**Abstract**

The goal of this project is to implement a Convolution Neural Network (CNN) on the fashion-mnist dataset. Although the idea of this project is on a very rudimentary level, we are going to concentrate on achieving considerable performance by tuning the network using different hyperparameters.

Fashion dataset is a visual representation of all the fashion dataset, similar to handwritten letter mnist dataset. CNN’s belong to a group of neural networks that handles visual data such as images and videos.  We are going to train the network with different biases and including dropouts to check for performance variations. The results are reported in intuitive charts using different plotting libraries available in Python.

The Deep Learning libraries such as TensorFlow and Keras are available which does the heavy lifting of fast computations using matrix operations which will bring down the time required for training the network. Optionally we include use of GPU’s for faster computation times. As far as the development environment is considered, we are going to use Python in Jupyter Notebook.

**Introduction**

Neural nets were used from the dawn of artificial intelligence. At least that is how we think. The simplest definition of a neural network, more properly referred to as an ‘artificial’ neural network (ANN), is provided by the inventor of one of the first neurocomputers, Dr. Robert Hecht-Nielsen. He defines a neural network as a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.

The design of Artificial Neural Networks (ANNs) is inspired by the animal brain. Although not as powerful as the brain (yet), artificial neural networks are the most powerful learning models in the field of machine learning.

Some problems in real life cannot be trained on traditional machine learning algorithms such as speech recognition, image recognition etc. There are special architectures specially designed for those. One such architecture is Convolution Neural Networks.

In this project, we are going to train a CNN network on top of fashion-mnist dataset. With further additions we are going to evaluate the performance analysis and importance of hyperparameters in the network. There will be nearly 7 experiments using varying hyper parameters to gain better understanding of how CNN works and its importance.

