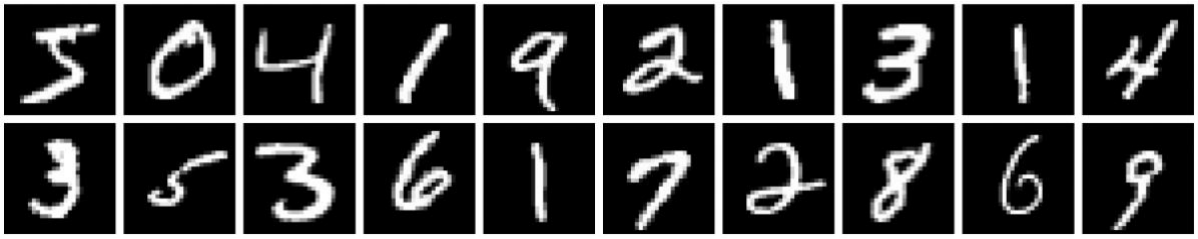
II. Approach and Procedure

(Source: Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, 86(11):2278-2324, November 1998)

(Source: http://yann.lecun.com/exdb/mnist/)

Train one simple convolutional neural network (CNN) derived from the LeNet-5 introduced by LeCun et al. Furthermore, apply it to the MNIST dataset. The MNIST dataset of handwritten digits, has a training set of 60,000 examples, and a test set of

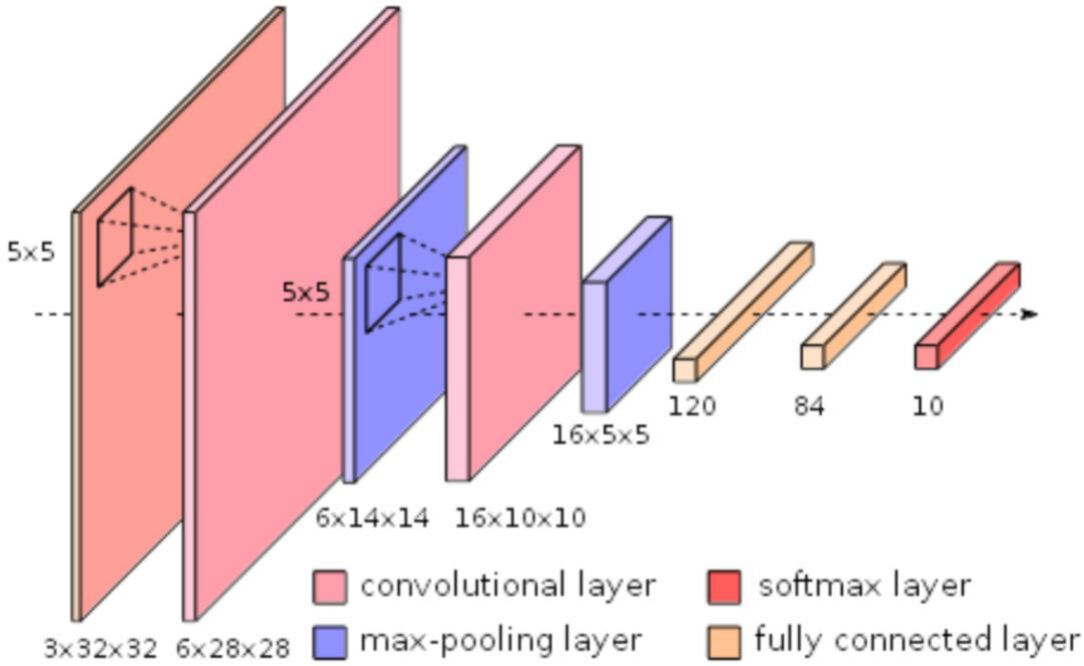
10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. Figure below shows some exemplary images from the MNIST dataset.



MNIST dataset

(Source: https://en.wikipedia.org/wiki/Rectifier_(neural_networks))

(Source: https://en.wikipedia.org/wiki/Softmax_function)



CNN architecture derived from LeNet-5

The CNN architecture for this assignment is given in the figure above. This network has two *conv* layers, and three *fc* layers. Each *conv* layer is followed by a *max pooling* layer. Both *conv* layers accept an input receptive field of spatial size *5x5*. The filter numbers of the first and the second *conv* layers are 6 and 16 respectively. The stride parameter is 1 and no padding is used. The two *max pooling* layers take an input window size of *2x2*, reduce the window size to *1x1* by choosing the maximum value of the four responses. The first two *fc* layers have 120 and 80 filters,

respectively. The last fc layer, the output layer, has size of 10 to match the number of object classes in the MNIST dataset. Use the popular ReLU activation function for all *conv* and all *fc* layers except for the output layer, which uses softmax to compute the probabilities.

Explain the architecture and operational mechanism of convolutional neural networks by performing the following tasks.

- Describe CNN components in your own words: 1) the fully connected layer, 2) the convolutional layer, 3) the max pooling layer, 4) the activation function, and 5) the softmax function. What are the functions of these components?
- What is the over-fitting issue in model learning? Explain any technique that has been used in CNN training to avoid the over-fitting.
- Why CNNs work much better than other traditional methods in many computer vision problems? We can use the image classification problem as an example to elaborate the points.
- Explain the loss function and the classical backpropagation (BP) optimization procedure to train such a convolutional neural network.

III. Discussion

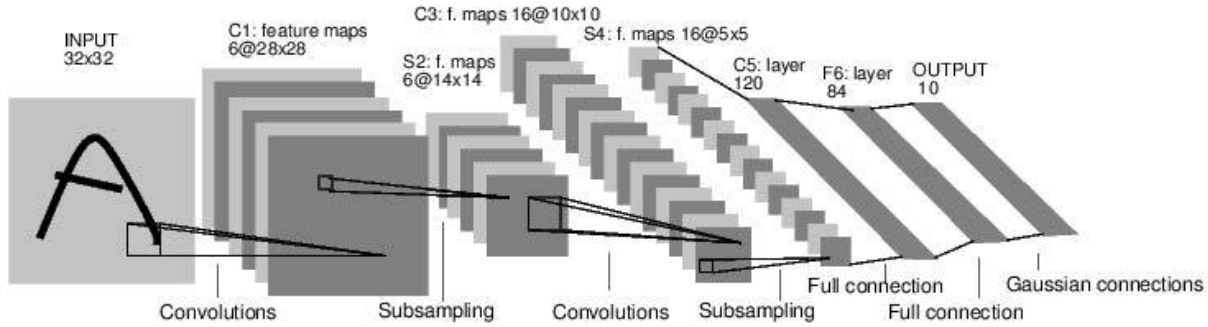
(Source: http://cs231n.github.io/convolutional-networks/#comp)

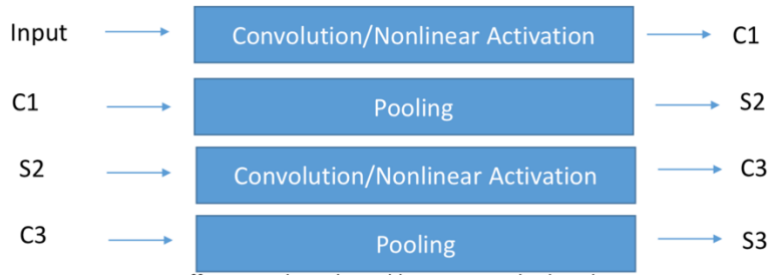
Convolutional Neural Networks are also called as CNN or ConvNet. These networks consist of neurons which have learnable weights and bias. They have been compared with the neurons of the human brain. These neurons receive the inputs and perform dot product between the weight and input and adds the bias plus follows it with a non-linearity. This network architecture is made up of a sequence of layers like the convolutional layer, the max pooling layer, the fully connected layer. Every layer transforms one volume of activations to the other one using differential function. It also consists of the softmax function as well as the activation function. CNN architecture assumes that the input is in the form of images and this allows to encode certain properties in the network. This helps to reduce the number of parameters used as well as make the forward function process very efficient. Each layer of the ConvNet has a 3D input depending upon the type of layer and number of filters and gives a 3D output. Each layer has it's own set of parameters and hyperparameters which we need to set or adjust and try to get the best accuracy for different settings.

- The different CNN components are described as follows:

(Source: LeCun, Bottou, Bengio, Haffner, Gradient-based learning applied to document recognition, Proc. IEEE, 1998)

LeNet-5





LeCun introduced the LeNet-5 model in 1998. The 5 in LeNet-5, LeCun network stands for the 5 layers – 2 convolutional and 3 fully connected layers. Convolutional layer is partial, it is region based and does not connect to all the nodes. Fully connected layers connect to all the nodes from input to output. In convolution, only last few layers are fully connected which are used for classification like in ANN and the ones before i.e. partially connected are used for feature extraction. The algorithm for ANN is called backward propagation, where define the cost function need to optimize. This process is a stochastic gradient descent to lower the minimal point as much as we can until get local or global minimum. However, the convergence curve is oscillating. We have a network model with input and output of high dimensional vector and we define the cost function and when training the graph of the cost function should go down. However, the curve is oscillating, which is stochastic in nature, called as stochastic gradient descent. To do this use the chain rule in calculus to derive the back propagation. Apply to a batch say of 1 image frame. Try to optimize the system by learning the weights most suitable to get the convergence.

2)The convolutional layer –

This is the most important part of the CNN. It performs very heavy computations. Now combine the C1 and S2 parts, this is actually one layer. C1 is basically consisting of 6 filters and each filter is a weight mask. So have a 5x5 weight mask, defined by it's mask weight. This 5x5 mask is applied to all the positions of the 32x32 input image. But no need of doing boundary extension. So start from location (3,3). The stride i.e. the number to move right or down is 1. This gives the response which is the weighted sum of the 5x5 filter mask multiplied by the 5x5 pixels. Do the weighted sum and give the value to the center. For each filter, we get 28x28 response because of the boundary effect. 28x28 response comes in spatial location and then 6 filters so 6 response in the spectral domain. The next step is pooling or subsampling. However, do not use the mean, for every 2x2 location, choose the largest one so called maximum pooling. Also this is non-overlapping. Thus after the pooling, get the 28x28 to 14x14 response, because after do the pooling will partition 28x28 to 14x14 non overlapping blocks. Do this for each filter, doing the spatial pooling. This convolution and pooling parts are called as one layer which takes input 32x32 and gives the output 14x14x6. Each filter is a spectral component here the number 6 denotes that. Thus the output obtained is called a tensor.

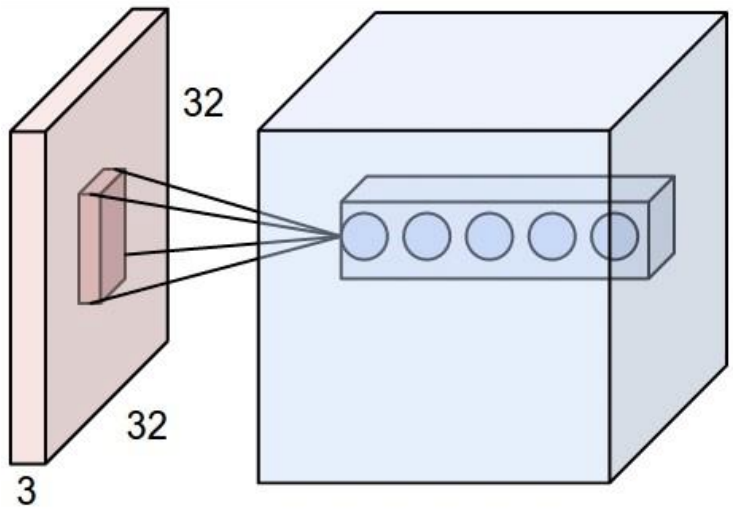
At the second stage again the operation is performed in the spatial and spectral domain. The second stage filter is of dimension 5x5x6 like a cuboid. The number of filters is 16. Consider the point wise multiplication for the filter weight and the data. This whole will give one response which put in center. Again start from the position of (3,3) because do not want boundary extension and go through the input so total positions get is 10x10. Operate this and get the 10x10 output and then each filter gives one slice. So 16 filters give 16 slices. So the original response got is 10x10x16 and then after pooling get 5x5x16 response. This is the second stage of convolution layer. The filter size is a small set of the input.

1)The fully connected layer –

Now for the third stage, we have the fully connected layer. So the input is the 5x5x16 and we have 120 filters, each of the shape of 5x5x16. The filter is same size as input and determine the weighted co-efficients and do the point wise multiplication which eventually gives one response. So the whole filter weight done on the data gives one response and since 120 filters get 120 responses because of 120 filters. The next stage each filter is of size 1x120 and then multiply the data and get one value and there are 84 filters each giving one value so 84 responses. Finally the final stage, again for fully connected layer, the filter is the same size as the input and get the 10 filter data of each having one value response.

2)The convolutional layer –

Convolution with the filter banks

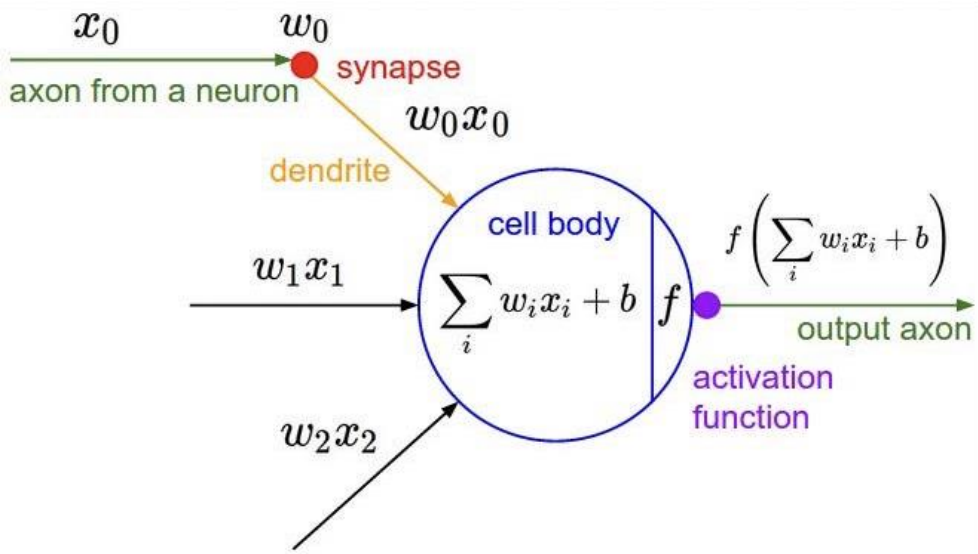


If the input size is 32x32 and there are 3 RGB channels, then the image dimension is 1024x3. But for the convolution layer, the window size is 5x5x3 i.e. 75 dimension and try to design a filter, as in weighted vector filter weight to get the response. The depth of the layer depends on the number of filters each corresponding to one response. For the spatial, it corresponds to the location. Learn the filter, and weight depending on the training data.

Here the input is small color image of size 32x32x3, called as tiny images. 3 denotes the color image. The filter size is a partial size of the input. The filter is like convolution kernel. The operation is actually similar to the correlation of the two matching filter and input data named as the convolution operation.

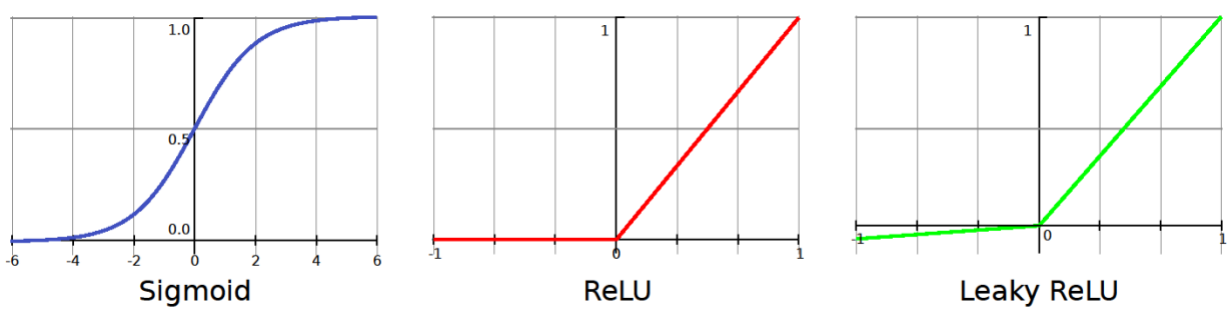
Convolution + Nonlinear Activation

4)The activation function –



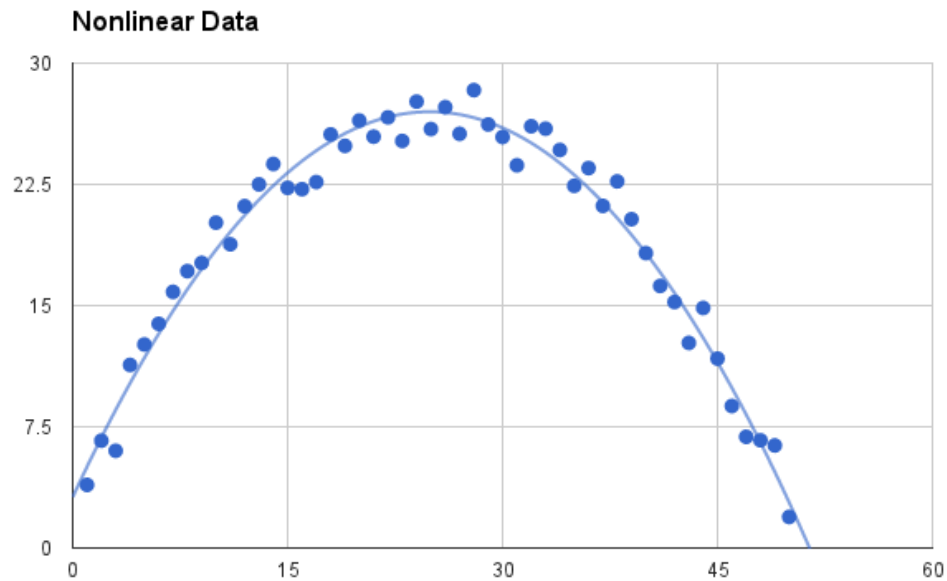
First do the weighted sum, say from w0 to w1, weights are denoted by w_i. Adjust the weights until it is converged. So do the weighted sum plus the bias term, to get the response which goes through a certain activation function. This activation function is called the sigmoid, relu and so on. This is quite similar to the human neuron.

This weighted sum multiplication plus bias is called affine transform rather than linear transform. The f is called the non linear activation.



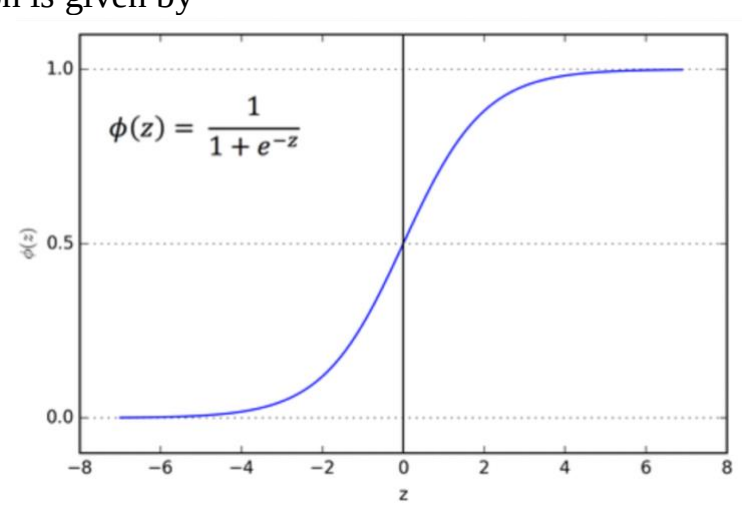
ReLU stands for rectified linear unit. The most commonly used non linear activation is called ReLU. For the sigmoid, if have any negative, is actually 0 there is a transition from 0 to 0.5 and for positive is actually 1 and there is a transition from 0.5 to 1. For ReLU, linear unit because can see on the positive side, a linear function and rectified stands for the negative part which is 0. Also there is leaky ReLU similar to ReLU, but for the negative side there are some negative values.

There are two types of activation functions at large – linear and non-linear. Different functions plot different graphs between -1 and 1 or 0 and 1.



(Source: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6)

Sigmoid function is given by

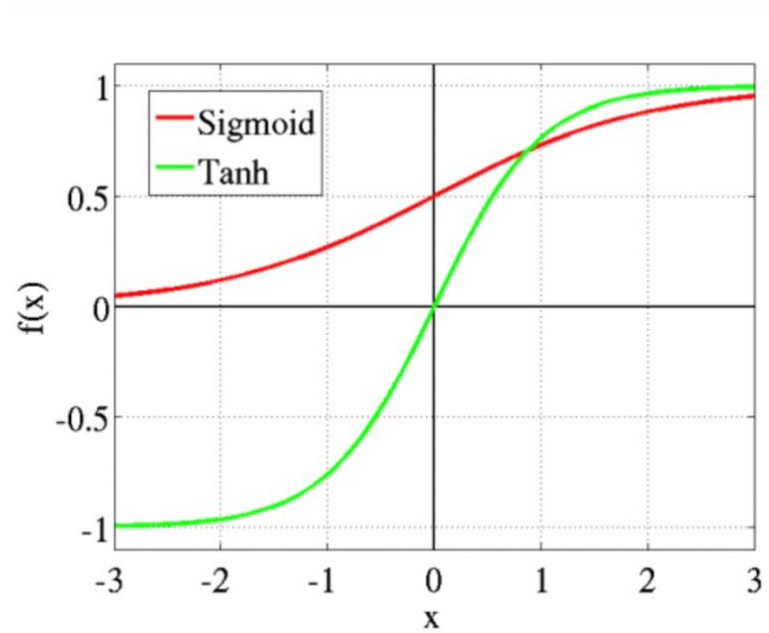


It is used in models where need to predict the probability as the output. This function is monotonic.

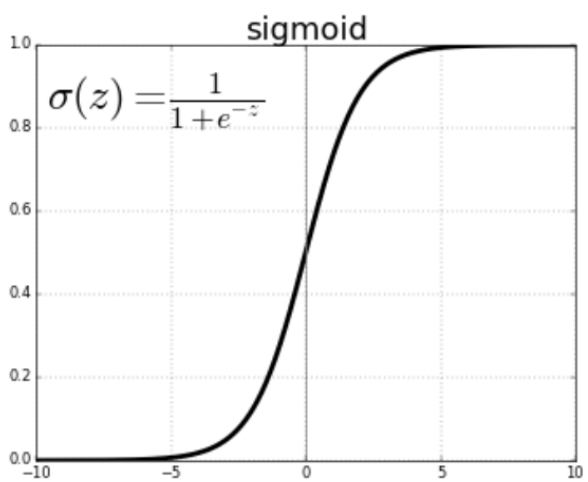
Tanh function is given by

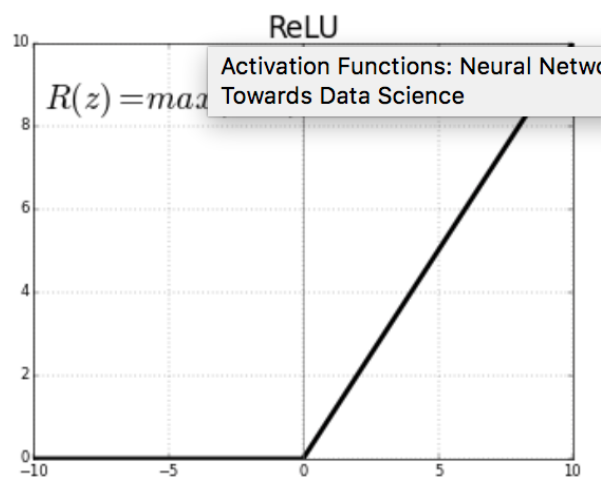
$\tanh(x)=2\,\sigma(2x) - 1$

Its range is from -1 to 1. This function is also sigmoidal and differentiable and monotonic. Its mainly used for the classification between 2 classes.