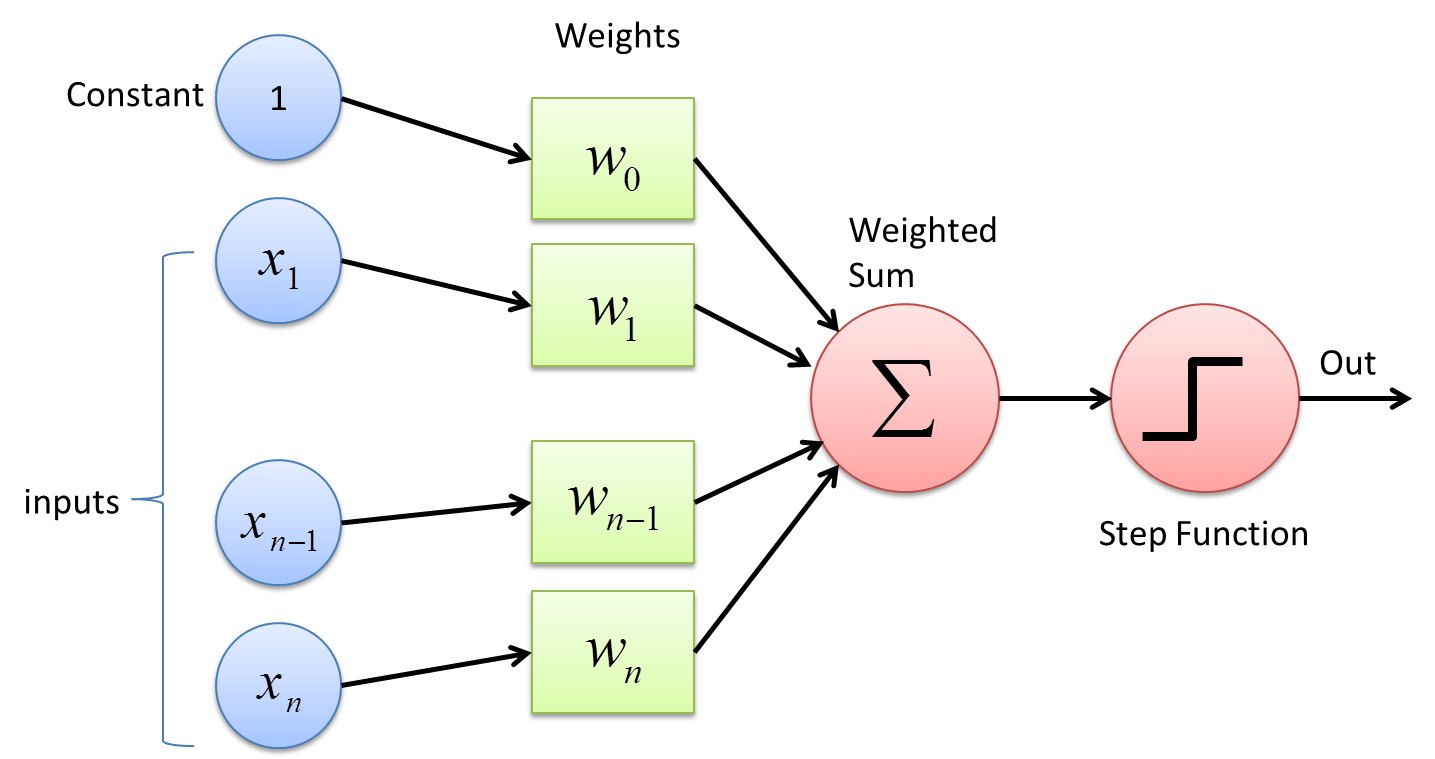
**Neural Networks**

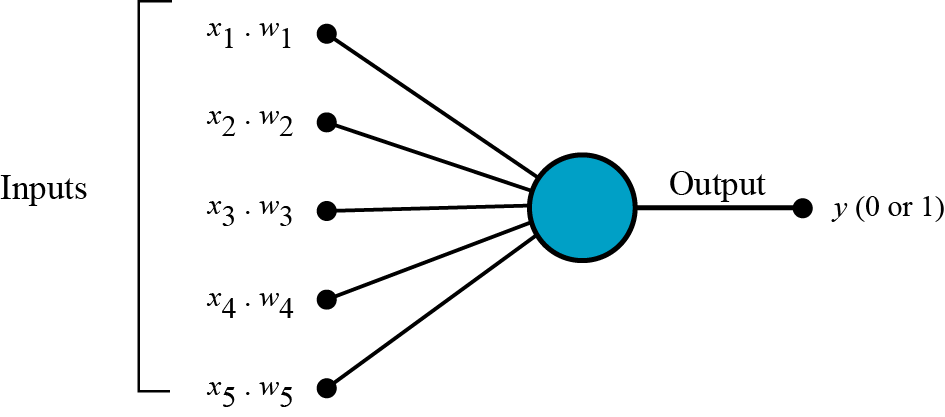
The first neural nets, also called as Artificial Neural Networks or ANNs are inspired by the structure of human brain. These are considered as extension of machine learning models, but these emerged as separate field of study. The progression of neural networks have enabled the machines to do complex tasks such as object and image recognition, automatic speech recognition (ASR), machine translation, video analysis and classification among others. These are called deep learning models.

In layman’s terms, Artificial Neural Networks are a collection of large number of simple devices called artificial neurons. These are functionally little similar to neurons we find in human brain. This so called artificial neuron learns from the input and we activate it if certain condition is matched, depends on the use case we are working on. These are trained to inhibit or amplify the input signals in order to perform certain tasks. These are called perceptron.

A perceptron takes a weighted sum of multiple inputs with different weights along with the bias and applies a function (mostly a step function) and return the binary output i.e. 1 if the input is positive and 0 if the input is negative.

Perceptrons can also be used for multi class classification. Only difference is we need multiple perceptrons. The network of perceptrons can act as universal function approximator. Typically for not such complex programs, we use AND or OR gate on the outputs of perceptrons to get a multi class outputs.





As far as input types are considered, we use arrays or matrices of data. In case of text data, we use one-hot vector or word embeddings. Images are usually represented in matrices. These numbers are the raw pixels of the image. In a neural network, each pixel of the input image is a feature. For example, the image of a MNIST dataset is an 18 x 18 array. Hence, it will be fed as a vector of size 324 to the network. If it is a RGB image, then we have 3 colours, so it is 324\*3 pixels/neurons. The output is generally a SoftMax output in case of MNIST dataset. It is a multiclass logistic function commonly used to compute the probability of an input belonging to one of the multiple classes.

Neural networks need rigorous training with different parameters called hyperparameters. Weights and biases are the hyperparameters in case of neural networks. During training, the neural network learning algorithm fits various models to the training data and selects the best model for prediction. The learning algorithm is trained with a fixed set of hyperparameters - the network structure (number of layers, number of neurons in the input, hidden and output layers etc.). It is trained on the weights and the biases, which are the parameters of the network.