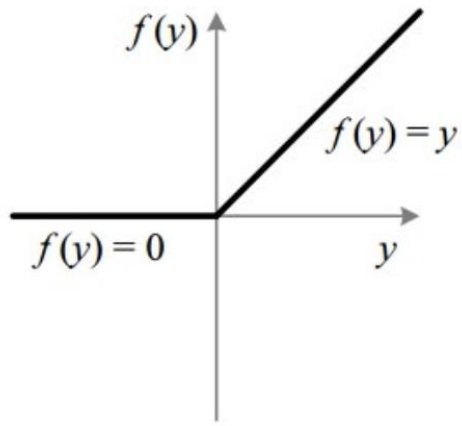


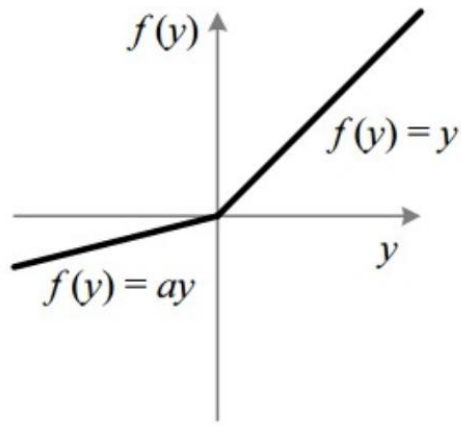
Relu (Rectified Linear Unit) function is given by
f(x)=max(0,x)
This is the most commonly used function in real world applications. Its range if from 0 to infinity. This function is monotonic.





Leaky relu is given by

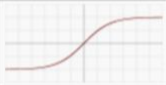
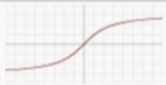




Its range is from minus infinity to infinity. There is a leak on the negative side. The slope takes negative values unlike in relu where it is assigned 0. This function is also monotonic.

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

Below is the table which shows the various activation functions along with their plot and equations:

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

It is important to know the operation and filter weight are given for the network. This is for the testing; training is where filter weights are frozen and testing is the one where filter weights are frozen. Thus the last layer gives the values which are converted into values for probability distribution. Thus for the above architecture there are 10 classes and get 10 probability distributions and then choose the response giving the highest probability. So for testing just go through the whole operation shown above. For training, find the different pairs of the input-output mapping finding the one hot vector. Also choose the filter weights by the method of back propagation.

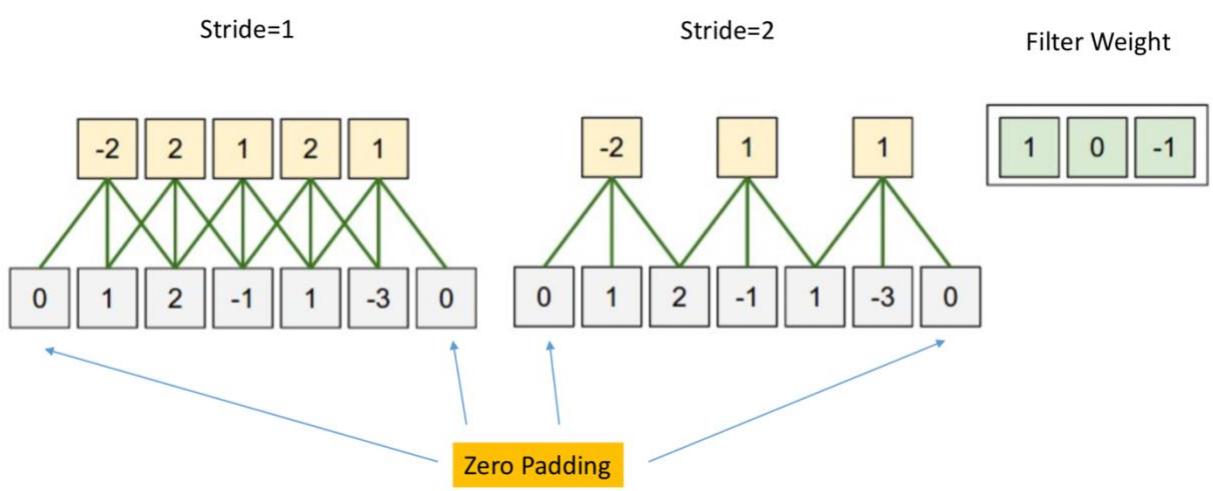
2)The convolutional layer –

Parameters in Conv Layer

For the given example

- Filter size: 5x5x3
  - 5x5: the receptive filter size
  - 3: the input filter channel size
- No. of filter weights: (5x5x3)+1=76
  - 1 denotes the bias term (b)
- No. of output filter numbers (channels):
  - 5 output channels are given in the example
- Filterbank: all output filters form a filter bank
  - Each filterbank has a center
  - The output is written to the corresponding center location
- Stride: the distance to slide the center of a filter bank
- Zero padding: pad zeros around the border region

Stride for larger images is chosen to be 2 or 3, because resolution too high so can skip. But for smaller images, stride is 1 to cover all the information. There are some stride examples given below.



The following parameters are given: input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. Using these we can compute the spatial size of the output. The formula for computing how many neurons fit is stated below:

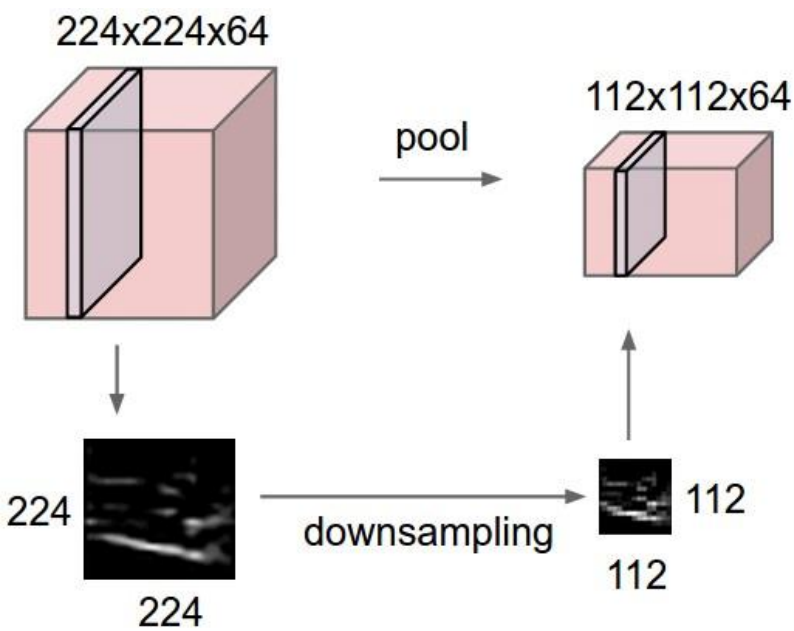
$(W-F+2P)/S+1$

The convolutional layer is explained below:

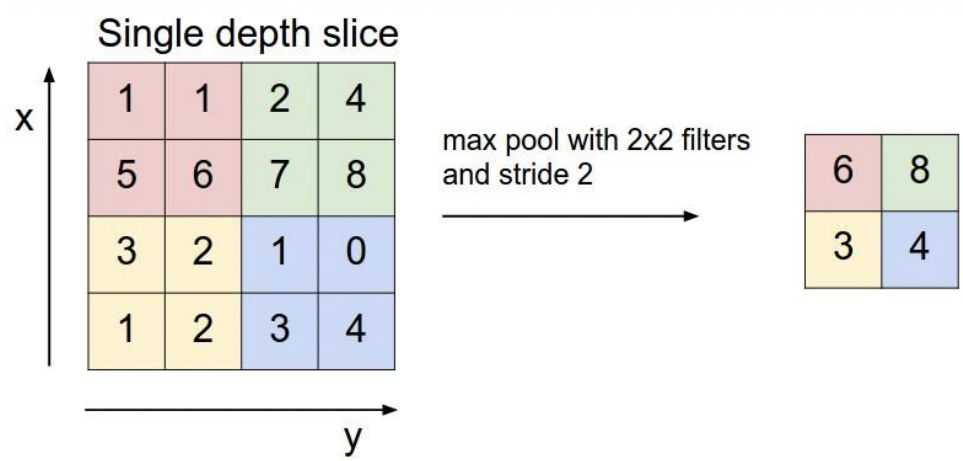
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

3)The max pooling layer –

Pooling (or down sampling or subsampling or spatial pooling)



We do not do pooling in the spectral, only in the spatial domain it is performed. Above we see that 224x224 becomes 112x112. It becomes one half of each dimension. There are different types of pooling – Average pooling, max pooling or min pooling. To do this, we use maximum pooling as below.



For every 2x2 block in the spatial domain, choose the maximum value. Repeat this for each filter spatially take 2x2 blocks and replace by one maximum value. We do not do average as sometimes linear is not good in adapting and learning. So we use non-linear. Maximum pooling performance is much better than non-linear pooling. Thus, choose the point which has maximum representation capability.

Max pooling layer is added post the convolutional layer along with the actuvation function like relu, sigmoid, tanh, leaky relu. It reduces the number of parameters to be trained by reducing the size of the spatiatl representation, thus it also helps to prevent or reduce overfitting of the network.

The pooling layer can be explained below:

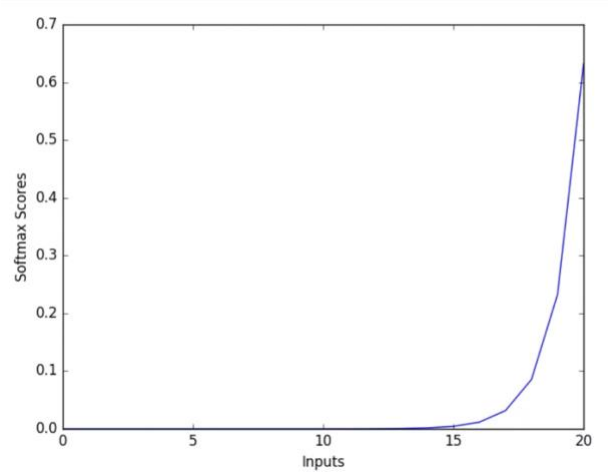
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- For Pooling layers, it is not common to pad the input using zero-padding.

(Source: https://en.wikipedia.org/wiki/Softmax_function)

5)The softmax function –

This is also known as the normalized exponential function. It is a function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities. It is used in CNN to map the non-normalized output of the network model to probability distribution of the output classes predicted.
The formula of softmax is given below:

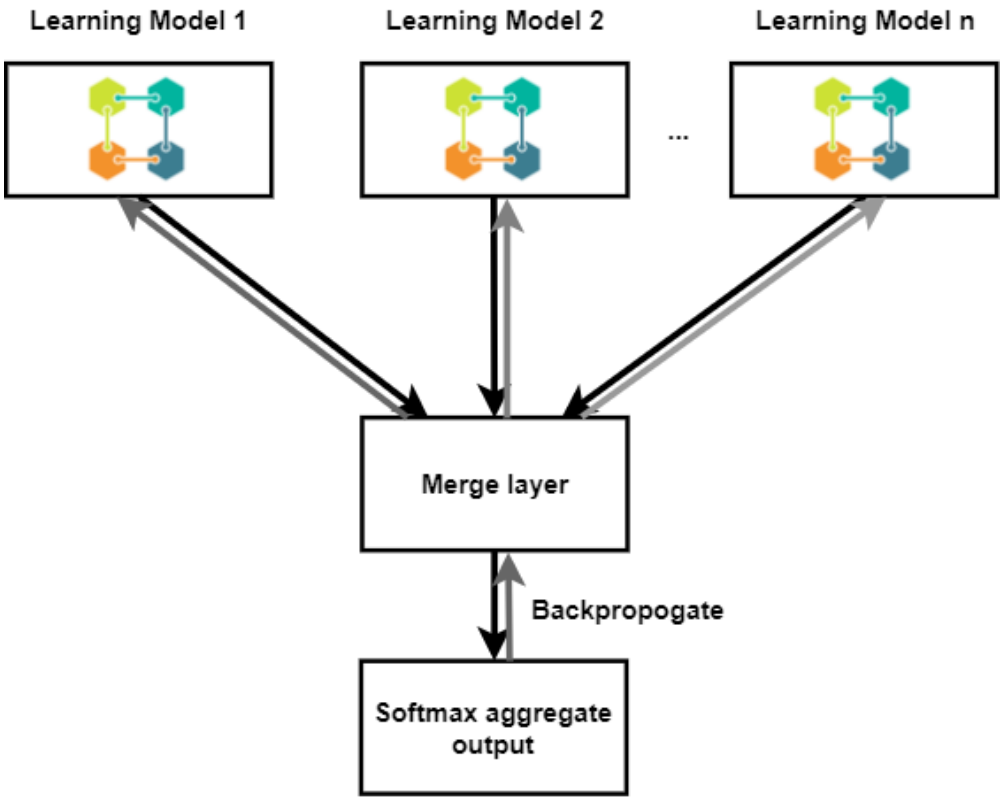
$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$



It is usually used in the final layer of the neural network classifier. It is present in the last dense layer to produce the output. This function is a more generalized logistic activation function, it is used for multiclass classification.

(Source: https://towardsdatascience.com/the-softmax-function-neural-net-outputs-as-probabilities-and-ensemble-classifiers-9bd94d75932)

Range is [0,1]. Maps the output in such a way that the total sum is 1. It is used for the multi-classification in logistic regression model, where the sum of probabilities is one. The softmax can be used for any number of classes. Say for example, can be used for the object recognition problem where there are different number of objects in the images data.



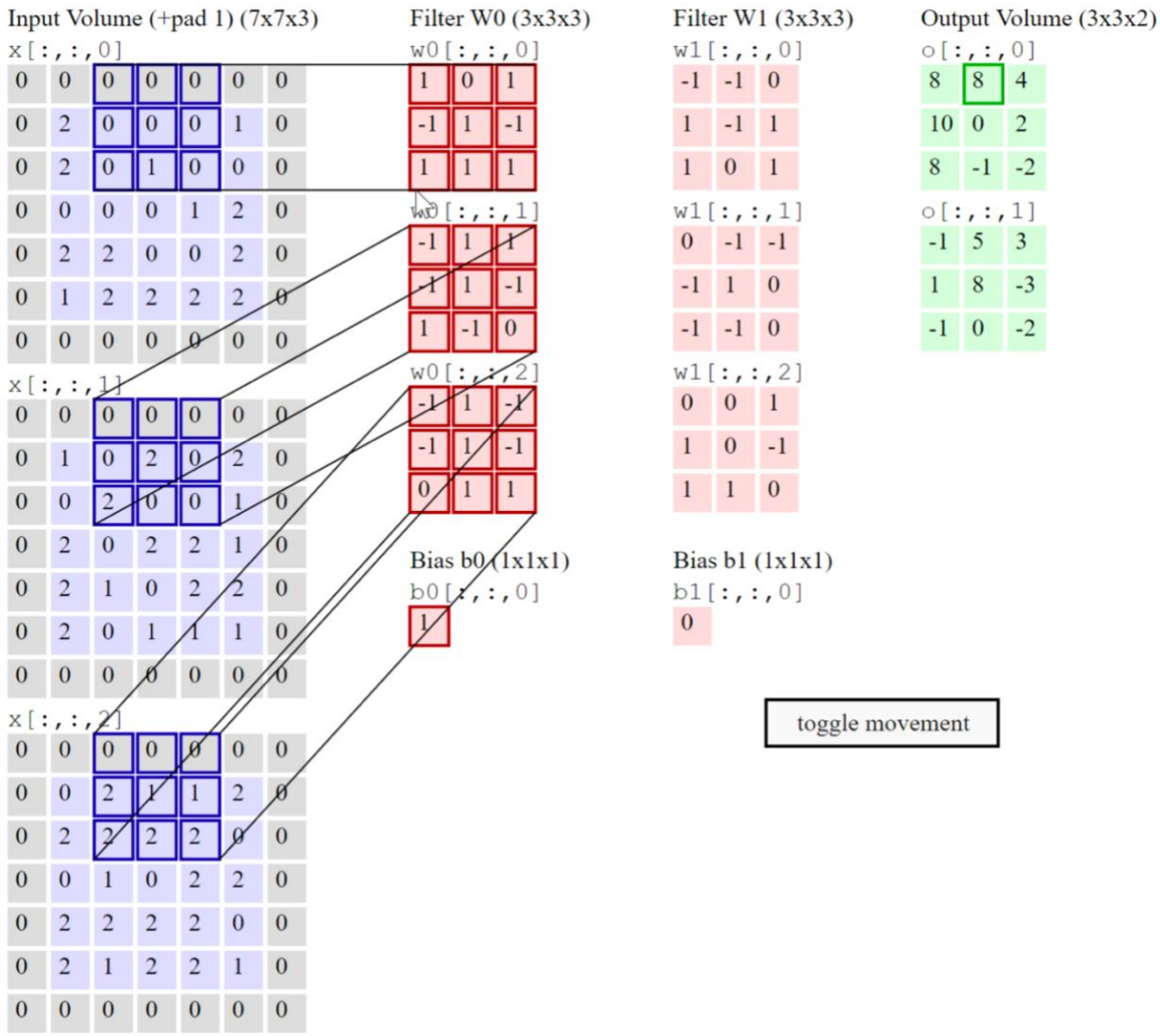
Output layer

- A 10-D probability vector
- The desired output
  - If the input is “0”, set the desired output to [1,0,0,…,0]^T
  - If the input is “1”, set the desired output to [0,1,0,…,0]^T
  - If the input is “2”, set the desired output to [0,0,1,…,0]^T

  etc.
- If the output of a training sample is V with ground-truth of digit k, we can define the cost function as the Euclidean distance between V and its corresponding unit vector
- If the output of a testing sample is W whose length is normalized to one, we choose its element that has the highest probability and assign it to its associated digit
  - Example: W = [0, 0, 0.65, 0.1, 0, 0, 0, 0.2, 0, 0.05], then the answer is “2”.

2)The convolutional layer –

Function of the Convolution layer:



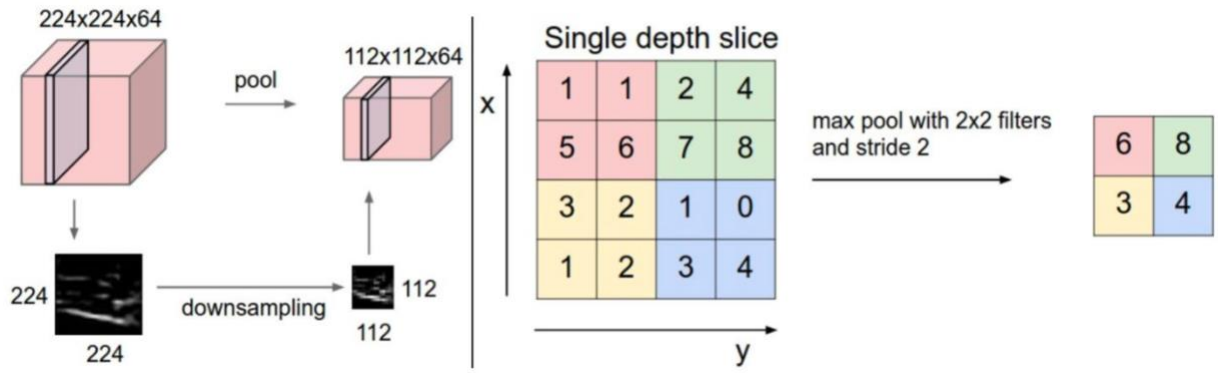
We have a 3x3 filter size and for each of the output map we got 3x3x3 rows and columns. So here the filter rate is 3x3x3. So we perform convolution with the input i.e. basically the dot products. So we compute the dot products, adding them up and end up with a result in the feature mapping at the output. So for 2 set of weights, we get 2 set of activation maps and we stack them together and that's the output. The stride i.e. step size we use is 1 which is similar to the image filter. There is no padding used. Thus the output size will shrink. Because we do not have padding for the boundary pixels, where we used to center at the boundary pixels when doing the convolution.

Say, taking a Numpy example. We have an input X (11, 11, 4). This is the shape of the input and if the filter size = 5 and stride is 2, we get the spatial size of the output as 4, giving a volume with width and height as 4. Also the output volume depends

on the number of the filters used. The operation is as follows. For the first elements in the outputs, say (0,0) elements in f, we basically take the first 5x5 block in the input and then we do the dot product with the filter weight plus the bias so as to get the final output. For the other parts, repeat the steps similarly, we move the filter down and start with 2 as stride is 2. Suppose we have 2 filters, doing the above procedure by sliding each filter across the input image, we get one activation map for the output column. After this, we consider another filter with a different weight and basically we repeat the same operation and get the second activation map for the output. This is what the conv layer is doing.

3)The max pooling layer –

Function of the Pooling layer:

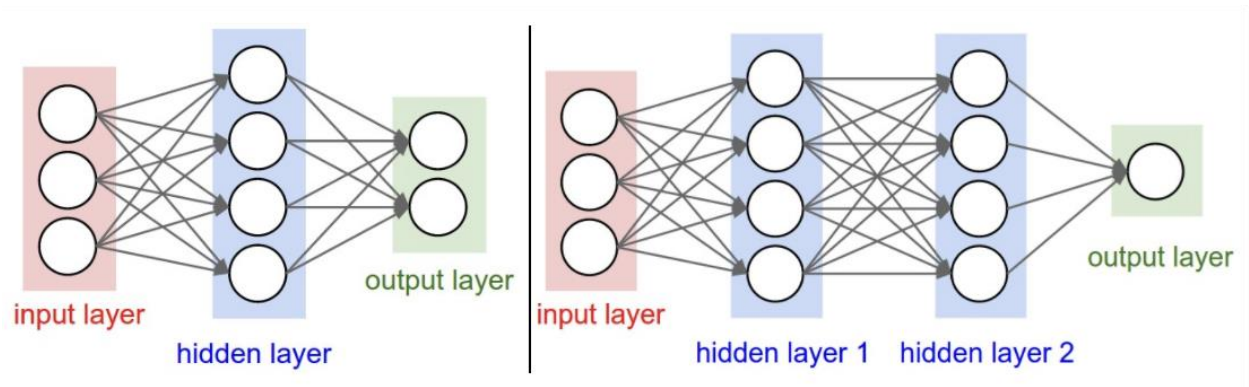


Basically for max pooling, we are doing that independently along each depth dimension. We are doing that spatially. So we are downsizing each one here spatially. And we keep the depth dimension here. So for each spatial size, for each depth size, suppose we are doing 2x2 filtering with stride 2 here, so we take the 2x2 window. If we are doing the max pooling, we take the maximum value here and putting it into output.