There are different activation functions used in the neural nets. All these have different properties. In choosing these activation functions, we need to make sure that they should be smooth and must inject non-linearity between input and output.

1. Logistic function
2. Hyperbolic tangent function
3. Rectilinear Unit (RELU)

**Convolution Neural Networks**

Convolutional Neural Networks, or CNNs, are neural networks specialised to work with visual data, i.e. images and videos (though not restricted to them). They are very similar to the vanilla neural networks (multilayer perceptrons) - every neuron in one layer is connected to every neuron in the next layer, they follow the same general principles of forward and backpropagation, etc. However, there are certain features of CNNs that make them perform extremely well on image processing tasks.

The challenges in visual recognition are

1. Viewpoint variation: Different orientations of the image with respect to the camera.
2. Scale variation: Different sizes of the object with respect to the image size.
3. Illumination conditions: Illumination effects.
4. Background clutter: Varying backgrounds.

Although the vanilla neural networks (MLPs) can learn extremely complex functions, their architecture does not exploit what we know about how the brain reads and processes images. For this reason, although MLPs are successful in solving many complex problems, they haven't been able to achieve any major breakthroughs in the image processing domain.

On the other hand, the architecture of CNNs uses many of the working principles of the animal visual system and thus they have been able to achieve extraordinary results in image-related learning tasks.



**The ImageNet Challenge**

CNNs had first demonstrated their extraordinary performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).  The ILSVRC uses a list of about 1000 image categories or "classes" and has about 1.2 million training images. The original challenge is an image classification task.

Applications of CNNs are

1. Object localization -Identifying the local region of the objects (as a rectangular area) and classifying them.
2. Semantic segmentation: Identifying the exact shapes of the objects (pixel by pixel) and classifying them.
3. Optical Character Recognition (OCR): Recognise characters in an image.

There are three main concepts you will study in CNNs:

1. Convolution, and why it 'shrinks' the size of the input image
2. Pooling layers
3. Feature maps

Convolution is the first layer to extract features from an input image. Convolution preserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs such as image matrix and a filter or kernel.

The VGGNet was specially designed for the ImageNet challenge which is a classification task with 1000 categories. Thus, the SoftMax layer at the end has 1000 categories. The blue layers are the convolutional layers while the yellow ones are pooling layers. You will study each one of them shortly.

 Finally, the green layer is a fully connected layer with 4096 neurons, the output from which is a vector of size 4096.

The most important point to notice is that the network acts as a feature extractor for images. For example, the CNN above extracts a 4096-dimensional feature vector representing each input image. In this case, the feature vector is fed to a SoftMax layer for classification, but you can use the feature vector to do other tasks as well (such as video analysis, object detection, image segmentation etc.).

Mathematically, the convolution operation is the summation of the element-wise product of two matrices.

**Strides**

Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on.

**Padding**

Sometimes filter does not fit perfectly fit the input image. We have two options:

1. Pad the picture with zeros (zero-padding) so that it fits
2. Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.

A feature map is a collection of multiple neurons each of which looks at different regions of the input with the same weights. All neurons in a feature map extract the same feature (but from different regions of the input). It is called a 'feature map' because it is a mapping of where a certain feature is found in the image.

After extracting features (as feature maps), CNNs typically aggregate these features using the pooling layer. Let's see how the pooling layer works and how it is useful in extracting higher-level features.

Pooling tries to figure out whether a particular region in the image has the feature we are interested in or not. It essentially looks at larger regions (having multiple patches) of the image and captures an aggregate statistic (max, average etc.) of each region. In other words, it makes the network invariant to local transformations.

The two most popular aggregate functions used in pooling are 'max' and 'average'. The intuition behind these are as follows:

1. Max pooling: If any one of the patches says something strongly about the presence of a certain feature, then the pooling layer counts that feature as 'detected'.
2. Average pooling: If one patch says something very firmly but the other ones disagree,  the pooling layer takes the average to find out.

Pooling has the advantage of making the representation more compact by reducing the spatial size (height and width) of the feature maps, thereby reducing the number of parameters to be learnt. On the other hand, it also loses a lot of information, which is often considered a potential disadvantage. Having said that, pooling has empirically proven to improve the performance of most deep CNNs.