For the pooling layer, here we are downsampling the volume. So we are downsampling in the spatial dimension. We are doing this independently for each depth size. So for each depth size, for example if we do max pooling, wherein we take the maximum value and put it in the output. And this is basically the operation we have for the pooling layer.

1)The fully connected layer –

The fully connected (FC) layer is quite similar to the one we have in neural networks. In the fully connected neural network, all the nodes are connected to the next and previous layer. This is a dense layer which has all the nodes connected to every other node in the next layer. Below we see a one layer and two layers neural networks. Also, all the neurons from the previous layer are connected to the present layer. This is usually among the last few layers in the network at the end. Also it has various hyperparameters like the other layers. The activation function is one among them, which usually for the last layer in the network is chosen to be the softmax function.



The functions of these layers were explained above.

- The over-fitting issue in model learning is stated as follows.

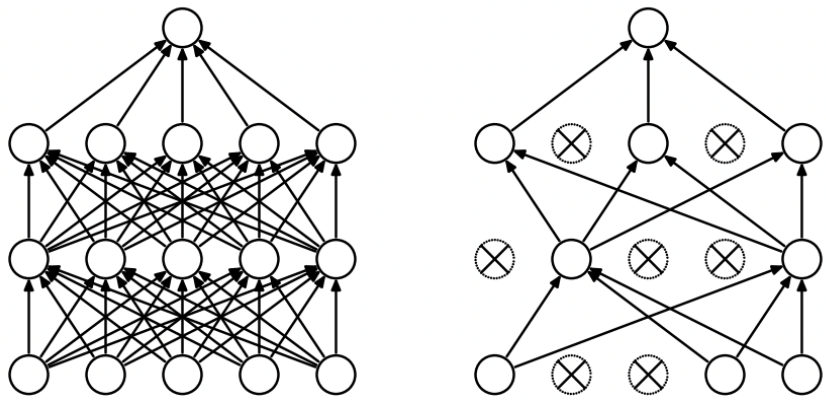
Overfitting:
There is a performance gap between the training and testing. Usually the graph of epoch vs error is in a decreasing fashion. For more epochs, it goes lower and lower, this is for training. When using for the testing data, it almost follows same as training. However, at a certain point the testing curve becomes either flat or starts going up. So the model seems good for training data but for testing data does not appear good. This means that the training data is used too much, so that the system is not general anymore. The model is too specialized such that it does not fit the testing data. So this is called overfitting. This occurs because the parameter number is too high, much greater than the number of training samples. Thus if the parameter of the model is too big, it has the potential or risk to overfitting.

The technique that has been used in CNN training to avoid the over-fitting is as follows:
The way to avoid overfitting is using dropout. This means when updating the weights, don’t update all the weights. Update some weights and some weights

assume that let the connections off for the back propagation. So when doing BP, drop out or eliminate some nodes. Dropout says to kill some nodes. This doesn't matter to eliminate some nodes because on average, so update the weight correspondingly. Next iteration change for other node locations and update the link. At one moment, say drop off 50% of the link so basically reduce the parameters by half. So whenever fitting actually fitting the model with half size. So making the parameter part smaller. But the weight is more like averaged along the time. So once if no update, next time it is updated. Each time only update 50% of them. This is random choose the weight update.

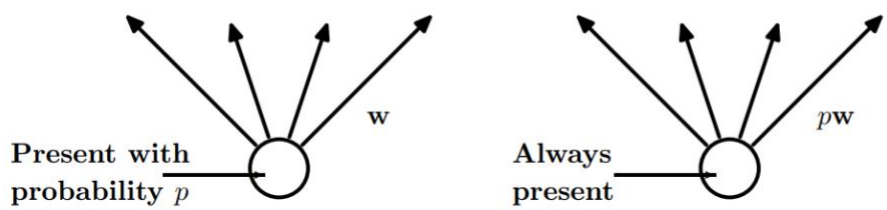
There are two concepts in machine learning – underfitting and overfitting. If the accuracy of the validation set is higher than that of the testing set then the model is called underfitting. The model performs very poorly and thus this is the concept of underfitting. However, for overfitting, this happens when the model fits too well on the training set. The training set provided is learned by the model in a too good way, such that when the testing data is provided, it cannot generalize and the testing accuracy comes out to be poor in value. Here, the accuracy of the training is very high as compared to the accuracy of the validation set, thus the model is called overfitting.

The techniques that can been used in CNN training to avoid the over-fitting are as follows:
One method used to control overfitting is using the pooling layer.
Another idea is to add more data. This is done to increase the variations in the training input provided to the network. It helps to reduce overfitting. However, this is a very costly and abrupt solution which is not used in real time as it becomes difficult to collect more data.
Also, can implement the idea of data augmentation. This provides the same image samples in the training data but with more number of variations to it, thus adding to the count in the training set.
Sometimes, simply trying different architectures will help to generalize the training data very well depending upon the type of data and application, which leads to getting a higher accuracy for the testing as well as validation data sets.
One of the most suitable solution to do is to use regularization. There are different methods in regularization like using dropouts, L1 regularization and L2 regularization. Dropout:



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.



(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights **w**. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

The explanation for this is mentioned above.
(Source: https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c)
L1 regularization also called as Lasso regression.

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

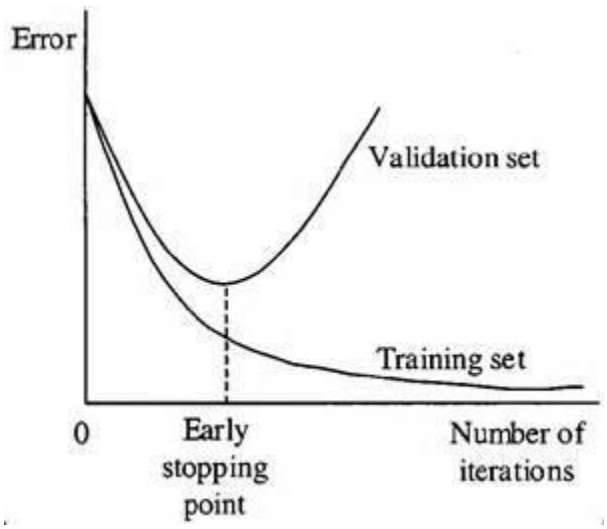
Cost function

L2 regularization also called as Ridge regression.

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

Cost function

These both work well for the feature selection, in cases where we have a large number of features. Also we need to decrease the network complexity.
Another solution is to stop the training early before the overfitting occurs, by calculating the performance of every iteration.



- CNNs work much better than other traditional methods in many computer vision problems. The reason for this mainly is that they perform the feature extraction as well as the classification part whereas traditionally only classification was done automatically.

CNN works so well than the traditional methods like SIFT, Bag of words followed by classification techniques like SVM or RF. This is because it can find features systematically and automatically. In the past used to separate the feature extraction and classification or regression. For the feature extraction required a lot of domain knowledge. These were called handcrafted features brought by human ad hoc trials. CNN made this process automatic. Then measure the cross entropy and modify and get better output for lesser error. This is done for all the training images, input taken output got and compared and made better. This process is called one epoch and do back propagation for each batch in it. Keep repeating the procedure for each epochs.

We can use the image classification problem as an example to elaborate the points. For example, in the image classification problem, the humans used to extract various

features previously like edge detection, texture classification and segmentation. However, when we use the CNN models, it automates the process of image classification and learns the various features by itself using back propagation idea wherein it keeps on improvising and uses it to get better results. Also, they are able to learn the non linear features present in the model and they are able to learn them quite nicely. This helps to understand the variations between the different classes. CNN performs the convolutional operation at the start of the process network and thus is able to capture the features correctly as uses many types of filters, so as to learn the different types of variations present in the data. The CNN models learn both the linear as well as the non-linear parts in the dataset.
Other advantages are robustness in feature extraction, fewer memory requirements, easier and better training, higher classification accuracy.
Recently post 2013, there are many papers written and published as well as studies done where we can see that CNN has been evolving and performing to its high standards much better than the previously suggested models like ANN, MLP and so on. There are research papers too which have shown the comparison between the traditional method and the new CNN model stating the progress achieved as well as in the domain of image classification, how efficiently CNN extracts features automatically and gets better accuracy for the output. This is a very strong model which uses layers of feature detection, decides the number of filters used and also the learning rate and it updates the weight using back propagation such that it gets better outputs and learns efficiently through feature extraction.

- The loss function and the classical backpropagation (BP) optimization procedure to train such a convolutional neural network is explained below.

Back propagation is a method used to compute the gradient of the cost function needed to calculate the weights to be used in the network. It is like the backward propagation of errors, since in the network as the error is computed at the output of the model, it is distributed in a backwards manner through the network layers. This is mainly used to train the CNN. BP is a supervised learning method for multi-layer feed forward networks.

(Source: http://neuralnetworksanddeeplearning.com/chap2.html)

Input x**:** Set the corresponding activation a^1 for the input layer.

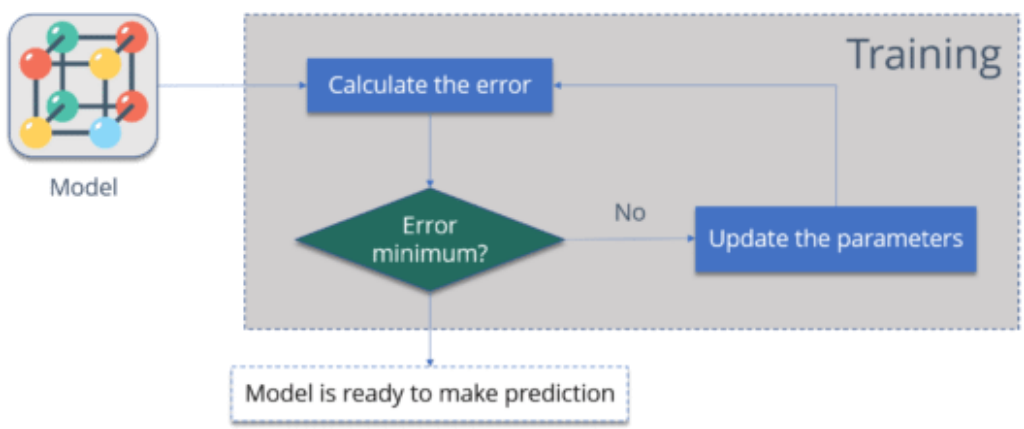
Feedforward: For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

Output error δ^L**:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

Backpropagate the error: For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

Output: The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

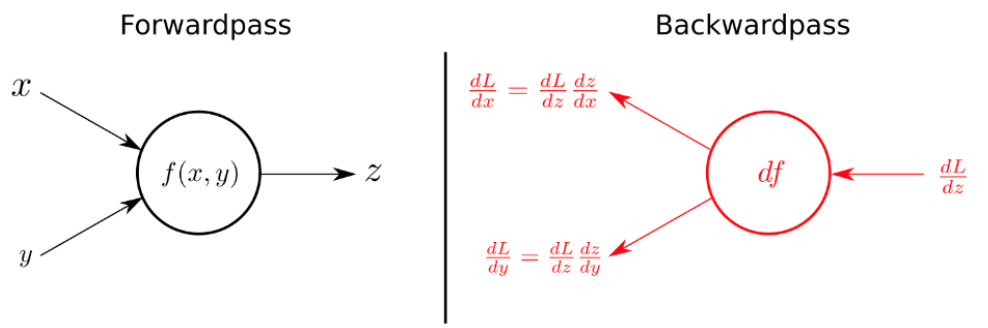
(Source: https://www.edureka.co/blog/backpropagation/)



The procedure is:

- Calculate the error – How far is your model output from the actual output.
- Minimum Error – Check whether the error is minimized or not.
- Update the parameters – If the error is huge then, update the parameters (weights and biases). After that again check the error. Repeat the process until the error becomes minimum.
- Model is ready to make a prediction – Once the error becomes minimum, you can feed some inputs to your model and it will produce the output.

(Source: https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199)



(Source: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/)

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

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

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

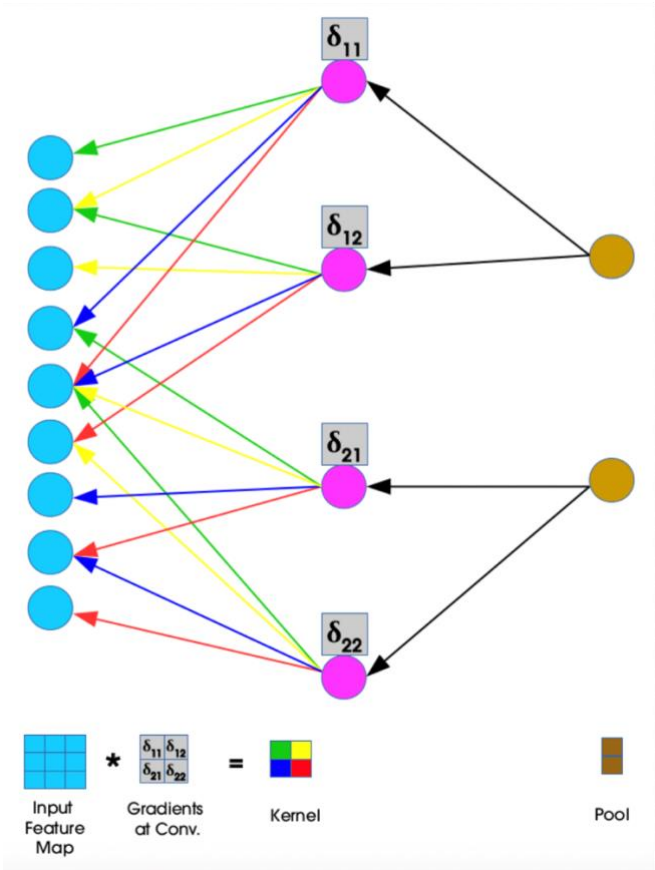
$$h_{22} = W_{11}X_{22} + W_{12}X_{23} + W_{21}X_{32} + W_{22}X_{33}$$

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

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

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

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



CNN made the feature extraction process automatic. Then measure the cross entropy and modify and get better output for lesser error. This is done for all the training images, input taken output got and compared and made better. This process is called one epoch and do back propagation for each batch in it. Keep repeating the procedure for each epochs. For example, MNIST usually require 20-30 epochs. For the MNIST dataset, there are 60,000 hand written digit images of size 32x32, which is grayscale, monochrome i.e. 8-bit. This is for training, put the images one by one and then do the back propagation one by one. Now say all the images in the training set are set once, then this is called one epoch. Basically there is random initialization of filter weights. Then do the back propagation to adjust these filter weights. Now do one epoch, other epoch and after several epochs say 20-25. Go on doing until the error does not reduce more. Should come to be flat curve and we can say it has reached convergence. Once this level is reached, the training is completed. And the filter weights got can be used for testing.

Loss function:
It basically denotes the difference between the original output and the desired output. The cost function should be low for the model to perform better.

The loss function is a function that maps values of one or more variables onto a real number intuitively representing some "cost" associated with those values. For backpropagation, the loss function calculates the difference between the network output and its expected output, after a training example has propagated through the network.

Assumptions [edit]

The mathematical expression of the loss function must fullfill two conditions in order for it to be possibly used in back propagation.[6] The first is that it can be written as an average $E = \frac{1}{n}\sum_x E_x$ over error functions E_x, for n individual training examples, x. The reason for this assumption is that the backpropagation algorithm calculates the gradient of the error function for a single training example, which needs to be generalized to the overall error function. The second assumption is that it can be written as a function of the outputs from the neural network.

Example loss function [edit]

Let y, y' be vectors in $\mathbb{R}^n$.

Select an error function $E(y, y')$ measuring the difference between two outputs. The standard choice is the square of the Euclidean distance between the vectors y and y':

$$E(y, y') = \frac{1}{2}\|y - y'\|^2$$

The error function over n training examples can then be written as an average of losses over individual examples:

$$E = \frac{1}{2n}\sum_x \|(y(x) - y'(x))\|^2$$

The cross entropy loss function is given below:

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

The mse is given below:

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

The loss function we used in the MNIST architecture is categorical_crossentropy. But there are many other types too like binary_crossentropy, mse (mean square error).

(b) Train LeNet-5 on MNIST Dataset

I. Abstract and Motivation

(Source: http://cs231n.github.io/convolutional-networks/)

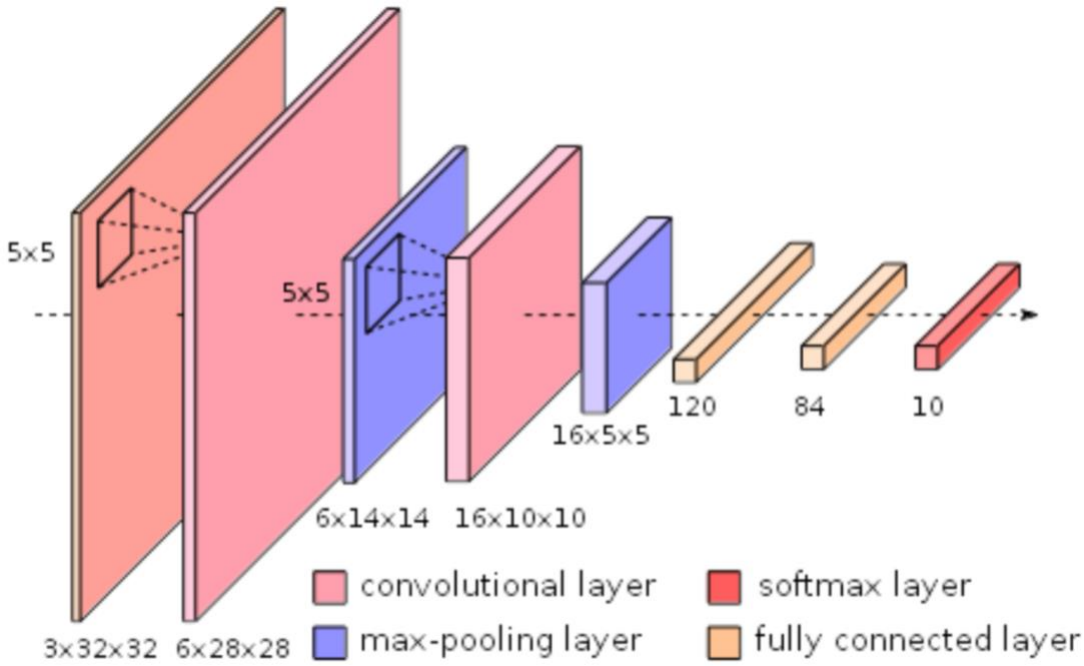
This problem is mainly performed to get a hands-on experience in implementing CNN – LeNet5 architecture on MNIST dataset to get better accuracy by adjusting various parameters. Thus knowing the effect of the various parameters and hyperparameters settings changed. Also, to become comfortable with the various software packages, platforms and languages and functions. I have used Keras (in

Python), running it on CPU, coded on the platform Anaconda – Sypder with the help of Tensorflow.

II. Approach and Procedure

Train the CNN using the 60,000 training images from the MNIST dataset.

- Compute the accuracy performance curves using the epoch-accuracy (or iteration-accuracy) plot on training and test datasets separately. Plot the performance curves under 5 different yet representative parameter settings. Discuss your observations and the effect of different settings.
- Find the best parameter setting to achieve the highest accuracy on the test set. Then, plot the performance curves for the test set and the training set under this setting.



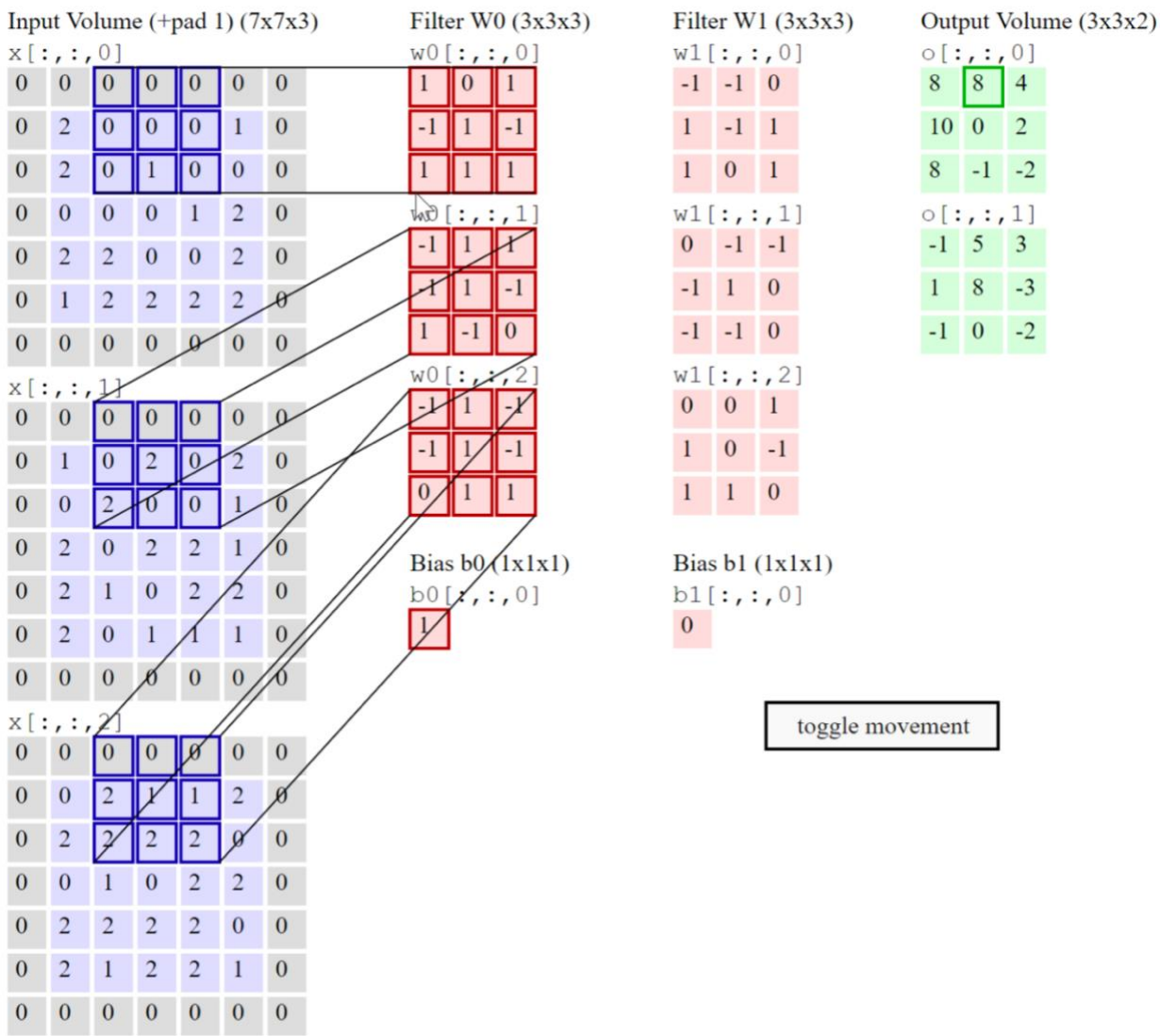
The diagram above uses 32x32 RGB images. But no need to use RGB input as shown in the diagram. Even using single channel i.e. grayscale input for MNIST data is okay. If we load MNIST from Keras, we notice that MNIST is actually 28x28, so the reason for using 32x32 is it is stated in the LeNet-5 paper. The key is that we want the receptive centers of the receptive fields of the last convolution layer. This is done as to cover every detail of the character. Because the character of MNIST will be 20x20. So it is actually carefully computed for the MNIST dataset. But for this problem, we can use 32x32 or 28x28 for the input. It does not matter which one

we use for this homework. Also, we can use 80 or 84 in the second fully connected layer.

MNIST stands for Modified National Institute of Standards and Technology.

(Source: Discussion lecture dated 03/19/2019)

Function of the Convolution layer:

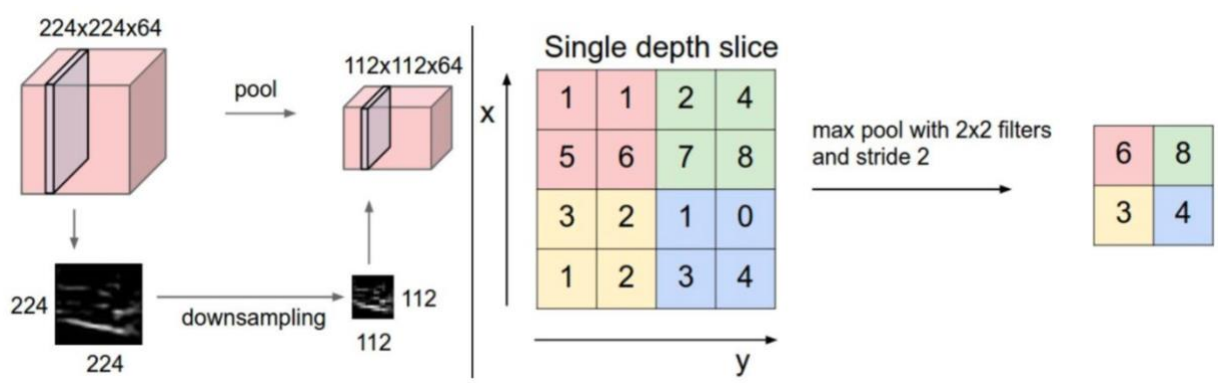


We have a 3x3 filter size and for each of the output map we got 3x3x3 rows and columns. So here the filter rate is 3x3x3. So we perform convolution with the input i.e. basically the dot products. So we compute the dot products, adding them up and end up with a result in the feature mapping at the output. So for 2 set of weights, we

get 2 set of activation maps and we stack them together and that's the output. The stride i.e. step size we use is 1 which is similar to the image filter. There is no padding used. Thus the output size will shrink. Because we do not have padding for the boundary pixels, where we used to center at the boundary pixels when doing the convolution.

Say, taking a Numpy example. We have an input X (11, 11, 4). This is the shape of the input and if the filter size = 5 and stride is 2, we get the spatial size of the output as 4, giving a volume with width and height as 4. Also the output volume depends on the number of the filters used. The operation is as follows. For the first elements in the outputs, say (0,0) elements in f, we basically take the first 5x5 block in the input and then we do the dot product with the filter weight plus the bias so as to get the final output. For the other parts, repeat the steps similarly, we move the filter down and start with 2 as stride is 2. Suppose we have 2 filters, doing the above procedure by sliding each filter across the input image, we get one activation map for the output column. After this, we consider another filter with a different weight and basically we repeat the same operation and get the second activation map for the output. This is what the conv layer is doing.

Function of the Pooling layer:



Basically for max pooling, we are doing that independently along each depth dimension. We are doing that spatially. So we are downsizing each one here spatially. And we keep the depth dimension here. So for each spatial size, for each depth size, suppose we are doing 2x2 filtering with stride 2 here, so we take the 2x2 window. If we are doing the max pooling, we take the maximum value here and putting it into output.

For the pooling layer, here we are downsampling the volume. So we are downsampling in the spatial dimension. We are doing this independently for each

depth size. So for each depth size, for example if we do max pooling, wherein we take the maximum value and put it in the output. And this is basically the operation we have for the pooling layer.

The fully connected (FC) layer is quite similar to the one we have in neural networks. Between the convolution layer and full connected layer, in the implementation add a flattened layer to flatten the outputs of the convolution layer, before doing the dense FC layer.

For this problem, for all activation, I have used ReLU, except for the last one, I used softmax to compute the probability. So the softmax is basically like taking a bunch of values like suppose have k values, and then normalizing it to make values represent probabilities, so probability must add upto 1. Thus normalizing it using some sort of function.

To run the code, following options can be used.

1. Google Colaboratory
2. Install Tensorflow/PyTorch on local machine (available for Ubuntu, Mac OS, Windows)
   https://www.tensorflow.org/install/pip
   https://pytorch.org/get-started/locally/
3. Virtual Machine

(Source: https://research.google.com/colaboratory/faq.html)

Google Colaboratory:

- Jupyter-like notebook
- Free GPU
- CPU can also be used (for this problem, it is okay)
- Tensorflow, Keras, Numpy and other dependencies pre-installed
- It allows a virtual machine each time used. There is a lifetime for the virtual machine. The homework problem can be implemented within this time (requires less than half an hour).

For this problem, CPU should be sufficient, however, can use GPUs if have the access. Deep learning frameworks/libraries are used such as Tensorflow or PyTorch, Keras or tf.keras.

Keras:

- High level language

- High-level neural network API
- Wrapper of Tensorflow, Theano and CNTK ("backend") – Keras is built on top of these languages. It is an abstraction of these languages. Can use either of the ends to run code on Keras.
- Comparing Keras vs Tensorflow (https://gist.github.com/ricgu8086/0ba44ce3aab19ec50425383a4d778b50)
- Two types of models:
  Sequential: a linear stack of layers and the other one is complicated
  Model class used with the function API
- Sequential would be sufficient for this homework

(Source: https://keras.io/getting-started/sequential-model-guide/)

The sequence model is created as shown below.
Pass a list of layers to the constructor when creating a Sequential model. Thus it defines the sequential model with those layers.

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

Or can create an empty model and then use .add() method to add the layers to the sequential model.

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

The layers used for this homework problem are:
- Conv2D
- MaxPooling2D
- Flatten
- Dense (Fully connected layer)

Here, I referred the Keras document for full usage of the functions above.

The model is pretty much build up using the steps above and now after the model is built, we need to configure the training process which include the optimizer, the loss function and the metric here is basically what we want to compute. So for the classification problem, we set the metric as accuracy as shown below. And is using the loss function as below, refer to the note. The note says that convert the label to the one hot format.

Configure the learning processing.
Use the .compile() method as below.

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Note: when using the categorical_crossentropy loss, your targets should be in categorical format (e.g. if you have 10 classes, the target for each sample should be a 10-dimensional vector that is all-zeros except for a 1 at the index corresponding to the class of the sample).
In order to convert *integer targets* into *categorical targets*, you can use the Keras utility to_categorical
(Refer to https://keras.io/losses/#usage-of-loss-functions for details)

(Source: https://keras.io/models/sequential/)

Start training.

- Keras models are trained on Numpy arrays of input data and labels
- Use .fit()

```
# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, labels, epochs=10, batch_size=32)
```

```
# For a single-input model with 10 classes (categorical classification):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

First step is to define the model using model sequential and add method. And then configure the training process. There are 3 inputs, the optimizer, loss and metrics. Then define some data and use model fit to actually train the model on the data.

We need to plot the accuracy vs epoch for the testing data as well. So can use the validation data, the input argument for the fit function. This argument is very useful for this job.

For this problem, replace the architecture shown above and replace the optimizer and loss function to experiment with, because 5 different settings are needed. Also, replace the data with the MNIST data. In Keras, MNIST is directly provided so no need of downloading and loading the data externally. And just fit the function to the data.

(Source: https://keras.io/models/sequential/)

The link above provides the API for the sequential model. model.evaluate() is a function used that can evaluate the model to compute the metric already defined. And there is also predict function and other functions which can be used from this link.

The requirement is to get at least 98% accuracy for the best model in part(b) and the proposed model in part(c). Also, need to experiment with the various settings or hyperparameters like learning rate, initialization methods, loss function, regularizer, optimizer, how to initialize the weights and so on. For part(b), need to include 5 different settings.

(Source: Discussion lecture dated 03/26/2019 and 04/02/2019)

Need to compute the mean and variance of the final training and testing accuracies over the 5 settings tried in part(b).

(Source: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf)

Also, LeNet-5 was designed for 32x32 input, the reason is that want to make sure that the centers of the receptive fields in the second Conv layer would be 28x28 in that case would cover ever useful detail. Because in MNIST, the original data was scaled to 28x28 and it contains all important detail and they then put it in 32x32 environment. Thus, we have MNIST's 32x32 and 28x28 images. And we have 2 choices, one is that manually pad the images with the background pixels to 32x32, another way is use 28x28 but change the padding setting for the first Conv layer. So for the first Conv layer, we don't have any padding right now. So if we have 32x32 input, we get 28x28 output for the first Conv layer. But if we change the padding setting to be the same, then the output will be the same size as the input. So, if we use 28x28 the output for the first Conv layer is actually 28x28 as well. So it wont change the activation map sizes in this case. Third option is that use 28x28 input and keep the no padding settings. In this case, it will shrink the output of the activation map. So the size of the activation map will be different in this case.

1) pad the images to 32x32 with background pixels, or
2) use 28x28 but use the "same" padding (padding the input such that the output has the same size as the original input) for the first conv layer, or
3) use 28x28 input without changing the layer settings: this will result in activation maps of different size as given.

Can also use PyTorch on Google Colab.

Starting with the building blocks for PyTorch. PyTorch uses Tensors. This is generalization for the matrices vectors. So, if have 1D – vector, 2D – matrix and if multi-dimension then imagine it as a tensor. Tensor is similar to numpy ndarray. It also allows GPU acceleration or speed up. It becomes important for the reason that it tracks the gradient automatically. So we do not need to deal with the back propagation again. The syntax is similar to numpy array as we see below.

Construct a matrix filled zeros and of dtype long:

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

Out:

```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

Above is defined zeros of 5x3. Some operations for tensors are shown below. For PyTorch, there is a bunch of in place operator and they all are postscript by the underscore. Here, we change the y directly, instead of later returning the result. There

are various operators mentioned in the source below. Also can use standard Numpy like indexing. For resizing, can use the function view.

(Source: https://pytorch.org/docs/stable/torch.html)

Addition:

```
y = torch.rand(5, 3)
print(x + y)

print(torch.add(x, y))
```

In-place version:

```
# adds x to y
y.add_(x)
print(y)
```

Standard Numpy-like indexing:

```
print(x[:, 1])
```

Resizing:

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
print(x.size(), y.size(), z.size())
```

Out:

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

As tensors are very similar to Numpy they can actually change to each other quite easily. But if we change them, they actually share the same memory location. So if

change one, change the others as well. Below, b is a numpy array such that a is a tensor and changed to b. Both a and b are changed when add 1 to a.

Torch Tensor and NumPy array will share their underlying memory locations. Thus, changing one will change the other.

Tensor → Numpy

```
a = torch.ones(5)
b = a.numpy()
```

```
a.add_(1)
print(a)
print(b)
```

Out:

```
tensor([2., 2., 2., 2., 2.])
[2. 2. 2. 2. 2.]
```

Numpy → Tensor

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

Out:

```
[2. 2. 2. 2. 2.]
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
```

Autograd

In PyTorch, the grad is built by running and the gradient is automatically trapped and no need to do the backward propagation. To track the history, need to set the required grad to True. The function torch.no_grad() says that if we wrap a bunch of code under with no_grad, the output of the operation will have required grad equals to false. And this is very convenient to do evaluation. But in this case we don't really need to check the gradient because we are doing evaluation and not training. So this saves a bunch of memory. When doing the evaluation code or testing, rewarp with no_grad, because not using the gradient.

Create a tensor and set `requires_grad=True` to track computation with it

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
```

```
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```

Out:

```
True
True
False
```

CLASS `torch.autograd.no_grad` [SOURCE]

Context-manager that disabled gradient calculation.

Disabling gradient calculation is useful for inference, when you are sure that you will not call `Tensor.backward()`. It will reduce memory consumption for computations that would otherwise have *requires_grad=True*. In this mode, the result of every computation will have *requires_grad=False*, even when the inputs have *requires_grad=True*.

Also functions as a decorator.

Example:

```
>>> x = torch.tensor([1], requires_grad=True)
>>> with torch.no_grad():
...   y = x * 2
>>> y.requires_grad
False
>>> @torch.no_grad()
... def doubler(x):
...     return x * 2
>>> z = doubler(x)
>>> z.requires_grad
False
```

(Source: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html)

Now showing the example given by PyTorch which is example code on CIFAR-10 showing how to load data, how to define the network etc.
Firstly, need to import torch. torchvision is used to load some common datasets. Using trainloader and dataloader, we can load the batch of data from the dataser. After we get the data, now we define the network. So there are 3 components which we need to consider – network, loss function and optimizer. To define the network, inherit the superclass Net(nn.Module). For initialization part, define a bunch of layers. The forward function is a must for the class. It basically defines what the network is doing. Get the input in and define the first conv layer, then the relu activation and the max pooling. And after that, we do the second conv layer, relu and pool. Now, we flatten the output of the convolution layers. Use this for the fully connected layers 1, 2 and 3. Then create an instance of the model.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Now, the loss function, it takes a pair of the output and the target. So it compares some loss for the output against the target. For the same, can use CrossEntropyLoss(). The optimizer basically does the thing to update the weight. Define it using SGD. It takes in the parameters that are needed to optimize and some other hyperparameters. And inside the training loop, call the optimizer step to update the weight. Also, need to zero out the grad inside the training loop or else the gradient will go on accumulate. Thus, use zero_grad to avoid this. Now ready to train. For training, decide how many epochs to run say 25, and inside each epoch, iterate through all the training data. Inside this loop, take the training data, the input as well as the labels. Then zero out the gradient and compute the loss. The loss depends on the output. The output is passing the input through the network. Then call loss.backward() and optimizer.step() to update weights. This will complete one iteration. Do this for every batch in the training set and that completes one epoch. And repeat it for many epochs.

(Source: https: //discuss. pytorch. org/t/what-is-the-difference-between-torch-nn-and-torch-nn-functional/33597)

(Source: https: //discuss. pytorch. org/t/beginner-should-relu-sigmoid-be-called-in-the-init-method/18689/5)

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

Firstly define a network and this network will subclass the nn module. And then in the init function, define some layers that has some learnable parameters. In the forward function, this is the part where we define what the network is really doing. And it consists of conv, relu, pooling, another conv, relu, pooling and flattened it and 3 FC layers and got the result and this is what this network is doing.

• NOTE

Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

• NOTE

`torch.nn` only supports mini-batches. The entire `torch.nn` package only supports inputs that are a mini-batch of samples, and not a single sample.

For example, `nn.Conv2d` will take in a 4D Tensor of `nSamples` `x` `nChannels` `x` `Height` `x` `Width`.

If you have a single sample, just use `input.unsqueeze(0)` to add a fake batch dimension.

After defined the network architecture, define the loss function as below. The example below shows the mse loss. And to compute the loss call the criterion – output, target.

```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
```

Need to create the optimizer as below. There are different classes of optimizer to choose from which are checked on the PyTorch website and compared which suits the best.

```
# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)
```

Now can start the training, below is the training loop. We run that for 2 epoch and in each epoch, iterate through all the data, all the training data samples and we take the input and label out of it. In each training loop, before updating the network, need to zero out the gradient. Otherwise, the gradient will accumulate. Next is to feed in the input and get the output. Then calculate the loss and do the backward propagation and updating the weights.

```python
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

The testing part is pretty similar. The code below is also calculating the accuracy. It is wrapping up the testing code under torch.no_grad flag, so as to use less memory. Under this, take out the testing batches, quite similar to the training code, except that now there is no backward part and stepping. Then pass through to get the network. Get the prediction from the output. Check if the output is the same as the labels, expected labels, if they are the same, we get 1. Add up the 1s which obtains the number of correct predictions. If we divide the total number of correct predictions by the total number of predictions, we get the accuracy. Thus the testing part is over.

For testing, iterate through all the data samples in the testing dataset. And then feeding it to the network and getting the output. And do the post processing. Below an example of calculating accuracy is shown. Can implement other type of post processing.

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

The summary is that first we need to determine the datasets, both for training and testing. Then load and get the batch of data every time. Second part is for defining the network, a forward function showing what the network is actually doing. Doing the conv1 layer, followed by relu, then max pooling then conv2 layer and repeat, flattened layer, fully connected layers 1 to 3. Third part is to determine the loss function and optimizer. Thus, the training loop is applied and then the testing is done. We find the accuracy results.

Try different 5 settings in this homework i.e. try some hyperparamters like learning rate, different loss function or different optimizer, different training settings, epochs, batch size etc. Also, can adjust the network architecture as well i.e. adjust the number of filters in each conv layer or can add dropout or can add more conv layers or activation function or number of neurons. Tune the hyperparameters in such a manner that the accuracy achieved is at least 98%. Now analyze for different hyperparameters settings if the performance got better or worse.

Need to compute the <u>mean and variance</u> of your final training and testing accuracies over the 5 settings we tried in part b). Included in the report:
1) the final training and testing accuracies of each setting;
2) the final training and testing accuracies of the best setting;
3) the mean and variance of your final training and testing accuracies over the 5 different settings.

If tried some bad or harmful values for settings, the mean is very low and variance becomes high. To plot the epoch-accuracy curve for the test set, since model.evaluate runs only for one epoch; the test set is passed as a validation set in

model.fit just for the plot. Pass the test set to fit(). Test the model at the end of every epoch. Use the epoch-accuracy plots to help analyze. The final testing accuracy is also helpful. Find the mean and variance of the 5 models.

Review on Keras:

```
# For a single-input model with 10 classes (categorical classification):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

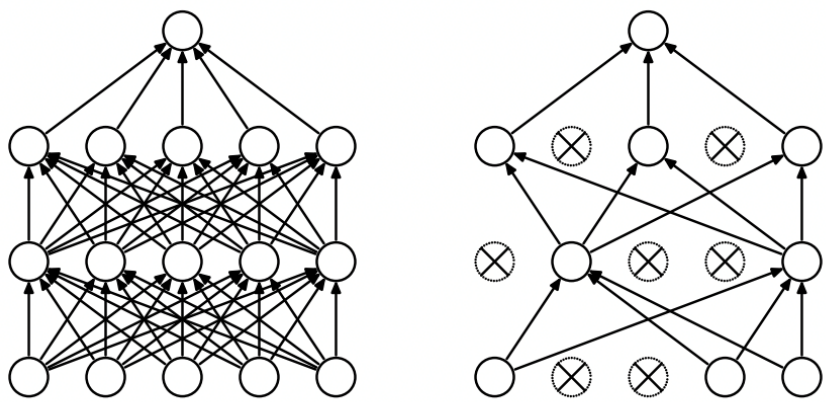
```
model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

First thing to do is to define the network. Define the network as sequential network which is basically a stack of layers. So train a sequential network and use add function to add layers to it like adding FC layers to it. One can also just pass the layers as arguments for the constructor for the sequential network. Use the compile function to configure the training process. Define what is the optimizer for the training. Can use the built-in classes for the optimizer like SGD and can pass the parameters like learning rate etc to the SGD object and then pass to the optimizer to adjust. Also define the loss function. Can choose from a bunch of loss functions. The dummy data part above is replaced by the MNIST data. And can load the MNIST

from Keras directly. Lastly for training, use the fit() function to fit the data and labels and can also configure the best size.

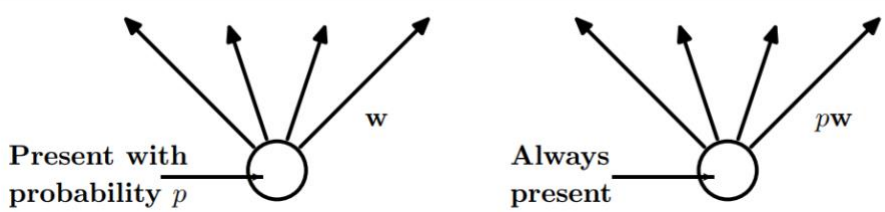
(Source: Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research* 15.1 (2014): 1929-1958)

Dropout:



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.



(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights **w**. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

This is not really needed in the problem solving for part(b). We randomly zero out some inputs for the layer. It's like we are dropping out nodes in the network in the training phase. For the implementation also need to consider the scaling as well. So in the paper it says that, if we have nodes that's present with probability p, then we

need to downscale the testing network output, testing time by p as well. So the network rate will be p.w at testing time. The expected output at training time is similar to the output at the testing time.

Layer Number	Layer Name	Number of filters used	Activation
1	Convolutional layer and Max pooling layer	6	Relu
2	Convolutional layer and Max pooling layer	16	Relu
3	Fully connected layer	120	Relu
4	Fully connected layer	84	Relu
5	Fully connected layer	10	Softmax

The various parameters and hyperparameters along with their varied values which I have adjusted to get better accuracy are given in the table below.

Model Parameters and Hyperparameters	Different settings adjusted
Kernel size	(3,3), (5,5)
Epochs	3, 6, 10, 15, 20, 25, 100
Dropout	0.25, 0.5
Optimizers	Rmsprop, adam, SGD
Batch-size	32, 64, 128
Stride	(1,1)
Activation function	Relu, tanh, sigmoid
Filter number	Depends on the layer chosen different filter numbers
Max pool size	(2,2), (3,3)
Learning rate	0.0001, 0.001, 0.01, 0.1, 0.5
Weight initialization	Random uniform and Glorot uniform
Momentum	0.7, 0.8, 0.9

This problem was performed on a Mac with 2.3 GHz Intel Core i5 configuration. Each epoch took approximately 13-18 secs on an average. The code shown below was run for different parameter settings mentioned in the table above. The testing dataset was basically the validation dataset given, for each epoch to compute the testing accuracy. The history of Training accuracy, Training loss, Validation Set or testing accuracy and Validation Set or testing loss were saved for every epoch or

iteration. Thus the plot for training epochs vs training as well as testing accuracy was computed. Also the final test accuracy got was printed. Also the loss and error were calculated.

Model parameters:

Batch size – The weights are updated in the network model for every batch size. Larger the batch size, slower will be the convergence. This is because, using different batch size for the training, it causes decrease in the variance of the filter weight change. It does not affect the accuracy so much.

Dropout- This is used to reduce overfitting of the data. It is used after the pooling layer followed by the convolution layer.