**AlexNet and VGGNet**

The AlexNet was one of the very first architectures to achieve extraordinary results in the ImageNet competition (with about a 17% error rate). It had used 8 layers (5 convolutional and 3 fully connected). One distinct feature of AlexNet was that it had used various kernels of large sizes such as (11, 11), (5, 5), etc. Also, AlexNet was the first to use dropouts, which were quite recent back then.

You are already familiar with VGGNet from the previous session. Recollect that the VGGNet has used all filters of the same size (3, 3) and had more layers (The VGG-16 had 16 layers with trainable weights, VGG-19 had 19 layers etc.).

The VGGNet had succeeded AlexNet in the ImageNet challenge by reducing the error rate from about 17% to less than 8%. Let's compare the architectures of both the nets.

The key idea in moving from AlexNet to VGGNet was to increase the depth of the network by using smaller filters. VGGNet is able to use a higher number of non-linear activations with a reduced number of parameters.

GoogleNet had increased the depth using a new type of convolution technique using the Inception module. Important features of the GoogleNet architecture are:

1. Inception modules stacked on top of each other, total 22 layers
2. Use of 1 x 1 convolutions in the modules
3. Parallel convolutions by multiple filters (1x1, 3x3, 5x5)
4. Pooling operation of size (3x3)
5. No FC layer, except for the last SoftMax layer for classification
6. Number of parameters reduced from 60 million (AlexNet) to 4 million

A close up of text on a black background

Description automatically generated

The key motivator for the ResNet architecture was the observation that, empirically, adding more layers was not improving the results monotonically.  This was counterintuitive because a network with n + 1 layers should be able to learn at least what a network with n layers could learn, plus something more.

The ResNet team [(Kaiming He et al) came up with a novel architecture](https://arxiv.org/pdf/1512.03385.pdf)with skip connections which enabled them to train networks as deep as 152 layers. The ResNet achieved ground-breaking results across several competitions - a 3.57% error rate on the ImageNet and the first position in many other ILSVRC and [COCO object detection](http://cocodataset.org/#home) competitions.

The skip connection mechanism was the key feature of the ResNet which enabled the training of very deep networks. Some other key features of the ResNet are summarised below.

1. ILSVRC’15 classification winner (3.57% top 5 error)
2. 152 layer model for ImageNet
3. Has other variants also (with 35, 50, 101 layers)
4. Every 'residual block' has two 3x3 convolution layers
5. No FC layer, except one last 1000 FC SoftMax layer for classification
6. Global average pooling layer after the last convolution
7. Batch Normalization after every convolution layer
8. SGD + momentum (0.9)
9. No dropout used

**Transfer Learning**

Transfer learning is the practice of reusing the skills learnt from solving one problem to learn to solve a new, related problem.  Before diving into how to do transfer learning, let's first look at some practical reasons to do transfer learning in the first place.

1. Data abundance in one task and data crunch in another related task.
2. Enough data available for training, but lack of computational resources

**Tensor Flow and Keras**

TensorFlow is an end-to-end open-source platform for machine learning. It’s a comprehensive and flexible ecosystem of tools, libraries and other resources that provide workflows with high-level APIs. The framework offers various levels of concepts for you to choose the one you need to build and deploy machine learning models.

For instance, if you need to do some large machine learning tasks, you can use the Distribution Strategy API in order to perform distributed hardware configurations and if you need a full production machine learning pipeline, you can simply use TensorFlow Extended (TFX). Some of the salient features are described below:

1. Easy Model Building: TensorFlow offers multiple levels of abstraction to build and train models.
2. Robust ML Production Anywhere: TensorFlow lets you train and deploy your model easily, no matter what language or platform you use.
3. Powerful Experimentation for Research: TensorFlow gives you flexibility and control with features like the Keras Functional API and Model sub classing API for the creation of complex topologies.

Keras, on the other hand, is a high-level neural networks library that is running on the top of TensorFlow, CNTK, and Theano. Using Keras in deep learning allows for easy and fast prototyping as well as running seamlessly on CPU and GPU. This framework is written in Python code which is easy to debug and allows ease for extensibility. The main advantages of Keras are described below:

1. User-Friendly: Keras has a simple, consistent interface optimized for common use cases which provides clear and actionable feedback for user errors.
2. Modular and Composable: Keras models are made by connecting configurable building blocks together, with few restrictions.
3. Easy to Extend: With the help of Keras, you can easily write custom building blocks for new ideas and researches.
4. Easy to Use: Keras offers consistent & simple APIs which helps in minimizing the number of user actions required for common use cases, also it provides clear and actionable feedback upon user error.