Learning rate – It is used to state the time for convergence of the model. this value should be small, to provide a better model. If the value is high, then the convergence occurs fast taking bigger steps and sometimes may lead to a wrong local minimum. Thus the value being small like 0.001 takes smaller steps to convergence and gives a good global minimum. However, it really depends on the type of data dealing with.

Kernel size – This is used to determine the size of the kernel window chosen for the correlation of the filter and the data. It should not be too big. I tried different filter size like (3,3), (5,5) and (7,7). Better output was obtained for smaller size (3,3). This is because a smaller window can capture small details and not let the patch be generalized for different type of data. This helps to use these details for image matching in a way which helps to get accurate output.

Pool size – This is the size of the window in which the elements go through max pooling operation where the highest value among the elements is chosen and replaced. The window size should not be too large.

Epochs – This denotes the number of iterations carried out for the network. It can vary from 3 to 200 or more. However, for the MNIST network, it can be on an average chosen to be 15 or 20. For low learning rates, the epoch is usually higher.

Activation function – This parameter depends on the model and usually in real life application, relu and leaky relu work the best. Even for our model it is the same. The curve for relu helps to obtain a better output, by getting better convergence.

Optimizers and loss function – These can be chosen depending on the network. For our MNIST model, rmsprop works good as well as sgd, and the loss function we chose is categorical crossentropy, this is because at the output we have 10 different classes, so this is chosen for the multi-class problem. Adam and rmsprop optimizer gives higher convergence.

Number of filters – Greater number of filters give better accuracy; this is due to more feature extraction with more number of filters. This improves the feature efficiency. The number of filters was varied from 6, 16 to 32, 64.

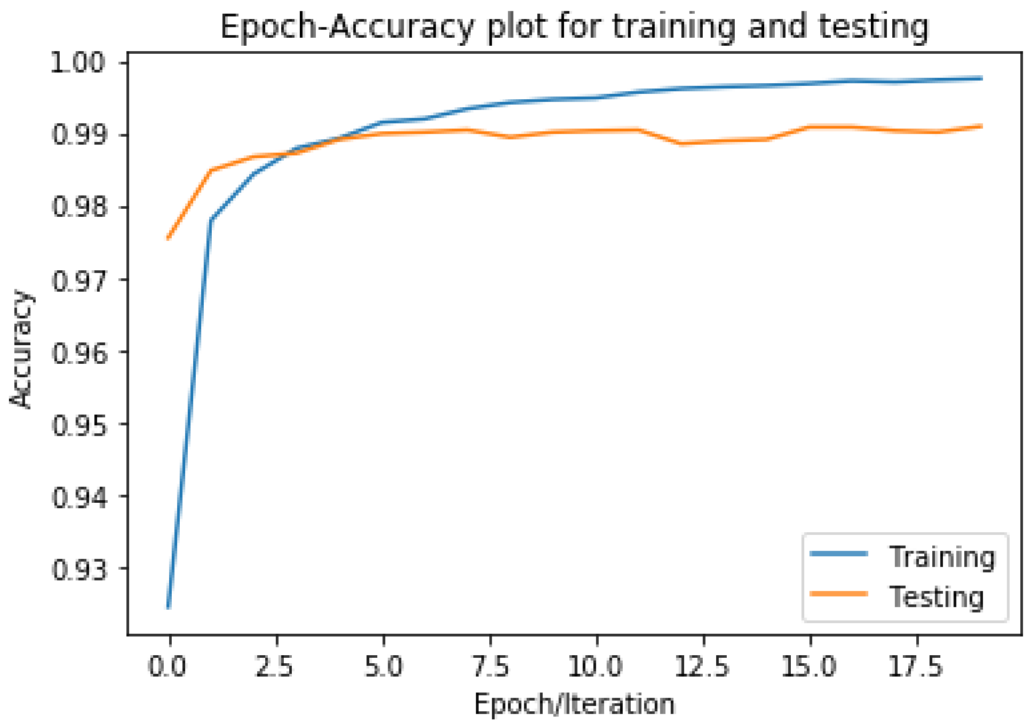
Algorithm implemented in Keras (Python):

- Split the dataset into the training and testing sets
- Load the training data, validation data and testing data from MNIST dataset
- Load the training data, returns in numpy array form
- Load the validation data, returns in numpy array form
- Load the testing data, returns in numpy array form
- Define the LeNet5 architecture network model
- Create a sequential model
- Add two convolution and max pooling layers in order and one flatten layer and three fully connected dense layers
- Define the optimizer and loss and metrics
- Compile the sequential model
- Build the sequential model
- Print the information of the model
- Train and fit the model defining epochs and batch size
- List all data in history
- Perform final evaluation of the lenet5 sequential model
- Printing the error, loss and accuracy for the model after the evaluation
- The plot model history function plots the graphs showing loss and accuracy vs epoch/iteration
- Summarize history for accuracy

III. Experimental Results

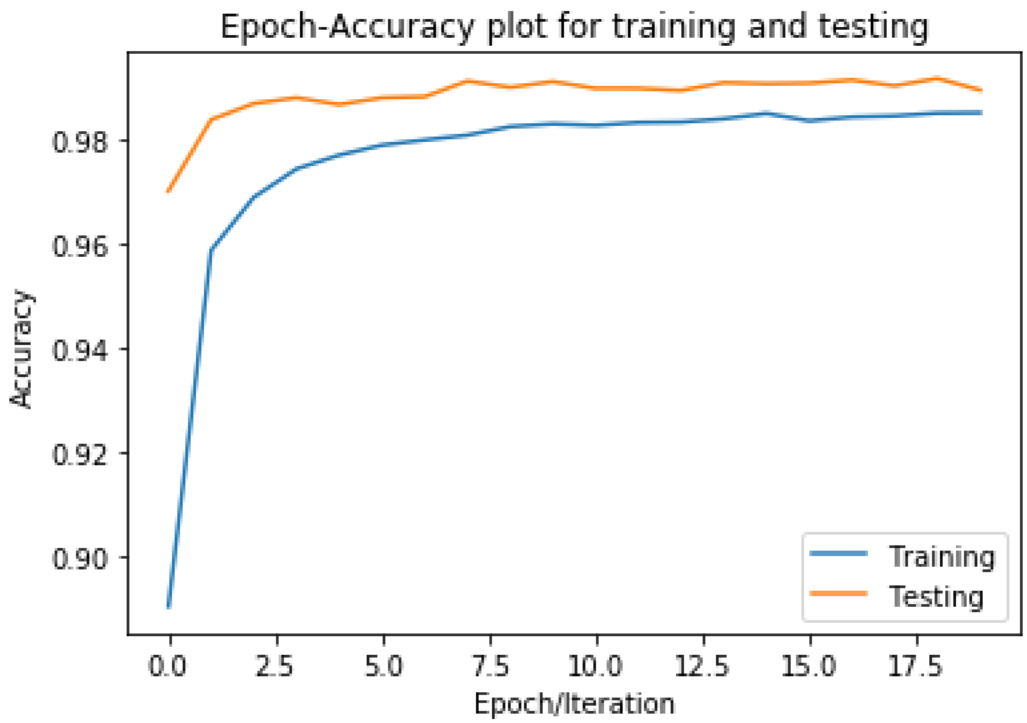
Setting 1: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001,

loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0, momentum = 0.9



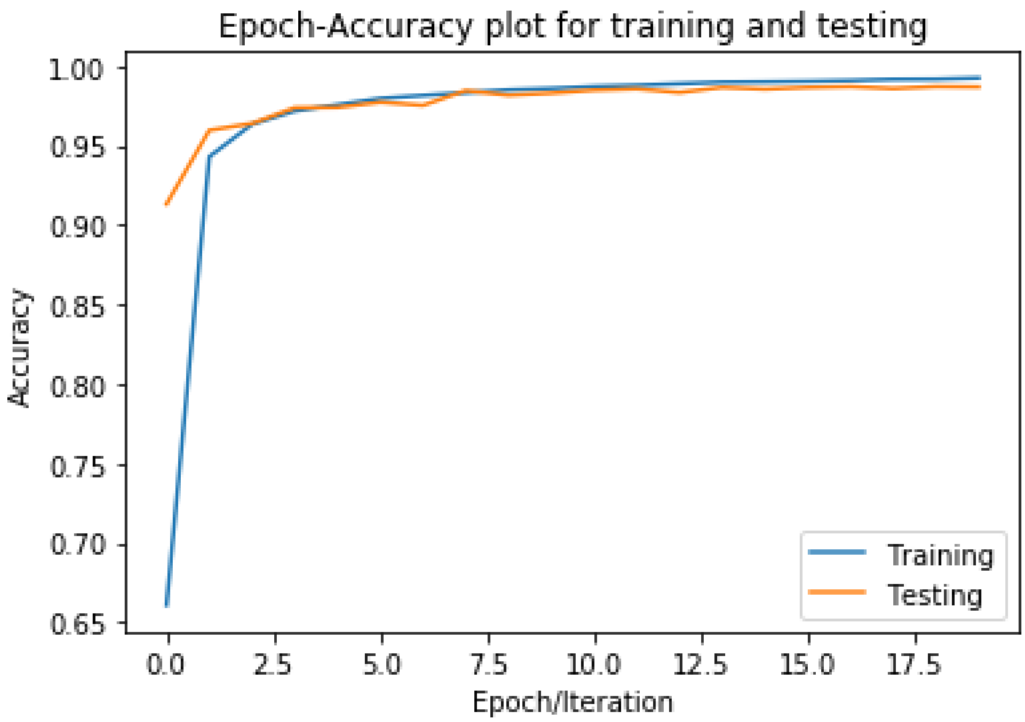
Training accuracy: 99.75%
Testing accuracy: 99.09%

Setting 2: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0.25



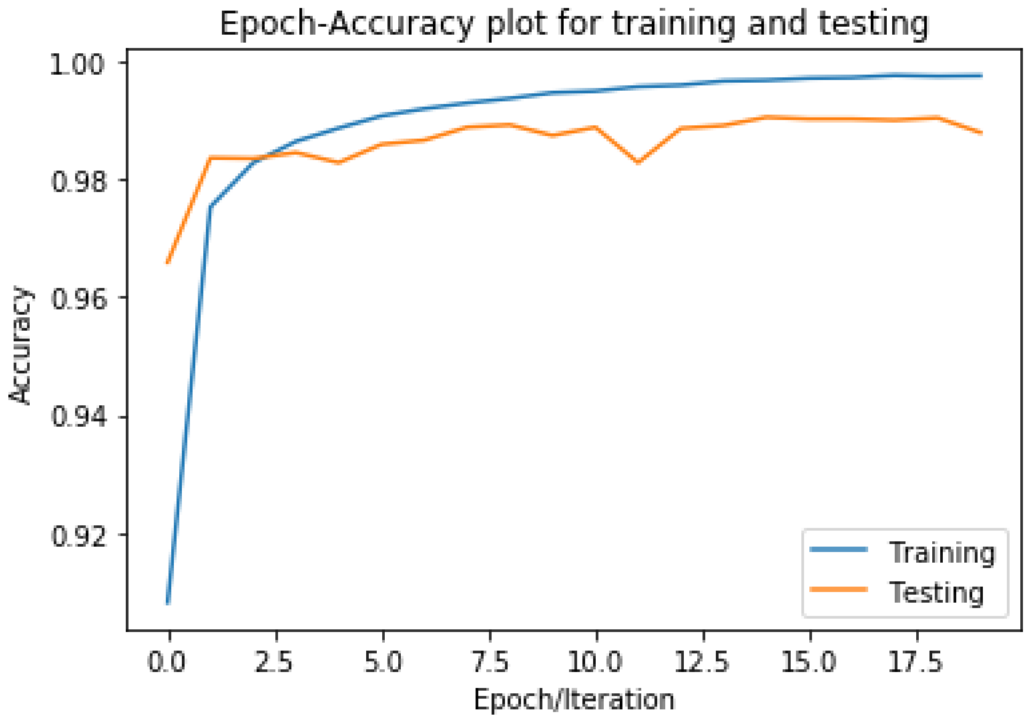
Training accuracy: 98.50%
Testing accuracy: 98.94%

Setting 3: kernel size = (5,5), strides = (1,1), activation = 'sigmoid', weight initialization = 'random uniform', pool size = (2,2), optimizer = 'rmsprop', learning rate = 0.001, loss = 'categorical crossentropy', epochs = 20, batch size = 64, dropout = 0



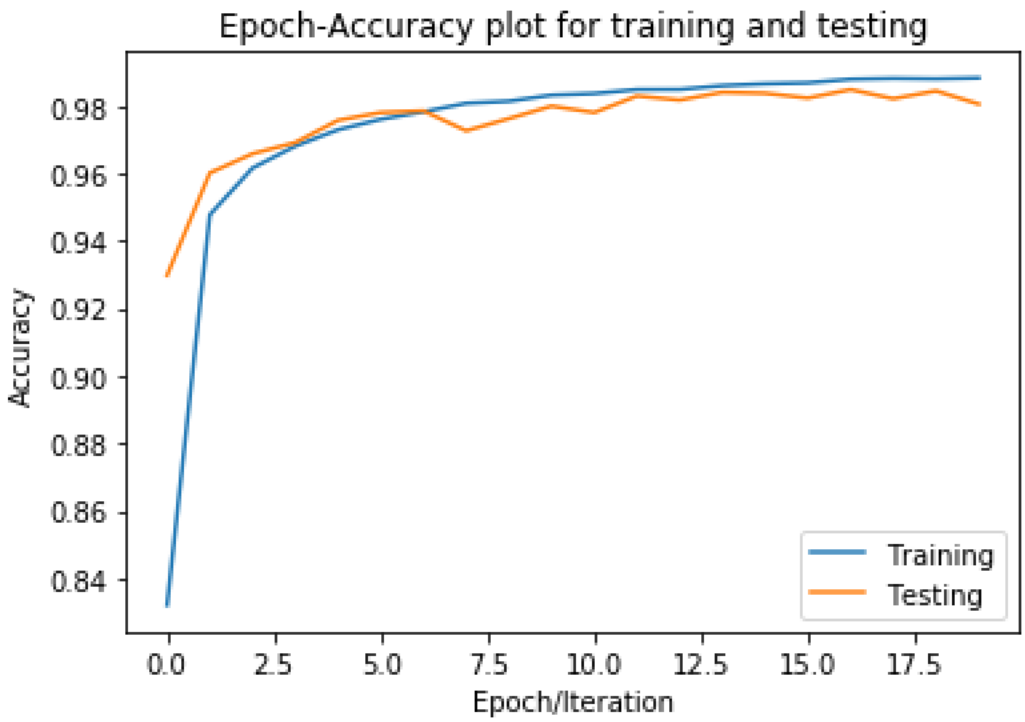
Training accuracy: 99.25%
Testing accuracy: 98.69%

Setting 4: kernel size = (3,3), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0



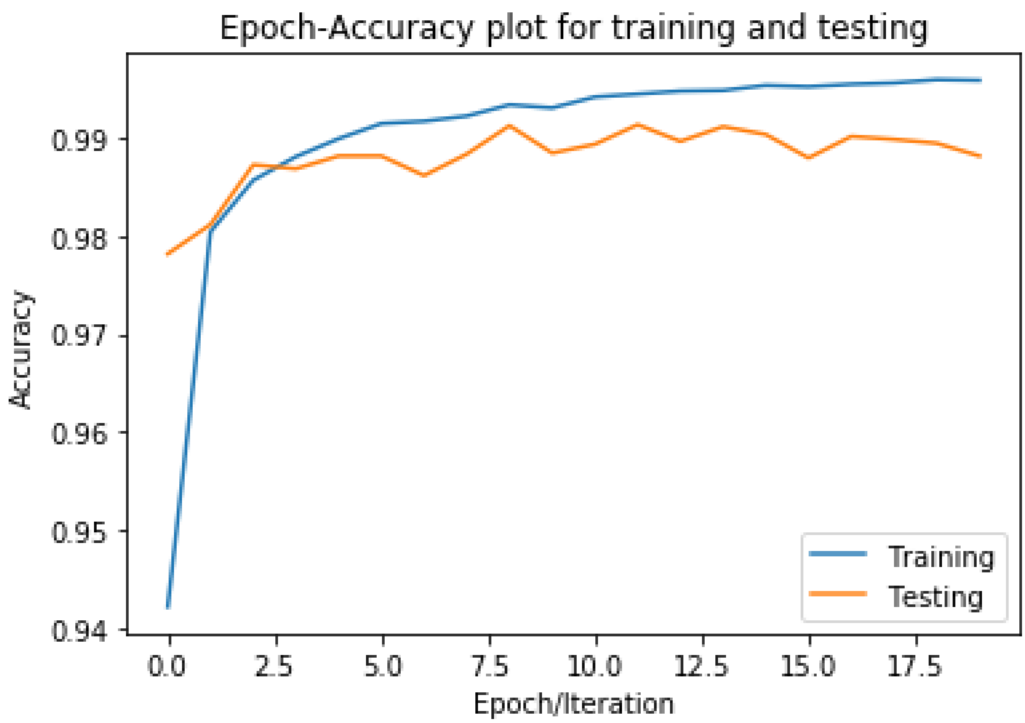
Training accuracy: 99.75%
Testing accuracy: 98.79%

Setting 5: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (3,3), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0



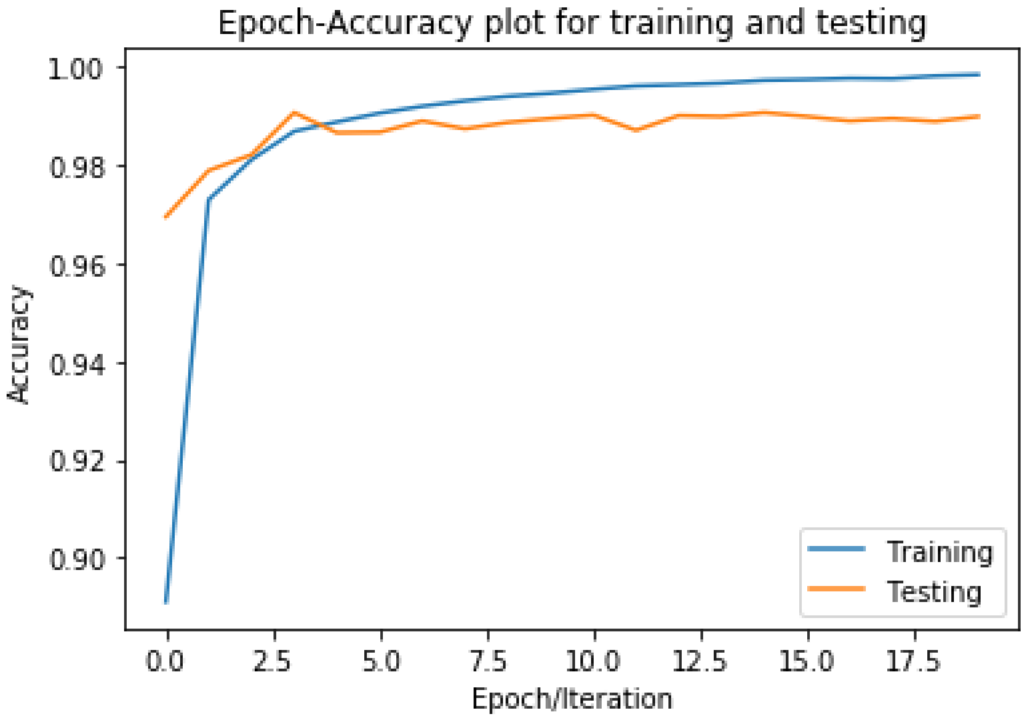
Training accuracy: 98.82%
Testing accuracy: 98.05%

Setting 6: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 32, dropout = 0



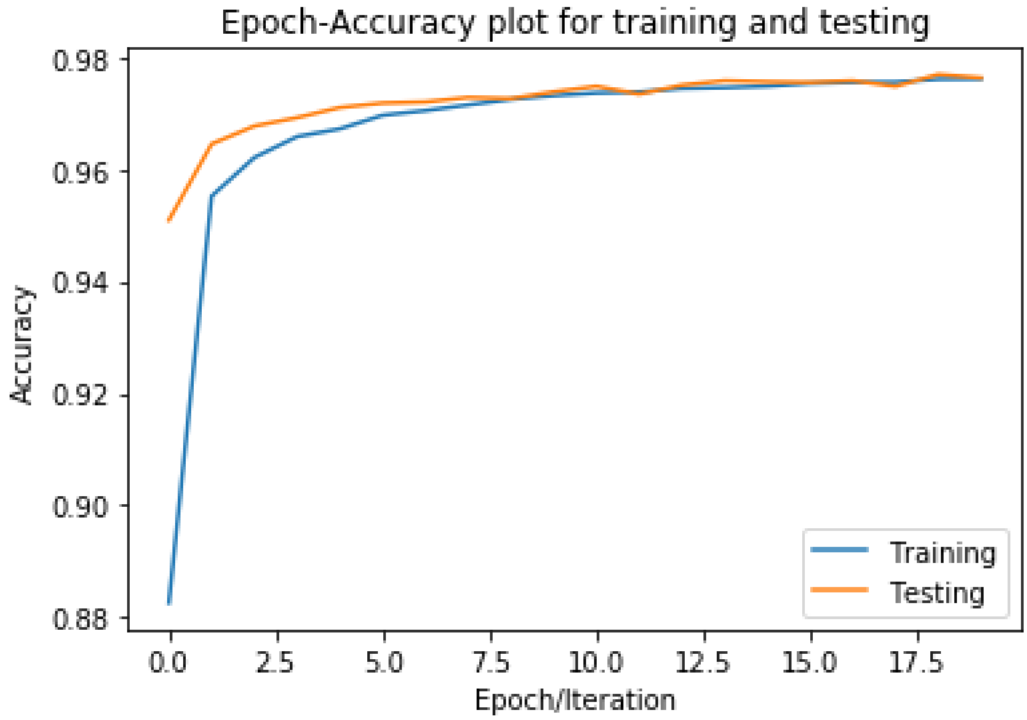
Training accuracy: 99.59%
Testing accuracy: 98.82%

Setting 7: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 128, dropout = 0



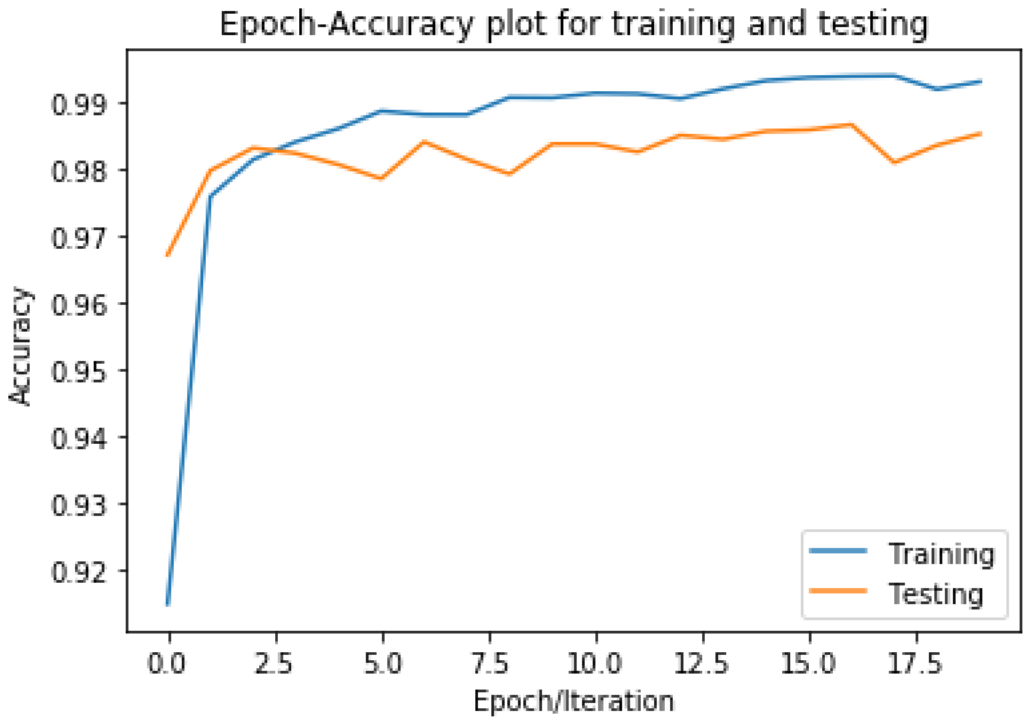
Training accuracy: 99.83%
Testing accuracy: 98.98%

Setting 8: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘sgd’, learning rate = 0.01, decay = 1e-6, momentum = 0.9, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0



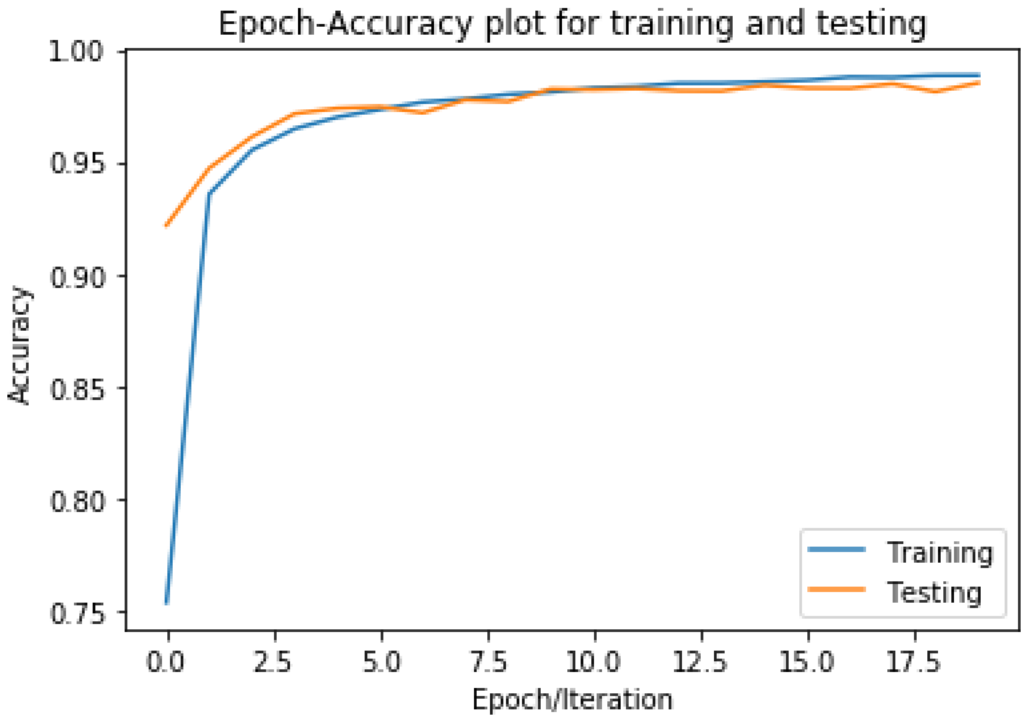
Training accuracy: 97.61%
Testing accuracy: 97.63%

Setting 9: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘sgd’, learning rate = 0.1, decay = 1e-6, momentum = 0.7, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0



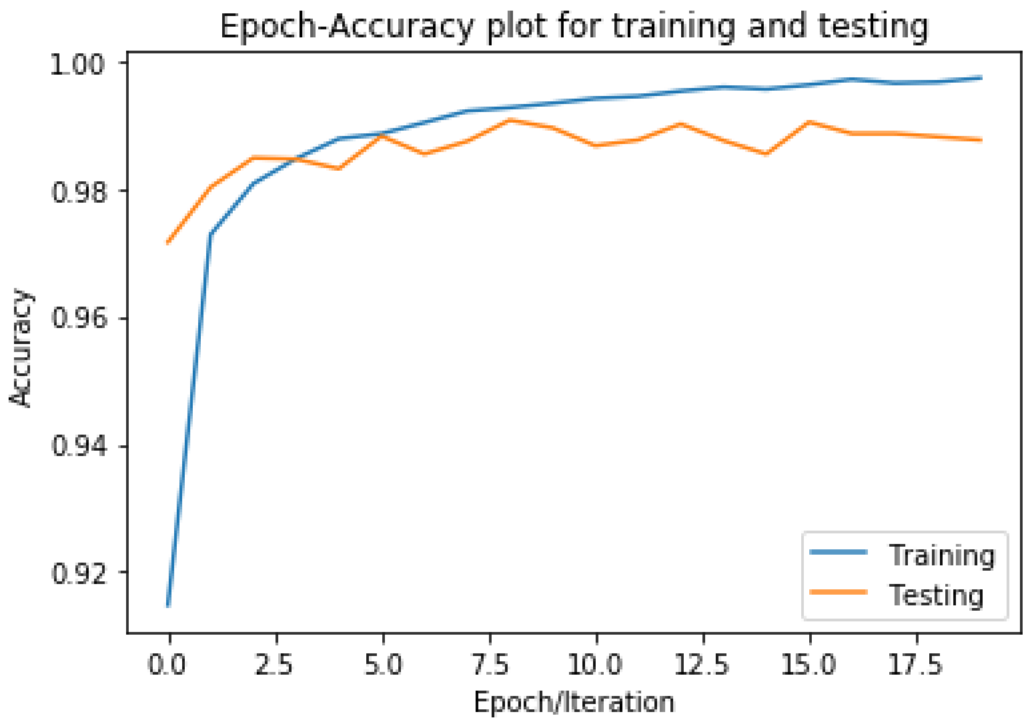
Training accuracy: 99.32%
Testing accuracy: 98.54%

Setting 10: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘glorot uniform’, pool size = (2,2), optimizer = ‘sgd’, learning rate = 0.001, decay = 1e-6, momentum = 0.9, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0



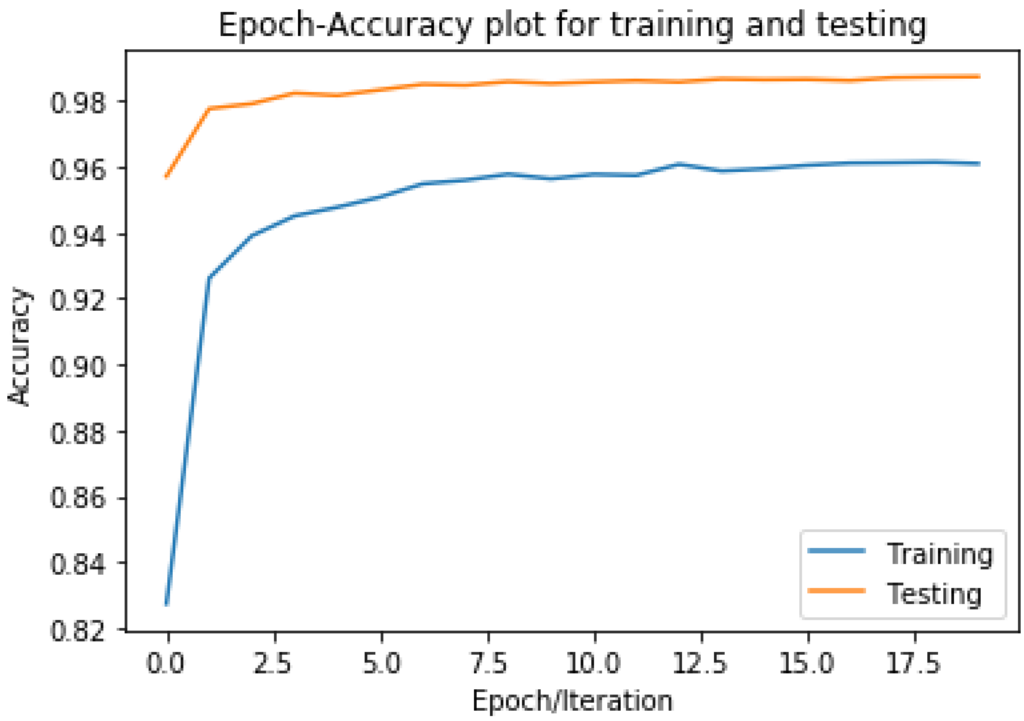
Training accuracy: 98.85%
Testing accuracy: 98.51%

Setting 11: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘adam’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0



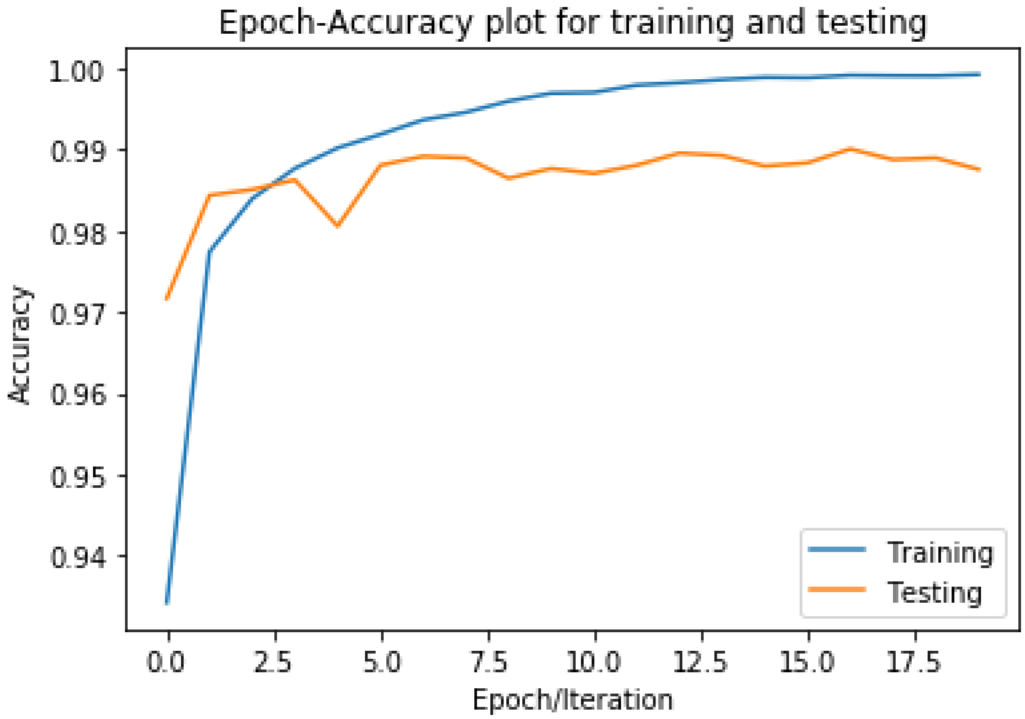
Training accuracy: 99.73%
Testing accuracy: 98.76%

Setting 12: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0.5



Training accuracy: 96.19%
Testing accuracy: 98.73%

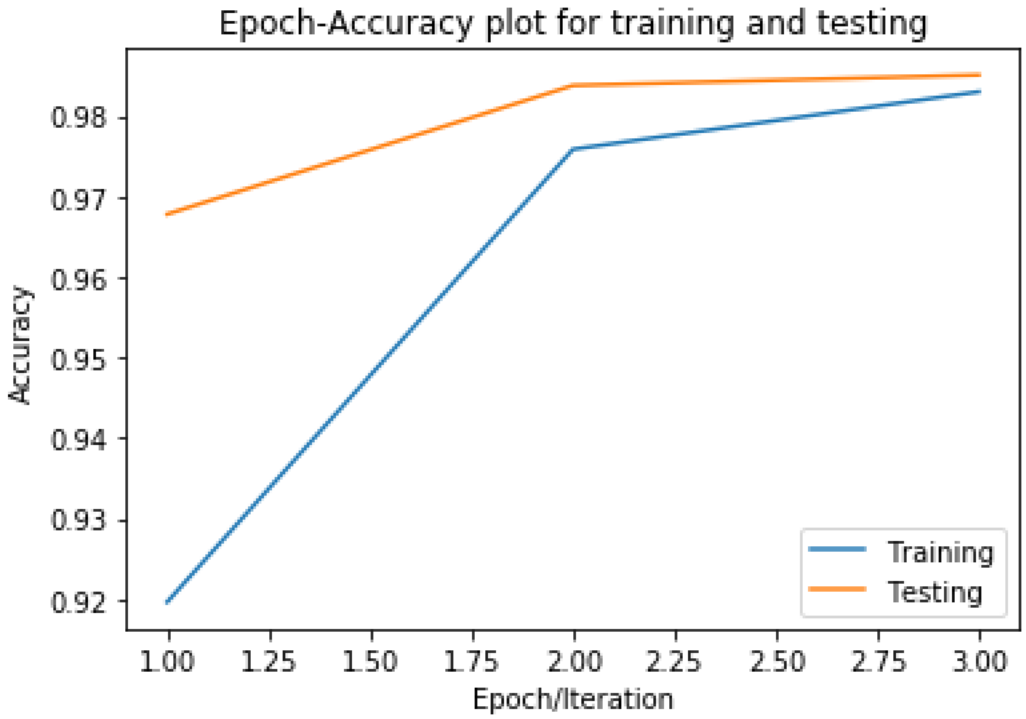
Setting 13: kernel size = (5,5), strides = (1,1), activation = 'tanh', weight initialization = 'random uniform', pool size = (2,2), optimizer = 'rmsprop', learning rate = 0.001, loss = 'categorical crossentropy', epochs = 20, batch size = 64, dropout = 0



Training accuracy: 99.93%
Testing accuracy: 98.76%

Setting 14: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 3, batch size = 64, dropout = 0

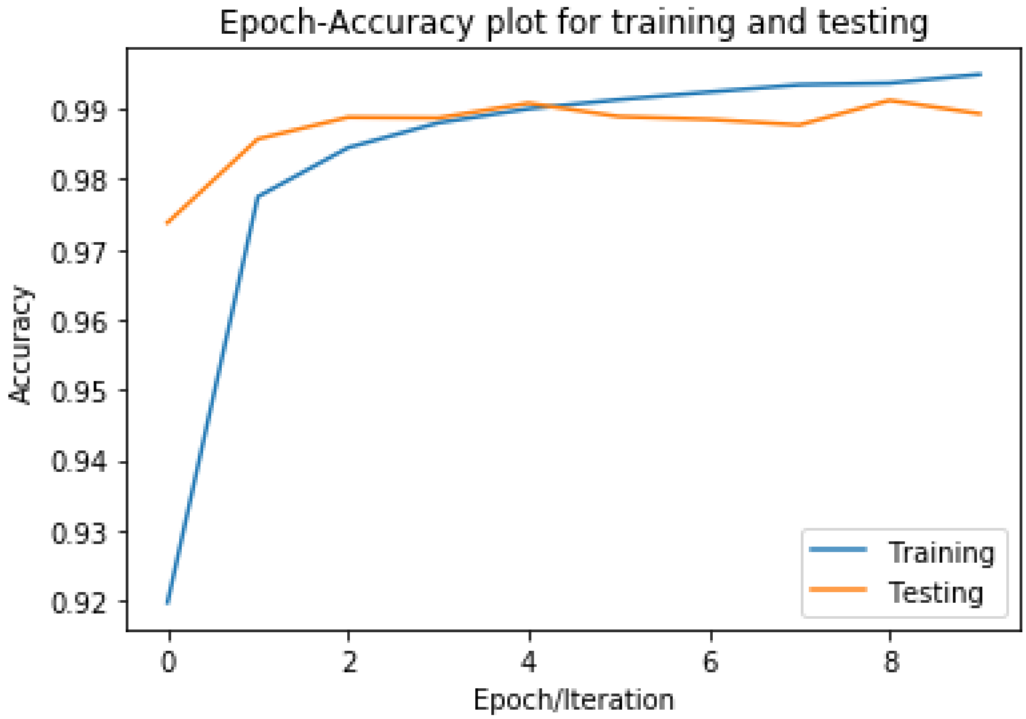
```
The LeNet5 model takes 56.77 seconds to train

dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
10000/10000 [==============================] - 1s 124us/step
LeNet5 architecture Error: 1.4900%
LeNet5 architecture Loss: 0.0444%
```




```
LeNet5 architecture Accuracy: 98.5100%
```

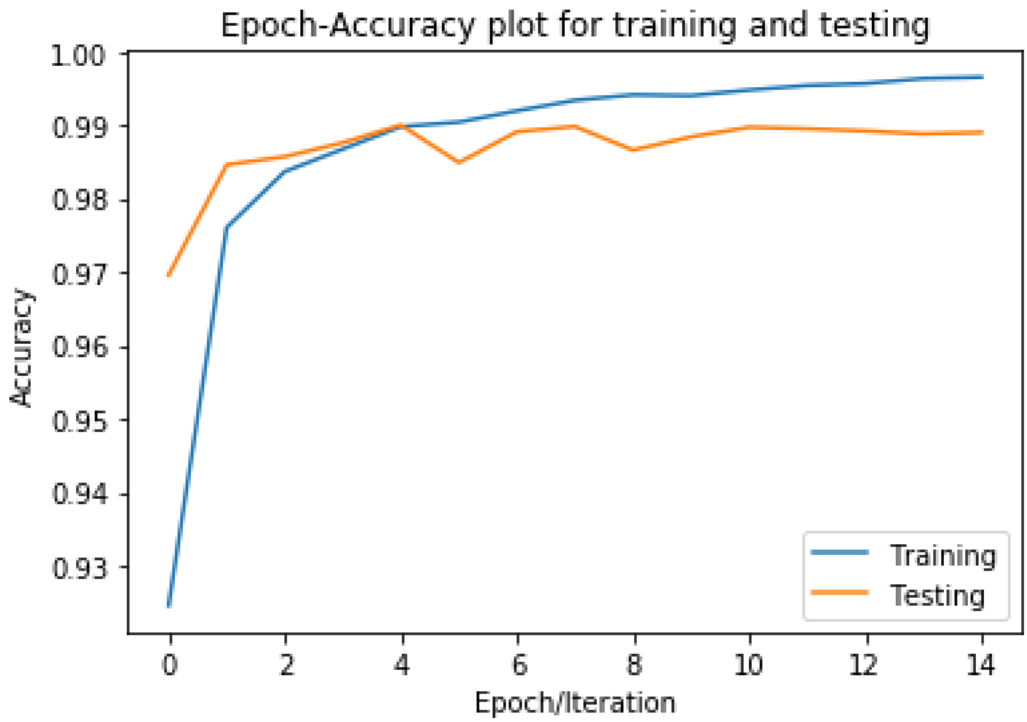
Training accuracy: 98.39%
Testing accuracy: 98.51%

Setting 15: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 10, batch size = 64, dropout = 0



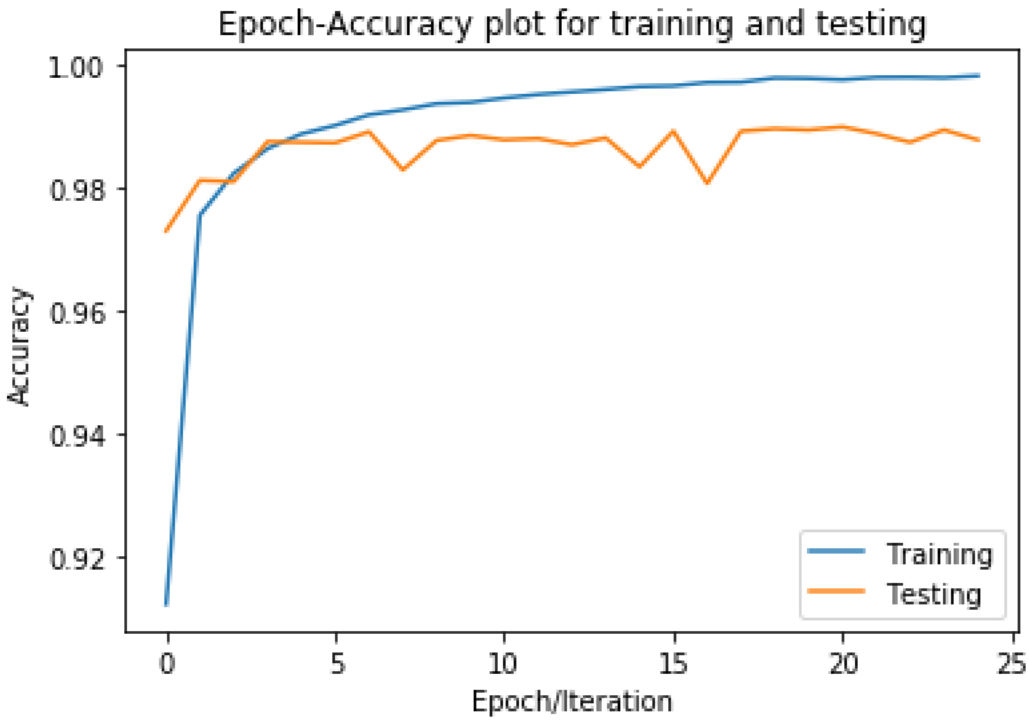
Training accuracy: 99.49%
Testing accuracy: 98.93%

Setting 16: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 15, batch size = 64, dropout = 0