Batch Normalization

We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers, that are changing all the time, and get 10 times or more improvement in the training speed.

Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift). To explain covariance shift, let’s have a deep network on cat detection. We train our data on only black cats’ images. So, if we now try to apply this network to data with colored cats, it is obvious; we’re not going to do well. The training set and the prediction set are both cats’ images but they differ a little bit. In other words, if an algorithm learned some X to Y mapping, and if the distribution of X changes, then we might need to retrain the learning algorithm by trying to align the distribution of X with the distribution of Y

Also, batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers.

We can use higher learning rates because batch normalization makes sure that there’s no activation that’s gone really high or really low. And by that, things that previously couldn’t get to train, it will start to train.

It reduces overfitting because it has a slight regularization effects. Similar to dropout, it adds some noise to each hidden layer’s activations. Therefore, if we use batch normalization, we will use less dropout, which is a good thing because we are not going to lose a lot of information. However, we should not depend only on batch normalization for regularization; we should better use it together with dropout.

Working of BN

* We can use higher learning rates because batch normalization makes sure that there’s no activation that’s gone really high or really low. And by that, things that previously couldn’t get to train, it will start to train.
* It reduces overfitting because it has a slight regularization effects. Similar to dropout, it adds some noise to each hidden layer’s activations. Therefore, if we use batch normalization, we will use less dropout, which is a good thing because we are not going to lose a lot of information. However, we should not depend only on batch normalization for regularization; we should better use it together with dropout.

VGG doesn’t have a batch norm layer in it because batch normalization didn’t exist before VGG. If we train it with it from the start, the pre-trained weight will benefit from the normalization of the activations. So adding a batch norm layer actually improves ImageNet, which is cool. You can add it to dense layers, and also to convolutional layers.

If we insert a batch norm in a pre-trained network, it will change the pre-trained weights, because it will subtract the mean and divide by the standard deviation for the activation layers and we don’t want that to happen because we need those pre-trained weights to stay the same. So, what we need to do is to insert a batch norm layer and figure out gamma and beta in order to undo the outputs change.

Dropout

The term “dropout” refers to dropping out units (both hidden and visible) in a neural network.

Simply put, dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By “ignoring”, I mean these units are not considered during a particular forward or backward pass.

More technically, At each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

Given that we know a bit about dropout, a question arises — why do we need dropout at all? Why do we need to literally shut-down parts of a neural networks?

The answer to these questions is “to prevent over-fitting”.

A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data.

In machine learning, regularization is way to prevent over-fitting. Regularization reduces over-fitting by adding a penalty to the loss function. By adding this penalty, the model is trained such that it does not learn interdependent set of features weights. Those of you who know Logistic Regression might be familiar with L1 (Laplacian) and L2 (Gaussian) penalties.

Dropout is an approach to regularization in neural networks which helps reducing interdependent learning amongst the neurons.

Training Phase:

Training Phase: For each hidden layer, for each training sample, for each iteration, ignore (zero out) a random fraction, p, of nodes (and corresponding activations).

Testing Phase:

Use all activations, but reduce them by a factor p (to account for the missing activations during training).



Figure 1 Srivastava, Nitish, et al. ”Dropout: a simple way to prevent neural networks from

Some Observations:

Dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.

Dropout roughly doubles the number of iterations required to converge. However, training time for each epoch is less.

With H hidden units, each of which can be dropped, we have  
2^H possible models. In testing phase, the entire network is considered, and each activation is reduced by a factor p.