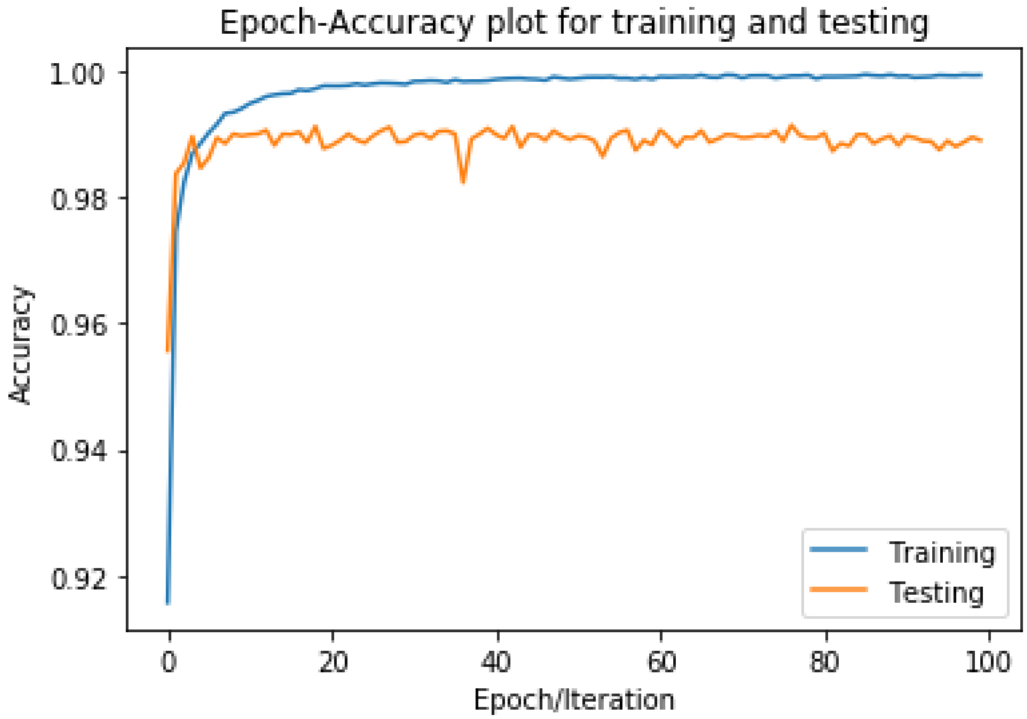
Training accuracy: 99.65%
Testing accuracy: 98.90%

Setting 17: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 25, batch size = 64, dropout = 0



Training accuracy: 99.81%
Testing accuracy: 98.77%

Setting 18: kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 100, batch size = 64, dropout = 0



Training accuracy: 99.94%
Testing accuracy: 98.91%

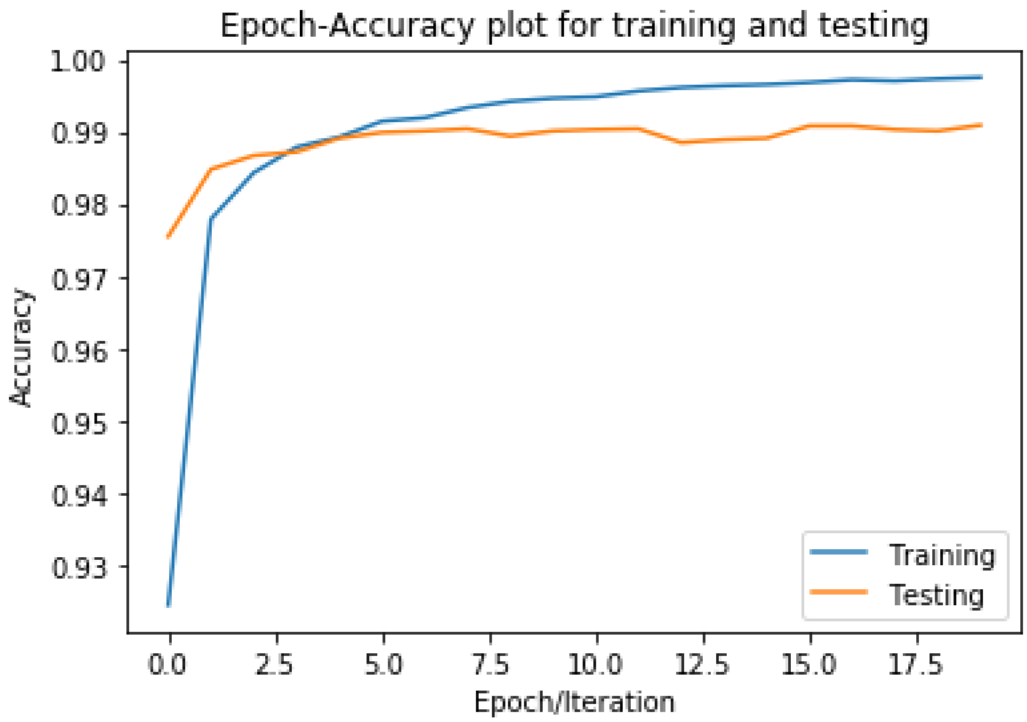
IV. Discussion

I have trained the CNN LeNet5 model on the 60,000 training images and also tested on the 10,000 testing images from the MNIST dataset. I have plotted the graphs showing the model accuracy for the training and testing versus the epoch or iteration number. The accuracy performance curves using the epoch-accuracy (or iteration-accuracy) plot on training and test datasets separately are computed and pasted above. The performance curves under different yet representative parameter settings are also shown for the same. And the mean and variance is calculated.

Relu and leaky Relu work the best for the activation function, different dropouts, number of filters or number of dense layers chosen depends and varies from case to case. For the optimizers, rmsprop and adam work the best. Learning rate for very small values like 0.01 or 0.001 or 0.0001 gives better outputs. Weight initialization must be chosen to be random uniform or glorot uniform, this gives better results as compared to others. Kernel size should be (5,5) and epochs in between 15 to 20 are more than enough to get a higher accuracy than 98%. Dropout may or may not be considered as in this case, even without the dropout, the accuracy is high. The appropriate batch size is 64, can choose 32 or 128 as well and stride should be (1,1).

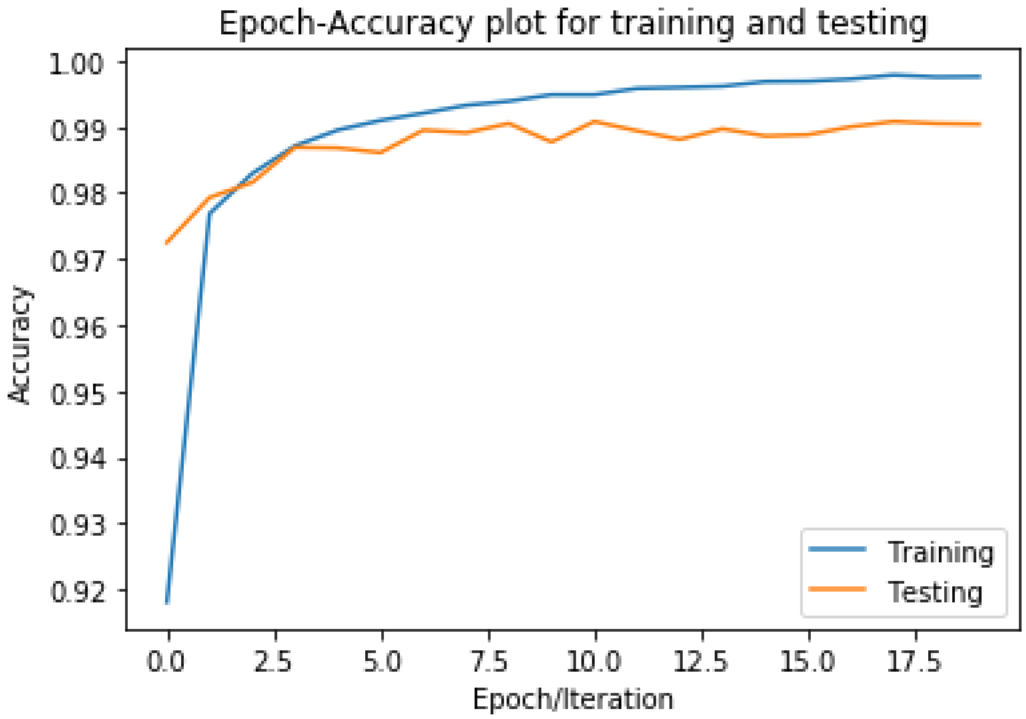
The best parameter settings to achieve the highest accuracy on the test set are: Setting 1 - kernel size = (5,5), strides = (1,1), activation = 'relu', weight initialization = 'random uniform', pool size = (2,2), optimizer = 'rmsprop', learning rate = 0.001, loss = 'categorical crossentropy', epochs = 20, batch size = 64, dropout = 0

Training loss and accuracy – 0.0081 and 99.76%
Testing loss and accuracy – 0.0596 and 99.04%
The performance curves plot for the test set and the training set under this setting is shown below.



```
Training the LeNet5 architecture model
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 20s 329us/step - loss: 0.2648 - acc: 0.9181 - val_loss: 0.0856 - val_acc: 0.9725
Epoch 2/20
60000/60000 [==============================] - 19s 318us/step - loss: 0.0759 - acc: 0.9769 - val_loss: 0.0640 - val_acc: 0.9793
Epoch 3/20
60000/60000 [==============================] - 17s 281us/step - loss: 0.0535 - acc: 0.9829 - val_loss: 0.0622 - val_acc: 0.9816
Epoch 4/20
60000/60000 [==============================] - 18s 299us/step - loss: 0.0417 - acc: 0.9871 - val_loss: 0.0397 - val_acc: 0.9869
Epoch 5/20
60000/60000 [==============================] - 17s 288us/step - loss: 0.0340 - acc: 0.9895 - val_loss: 0.0459 - val_acc: 0.9868
Epoch 6/20
60000/60000 [==============================] - 17s 285us/step - loss: 0.0295 - acc: 0.9910 - val_loss: 0.0466 - val_acc: 0.9861
Epoch 7/20
60000/60000 [==============================] - 15s 258us/step - loss: 0.0257 - acc: 0.9921 - val_loss: 0.0343 - val_acc: 0.9895
Epoch 8/20
60000/60000 [==============================] - 15s 256us/step - loss: 0.0223 - acc: 0.9932 - val_loss: 0.0366 - val_acc: 0.9891
Epoch 9/20
60000/60000 [==============================] - 15s 258us/step - loss: 0.0202 - acc: 0.9939 - val_loss: 0.0385 - val_acc: 0.9905
Epoch 10/20
60000/60000 [==============================] - 16s 269us/step - loss: 0.0174 - acc: 0.9949 - val_loss: 0.0476 - val_acc: 0.9877
Epoch 11/20
60000/60000 [==============================] - 19s 315us/step - loss: 0.0170 - acc: 0.9949 - val_loss: 0.0360 - val_acc: 0.9908
Epoch 12/20
60000/60000 [==============================] - 17s 288us/step - loss: 0.0140 - acc: 0.9958 - val_loss: 0.0428 - val_acc: 0.9894
Epoch 13/20
60000/60000 [==============================] - 17s 283us/step - loss: 0.0132 - acc: 0.9960 - val_loss: 0.0522 - val_acc: 0.9881
Epoch 14/20
60000/60000 [==============================] - 16s 266us/step - loss: 0.0126 - acc: 0.9961 - val_loss: 0.0404 - val_acc: 0.9897
Epoch 15/20
60000/60000 [==============================] - 15s 258us/step - loss: 0.0114 - acc: 0.9968 - val_loss: 0.0529 - val_acc: 0.9886
Epoch 16/20
60000/60000 [==============================] - 15s 258us/step - loss: 0.0103 - acc: 0.9969 - val_loss: 0.0464 - val_acc: 0.9888
Epoch 17/20
60000/60000 [==============================] - 15s 256us/step - loss: 0.0095 - acc: 0.9972 - val_loss: 0.0491 - val_acc: 0.9900
Epoch 18/20
60000/60000 [==============================] - 15s 258us/step - loss: 0.0080 - acc: 0.9978 - val_loss: 0.0559 - val_acc: 0.9908
Epoch 19/20
60000/60000 [==============================] - 15s 257us/step - loss: 0.0079 - acc: 0.9975 - val_loss: 0.0603 - val_acc: 0.9905
Epoch 20/20
60000/60000 [==============================] - 18s 293us/step - loss: 0.0081 - acc: 0.9976 - val_loss: 0.0596 - val_acc: 0.9904
```

```
The LeNet5 model takes 334.66 seconds to train

dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
10000/10000 [==============================] - 1s 134us/step
LeNet5 architecture Error for testing: 0.9600%
LeNet5 architecture Loss for testing: 0.0596%
```




```
LeNet5 architecture Accuracy for testing: 99.0400%
```

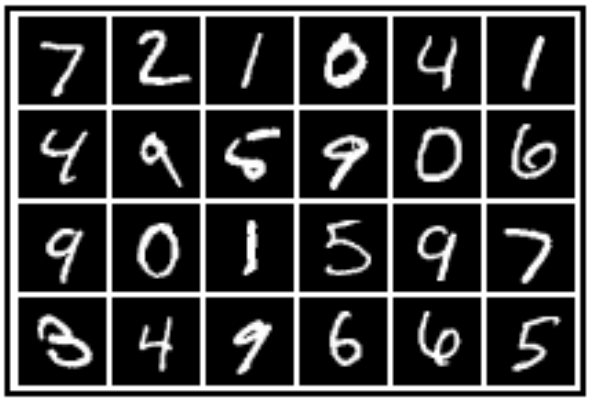
The mean and variance of the training accuracy are 99.17% and 0.867 respectively.
The mean and variance of the testing accuracy are 98.70% and 0.118 respectively.

(c) Apply trained network to negative images

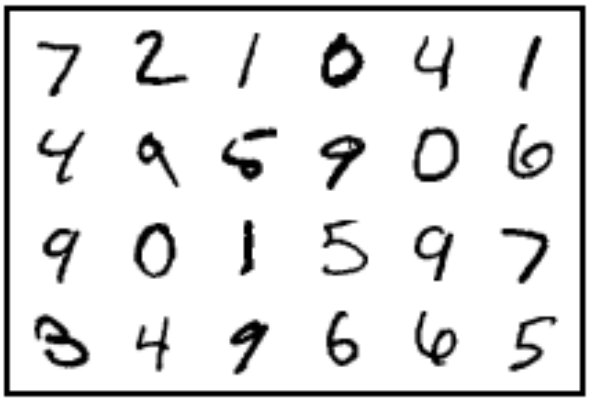
I. <u>Abstract and Motivation</u>

(Source: Lecture by Prof. Jay Kuo dated Feb, Mar 2019)

Training the network on MNIST with original black background and white foreground and then using the same network on the negative of these images (flip around the images) with the white background and black foreground, it gives the following results for testing.



Original



Negative

Test Performance of LeNet-5

- Original: 98.94% (trained by original)
- Negative: 37.36% (trained by original)

Test Performance of LeNet-5

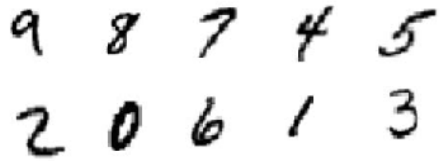
- Original: 37.36% (trained by negative)
- Negative: 98.94% (trained by negative)

The accuracy is too low in the second case. This shows that for LeNet5 when train a network it is really sensitive to the input. To handle both the kind of data shown above, to get better results and accuracy, one solution is to train both the datasets. But also filter number has to be more. More filters are needed to train both the datasets together.

Comparison of LeNet5 initializations

Random Initialization

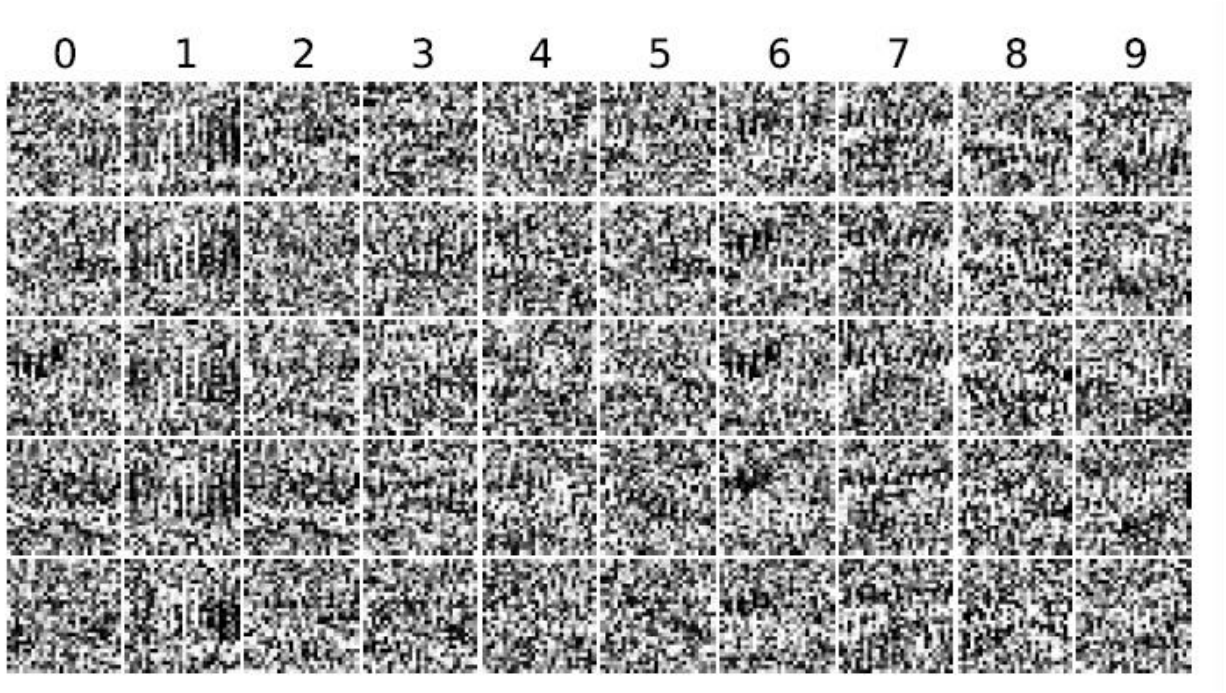
K-Means Initialization



- Say random initialize the weights, then go through the whole process and architecture as discussed above, eventually find the probability i.e. output, there is some output feature vector. Then cluster the feature vector and find the centroid and find the image closest to the centroid and see what it is. Here everything is random.
- Now change the strategy and for each small input patch, consider the k means clustering first. Cluster the input 5x5 into say 10 clusters and find the centroid and use the centroid as the filter weight. So in the next stage collect the input and do the k means clustering and then use the centroid as the weight. So each weight determines the output from the previous stage. Here use the k means to define the filter. And then do the matched filter using the input distribution and then find cluster and centroid. So find the matched filter for that weight.

In the output we observe, that in random initialization, there are other numbers repeated, however, in the k means initialization, we get all the numbers from 0 to 9. This is because there is a self-organizing property of the network. Even if there is no label or back propagation. Just do the feed forward, initialize the filter weights according to k means.

There are some flaws in CNN, one issue is robustness, it gets easily fooled by adversarial perturbation.



The CNN has 99.99% confidence in recognizing the images to be the digit in the top row.

Another reason is scalability. We need to retrain the network from scratch is the scale of the dataset is varied. Say example if add or delete one class of images or add more training samples.

Another problem is portability. Say train the network for one domain and apply to another domain, it does not do a good job.

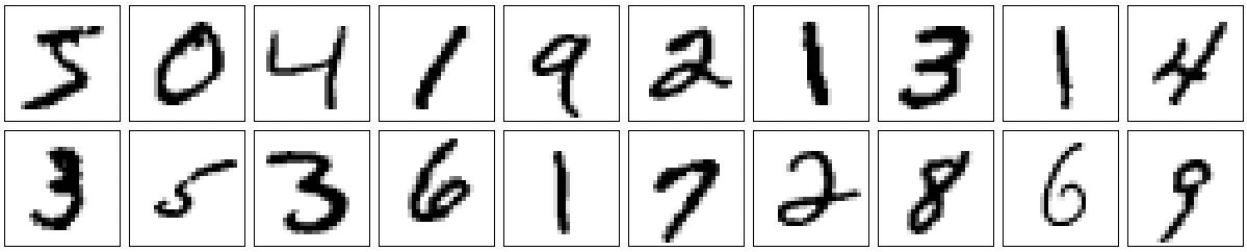
The reason for the above is feature extraction is not invertible. Critical information is lost, need to understand the information loss.

In this part of the problem, we try the different parameters and hyperparameters settings as done in part(b). However, this time the training data consists of 1,20,000 samples and the testing data contains 20,000 samples. This is because the dataset contains the original MNIST dataset as well as the negative of the images present in the dataset. So both the positive and negative are added in the dataset for training and testing and the model is tested for various parameters to get the best accuracy. One method of this is using data augmentation to prepare the training and testing dataset containing both the positive and negative images.

II. Approach and Procedure

We may achieve good recognition performance on the MNIST dataset in Problem 1. Does the LeNet-5 understands the handwritten digits as well as human beings? One test is to provide a negative of each test image as shown in the figure below, where the value of the negative image at pixel *(x,y)*, denoted by *r(x,y)*, is computed via *r(x,y)=255-p(x,y)*, where *p(x,y)* is the value of the original image at the same location. Humans have no difficulty in recognizing digits of both types. How about the LeNet-5?

1) Report the accuracy on the negative test images using the LeNet-5 trained in part (b). Discuss the result.
2) Design and train a new network that can recognize both original and negative images from the MNIST test dataset. Test the proposed network, report the accuracy and make discussion.



Sample images from the negatives of the MNIST dataset

(Source: Discussion lecture dated 03/19/2019)

Part(c) is basically about getting the negative images of MNIST and then need to propose the model that makes it work for both the negative and the original images. And need to get 98% accuracy for this part as well.

For part(c).2, no need to change in network architecture. It is okay to do data processing/augmentation only.

Algorithm implemented in Keras (Python):

- Split the dataset into the training and testing sets
- Load the training data, validation data and testing data from MNIST dataset
- Load the training data, returns in numpy array form
- Load the validation data, returns in numpy array form
- Load the testing data, returns in numpy array form
- Take the negative of the above data and store in numpy array

- Now combine the original and negative dataset for the training and testing
- Use this combined set of original and negative images data for the training and testing of the model
- Define the LeNet5 architecture network model
- Create a sequential model
- Add two convolution and max pooling layers in order and one flatten layer and three fully connected dense layers
- Define the optimizer and loss and metrics
- Compile the sequential model
- Build the sequential model
- Print the information of the model
- Train and fit the model defining epochs and batch size
- List all data in history
- Perform final evaluation of the lenet5 sequential model
- Printing the error, loss and accuracy for the model after the evaluation
- The plot model history function plots the graphs showing loss and accuracy vs epoch/iteration
- Summarize history for accuracy

Model parameters:

Batch size – The weights are updated in the network model for every batch size. Larger the batch size, slower will be the convergence. This is because, using different batch size for the training, it causes decrease in the variance of the filter weight change. It does not affect the accuracy so much.

Dropout- This is used to reduce overfitting of the data. It is used after the pooling layer followed by the convolution layer. 0.5 works good.

Learning rate – It is used to state the time for convergence of the model. this value should be small, to provide a better model. If the value is high, then the convergence occurs fast taking bigger steps and sometimes may lead to a wrong local minimum. Thus the value being small like 0.001 takes smaller steps to convergence and gives a good global minimum. However, it really depends on the type of data dealing with.

Kernel size – This is used to determine the size of the kernel window chosen for the correlation of the filter and the data. It should not be too big. I tried different filter size like (3,3), (5,5) and (7,7). Better output was obtained for smaller size (3,3). This is because a smaller window can capture small details and not let the patch be

generalized for different type of data. This helps to use these details for image matching in a way which helps to get accurate output.

Pool size – This is the size of the window in which the elements go through max pooling operation where the highest value among the elements is chosen and replaced. The window size should not be too large.

Epochs – This denotes the number of iterations carried out for the network. It can vary from 3 to 200 or more. However, for the MNIST network, it can be on an average chosen to be 15 or 20. For low learning rates, the epoch is usually higher.

Activation function – This parameter depends on the model and usually in real life application, relu and leaky relu work the best. Even for our model it is the same. The curve for relu helps to obtain a better output, by getting better convergence.

Optimizers and loss function – These can be chosen depending on the network. For our MNIST model, adam works good as well as sgd, and the loss function we chose is categorical crossentropy, this is because at the output we have 10 different classes, so this is chosen for the multi-class problem. Adam and rmsprop optimizer gives higher convergence.

Number of filters – Greater number of filters give better accuracy; this is due to more feature extraction with more number of filters. This improves the feature efficiency. The number of filters was varied from 6, 16 to 32, 64.

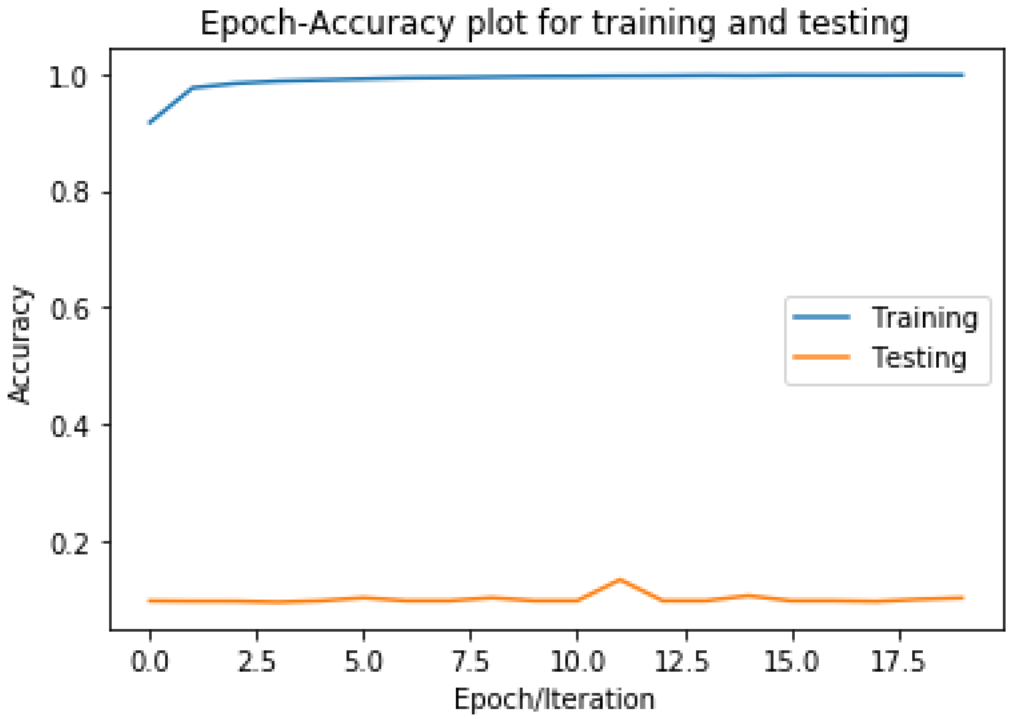
III. Experimental Results

After training the model for the original data and testing the network on the negative data, I got the following output. This is done with the best settings I got in prob (b).

Setting 1 - kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0

```
Training the LeNet5 architecture model
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 17s 288us/step - loss: 0.2688 - acc: 0.9163 - val_loss: 14.5385 - val_acc: 0.0980
Epoch 2/20
60000/60000 [==============================] - 16s 268us/step - loss: 0.0790 - acc: 0.9756 - val_loss: 2.3167 - val_acc: 0.0974
Epoch 3/20
60000/60000 [==============================] - 15s 255us/step - loss: 0.0531 - acc: 0.9832 - val_loss: 2.3240 - val_acc: 0.0974
Epoch 4/20
60000/60000 [==============================] - 15s 257us/step - loss: 0.0417 - acc: 0.9872 - val_loss: 13.1837 - val_acc: 0.0958
Epoch 5/20
60000/60000 [==============================] - 15s 255us/step - loss: 0.0344 - acc: 0.9890 - val_loss: 14.4984 - val_acc: 0.0982
Epoch 6/20
60000/60000 [==============================] - 16s 259us/step - loss: 0.0292 - acc: 0.9909 - val_loss: 14.4547 - val_acc: 0.1032
Epoch 7/20
60000/60000 [==============================] - 18s 303us/step - loss: 0.0241 - acc: 0.9927 - val_loss: 14.5353 - val_acc: 0.0982
Epoch 8/20
60000/60000 [==============================] - 17s 282us/step - loss: 0.0218 - acc: 0.9933 - val_loss: 13.6640 - val_acc: 0.0982
Epoch 9/20
60000/60000 [==============================] - 17s 280us/step - loss: 0.0190 - acc: 0.9942 - val_loss: 14.4547 - val_acc: 0.1032
Epoch 10/20
60000/60000 [==============================] - 17s 276us/step - loss: 0.0159 - acc: 0.9950 - val_loss: 14.5353 - val_acc: 0.0982
Epoch 11/20
60000/60000 [==============================] - 16s 268us/step - loss: 0.0149 - acc: 0.9954 - val_loss: 14.5353 - val_acc: 0.0982
Epoch 12/20
60000/60000 [==============================] - 17s 289us/step - loss: 0.0137 - acc: 0.9960 - val_loss: 11.4851 - val_acc: 0.1342
Epoch 13/20
60000/60000 [==============================] - 17s 289us/step - loss: 0.0130 - acc: 0.9962 - val_loss: 14.5353 - val_acc: 0.0982
Epoch 14/20
60000/60000 [==============================] - 20s 328us/step - loss: 0.0115 - acc: 0.9969 - val_loss: 14.5353 - val_acc: 0.0982
Epoch 15/20
60000/60000 [==============================] - 17s 287us/step - loss: 0.0115 - acc: 0.9966 - val_loss: 13.2321 - val_acc: 0.1067
Epoch 16/20
60000/60000 [==============================] - 16s 263us/step - loss: 0.0088 - acc: 0.9974 - val_loss: 14.5353 - val_acc: 0.0982
Epoch 17/20
60000/60000 [==============================] - 15s 253us/step - loss: 0.0095 - acc: 0.9974 - val_loss: 14.5353 - val_acc: 0.0982
Epoch 18/20
60000/60000 [==============================] - 20s 330us/step - loss: 0.0093 - acc: 0.9973 - val_loss: 14.1060 - val_acc: 0.0971
Epoch 19/20
60000/60000 [==============================] - 22s 363us/step - loss: 0.0081 - acc: 0.9977 - val_loss: 13.1013 - val_acc: 0.1006
Epoch 20/20
60000/60000 [==============================] - 15s 254us/step - loss: 0.0093 - acc: 0.9975 - val_loss: 13.2562 - val_acc: 0.1030
```

```
The LeNet5 model takes 339.08 seconds to train

dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
10000/10000 [==============================] - 1s 115us/step
LeNet5 architecture Error: 89.7000%
LeNet5 architecture Loss: 13.2562%
```




```
LeNet5 architecture Accuracy: 10.3000%
```

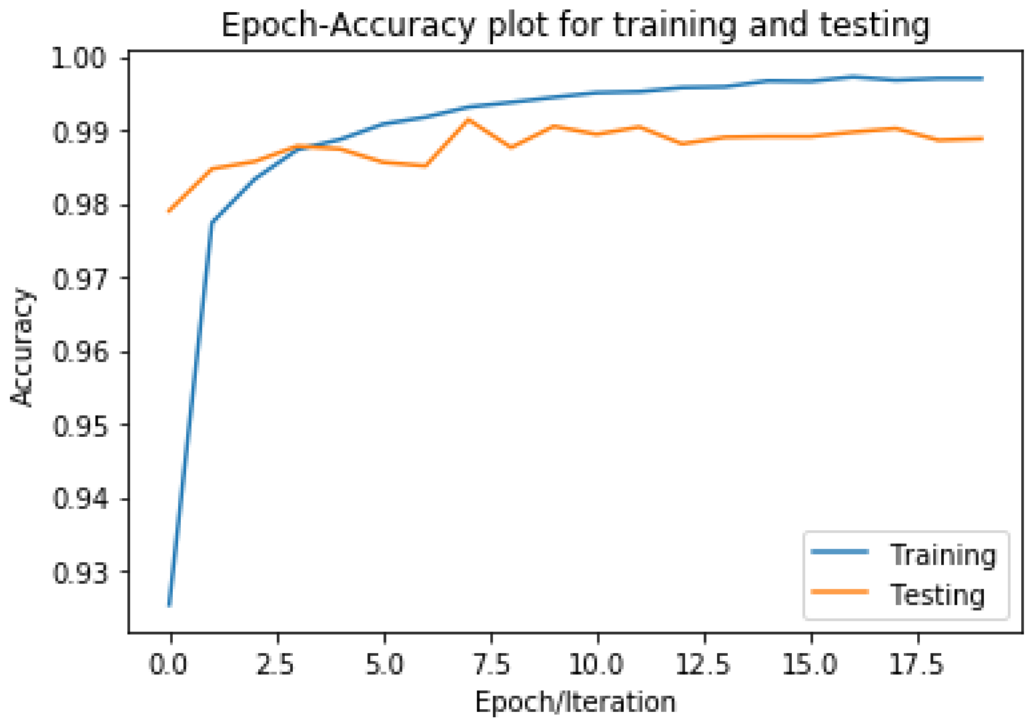
Training accuracy and loss: 99.75% and 0.0093
Testing accuracy and loss: 10.30% and 13.2562

We see above that the accuracy was very low. This is because the model is not robust and can work in an accurate manner when the testing data is similar to the training data, however, when the testing data is different than provided in the training data, we get very low accuracy. The model does not work in a good fashion.

When the training and testing both includes both the original and negative (inverted) images dataset, the model network is designed and trained in such a manner that it can recognize both the original and negative (inverted) images from the MNIST dataset. This is done with the best settings I got in prob (b).

Setting 1 - kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0

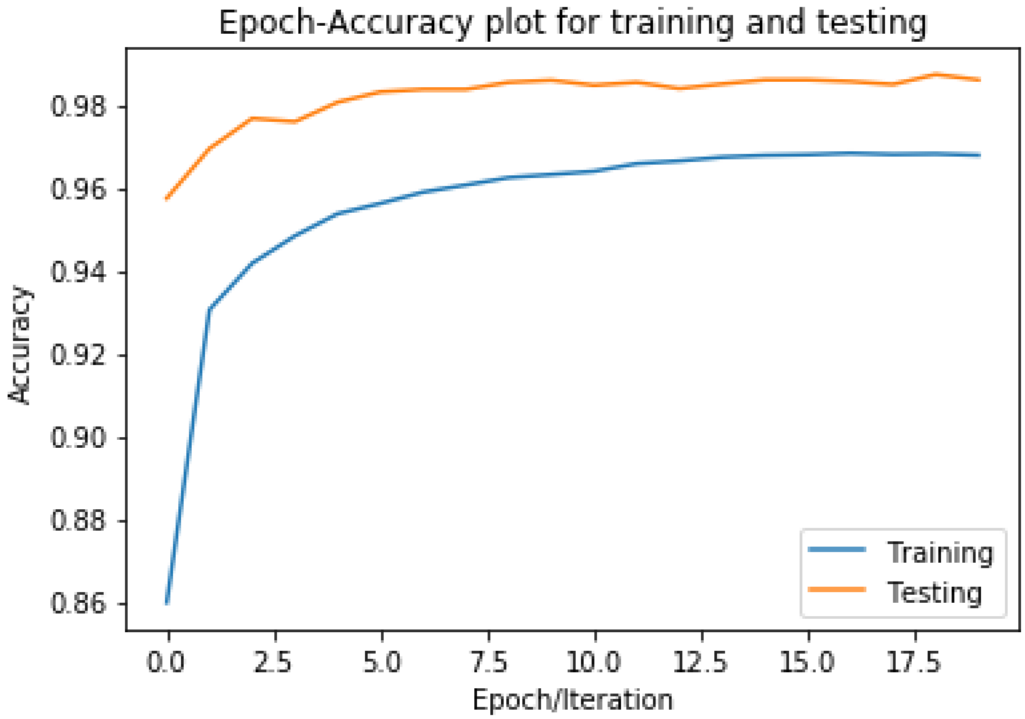
```
The LeNet5 model takes 368.81 seconds to train

dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
10000/10000 [==============================] - 2s 158us/step
LeNet5 architecture Error for testing: 1.1200%
LeNet5 architecture Loss for testing: 0.0663%
```




```
LeNet5 architecture Accuracy for testing: 98.8800%
```

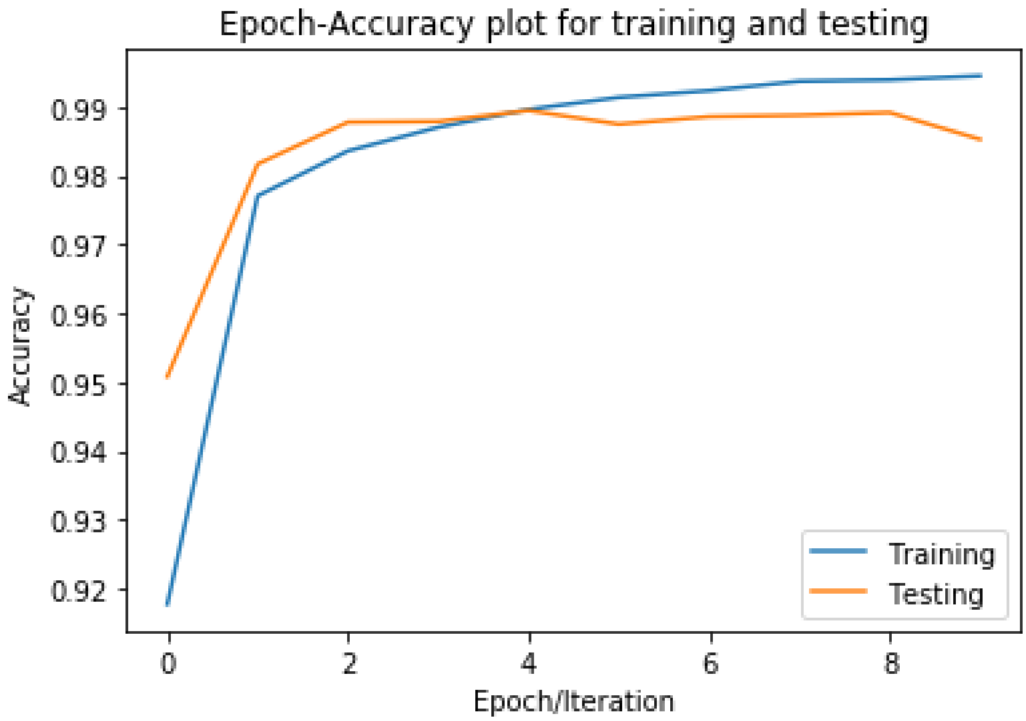
Training accuracy and loss: 99.70% and 0.0103
Testing accuracy and loss: 98.88% and 0.0663

Setting 2 - kernel size = (5,5), strides = (1,1), activation = ‘sigmoid’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘sgd’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0.5



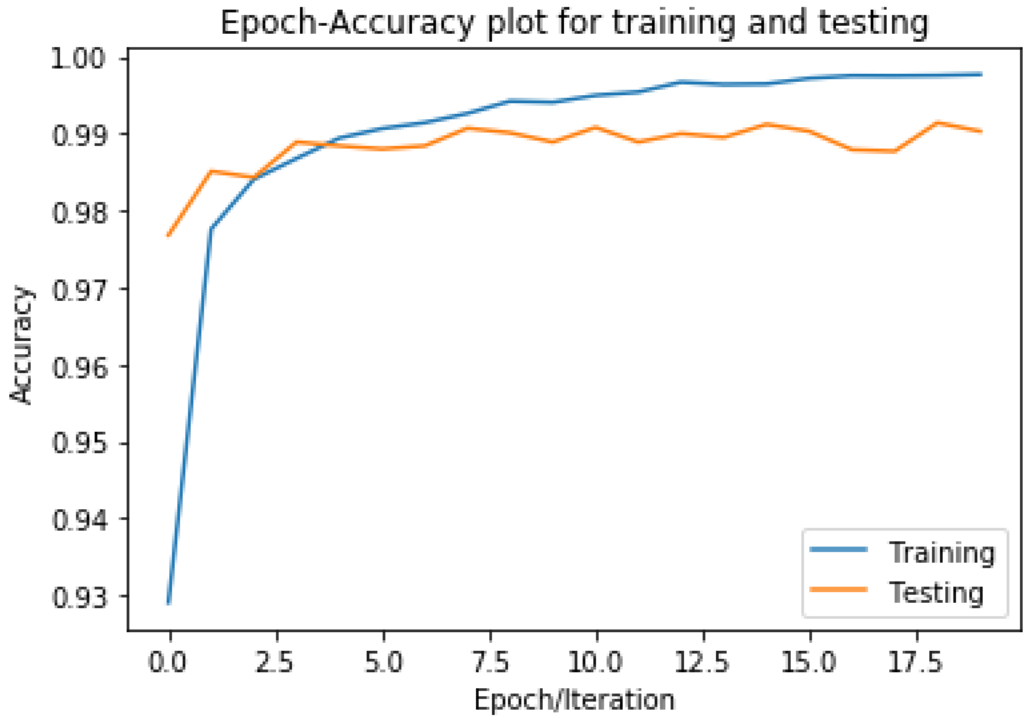
Training accuracy: 97.12%
Testing accuracy: 98.60%

Setting 3 - kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘random uniform’, pool size = (2,2), optimizer = ‘rmsprop’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 10, batch size = 64, dropout = 0



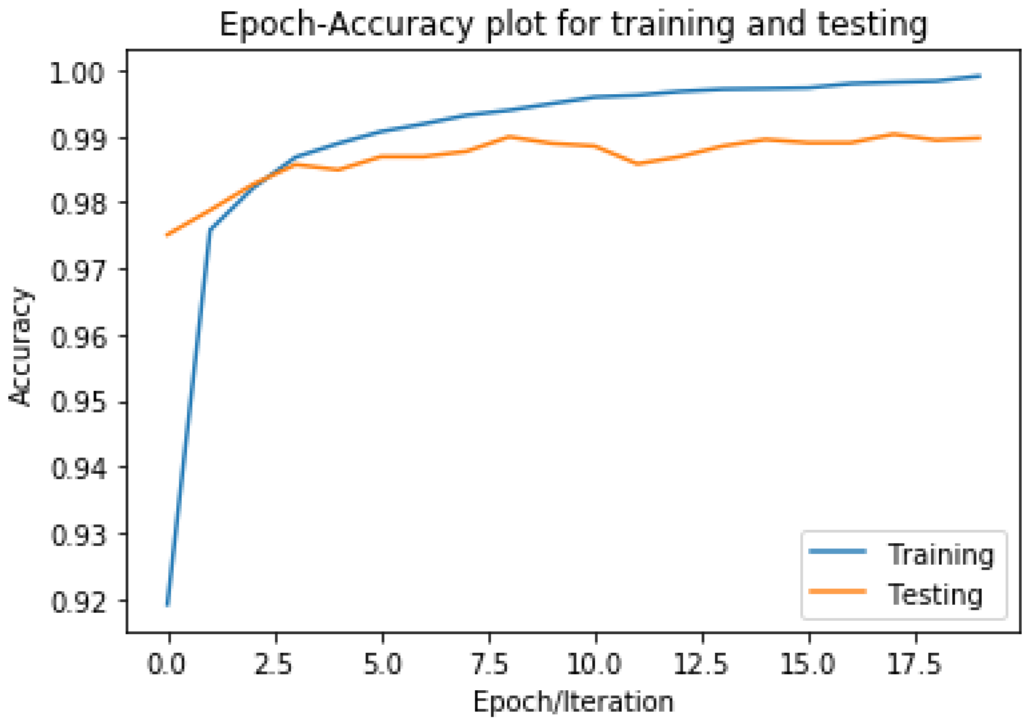
Training accuracy: 99.45%
Testing accuracy: 98.53%

Setting 4 – kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘glorot uniform’, pool size = (2,2), optimizer = ‘adam’, learning rate = 0.001, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0.10



Training accuracy and loss: 99.77% and 0.0070
Testing accuracy and loss: 99.03% and 0.0419

Setting 5 - kernel size = (5,5), strides = (1,1), activation = ‘relu’, weight initialization = ‘glorat uniform’, pool size = (2,2), optimizer = ‘sgd’, learning rate = 0.01, loss = ‘categorical crossentropy’, epochs = 20, batch size = 64, dropout = 0.25



Training accuracy and loss: 99.91% and 0.0034
Testing accuracy and loss: 98.97% and 0.0491

IV. Discussion

I have trained the CNN LeNet5 model on the 70,000 training images and also tested on the 20,000 testing images from the MNIST dataset. I have plotted the graphs showing the model accuracy for the training and testing versus the epoch or iteration number. The accuracy performance curves using the epoch-accuracy (or iteration-accuracy) plot on training and test datasets separately are computed and pasted above. The performance curves under different yet representative parameter settings are also shown for the same. And the mean and variance is calculated.

Relu and leaky Relu work the best for the activation function, different dropouts, number of filters or number of dense layers chosen depends and varies from case to case. For the optimizers, adam and sgd work the best. Learning rate for very small values gives better outputs. Weight initialization must be chosen to be glorot uniform or sometimes random uniform, this gives better results as compared to others. Kernel size should be (5,5) and epochs in between 15 to 20 are more than enough to get a higher accuracy than 98%. Dropout may be considered as in this case like 0.25 or 0.5 to avoid the overfitting problem. The appropriate batch size is 64 and stride should be (1,1). Also, the filter size was varied for various layers to get the output for more accuracy. It was increased as the number of features also increased.

The accuracy on the negative test images using the LeNet-5 trained in part b) is quite bad. This is because the LeNet 5 model we designed was trained on MNIST datasets and for the testing the negative of the images in the MNIST dataset was provided which led to decrease in the accuracy drastically. This happened because the CNN is not so flexible and not robust. The main reason being that for the training the images provided to CNN were with black background and white foreground whereas for the testing the images had white background and black foreground. Since the images were from different domain, and did not have similar color features, the CNN could not recognize the numbers in the negative image and performed poorly. This is because of it's dependency on the pixel values of the image. This dependency can't change so easily and thus we need to adjust the training parameters model accordingly to get better flexibility and elasticity in the CNN, making the model robust for operation on negative images as well.

Designed and trained a new network that can recognize both original and negative images from the MNIST test dataset, by providing both the original and negative images in the training. Tested the proposed network and reported the accuracy. This new model which was applied gave a high accuracy of more than 98% and also gave the plot of epoch vs accuracy showing 98.88% which is quite good.

The mean and variance of the training accuracy are 99.19% and 1.093 respectively. The mean and variance of the testing accuracy are 98.802% and 0.040 respectively.