**Design and Experiment**

**Design**

For the fashion dataset, we are going to load the data using in-built datasets in keras

from keras.datasets import fashion\_mnist

Then we split the training and test datasets and check their dimensions

# Splitting the data between train and test

(x\_train, y\_train), (x\_test, y\_test) = fashion\_mnist.load\_data()

print('x\_train shape:', x\_train.shape)

print(x\_train.shape[0], 'train samples')

print(x\_test.shape[0], 'test samples')

x\_train shape: (60000, 28, 28)

60000 train samples

10000 test samples

Check the sample data

# plotting some random 10 images

class\_names = ['T\_shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

fig = plt.figure(figsize=(8,3))

for i in range(num\_classes):

    ax = fig.add\_subplot(2, 5, 1 + i, xticks=[], yticks=[])

    idx = np.where(y\_train[:]==i)[0]

    features\_idx = x\_train[idx,::]

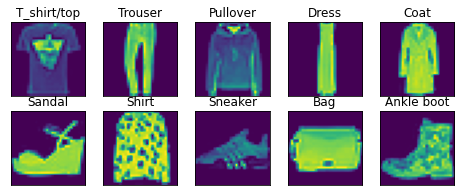
    img\_num = np.random.randint(features\_idx.shape[0])

    im = (features\_idx[img\_num,::])

    ax.set\_title(class\_names[i])

    plt.imshow(im)

plt.show()



Class distribution

def plot\_label\_per\_class(data):

    f, ax = plt.subplots(1,1, figsize=(12,4))

    g = sns.countplot(data.label, order = data["label"].value\_counts().index)

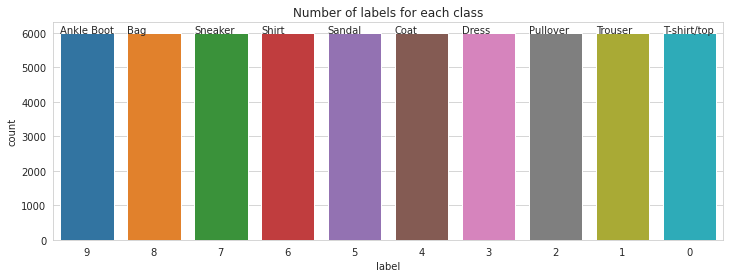
    g.set\_title("Number of labels for each class")

    for p, label in zip(g.patches, data["label"].value\_counts().index):

        g.annotate(labels[label], (p.get\_x(), p.get\_height()+0.1))

    plt.show()

plot\_label\_per\_class(fashion\_train\_df)



Ankle Boot : 10.0% - 6000 entries

Bag : 10.0% - 6000 entries

Sneaker : 10.0% - 6000 entries

Shirt : 10.0% - 6000 entries

Sandal : 10.0% - 6000 entries

Coat : 10.0% - 6000 entries

Dress : 10.0% - 6000 entries

Pullover : 10.0% - 6000 entries

Trouser : 10.0% - 6000 entries

T-shirt/top : 10.0% - 6000 entries

We will use a Sequential model.

The Sequential model is a linear stack of layers. It can be first initialized and then we add layers using add method or we can add all layers at initial stage. The layers added are as follows:

Conv2D is a 2D Convolutional layer (i.e. spatial convolution over images). The parameters used are:

* **Filters** - the number of filters (Kernels) used with this layer;  
  ***kernel\_size*** - the dimension of the Kernel;   
  ***activation*** - is the activation function used;  
  ***kernel\_initializer*** - the function used for initializing the kernel;  
  ***input\_shape*** - is the shape of the image presented to the CNN. The input and output of the Conv2D is a 4D tensor.
* **MaxPooling2D** is a Max pooling operation for spatial data. Parameters used here are:

pool\_size - representing the factors by which to downscale in both directions.

* **Flatten** layer Flattens the input. Does not affect the batch size. It is used without parameters.
* **Dense** layer is a regular fully connected NN layer. It is used without parameters.
* **units** - this is a positive integer, with the meaning: dimensionality of the output space activation - activation
* **Dense**. This is the final layer (fully connected). It is used with the parameters:

***units***: the number of classes (in our case 10);  
***activation***: softmax; for this final layer it is used softmax activation (standard for multiclass classification)

* Then we compile the model, specifying the following parameters as well: ***loss; optimizer; metrics***.

Then, we are going to experiment different training models.

**Experiment - I:** Dropouts After Conv and FC layers

**Experiment - II:** Remove the dropouts after the convolutional layers (but retain them in the FC layer). Also, use **batch normalization** after every convolutional layer.

**Experiment - III:** Use batch normalization and dropouts after every convolutional layer. Also, retain the dropouts in the FC layer.

**Experiment - IV:** Remove the dropouts after the convolutional layers and use L2 regularization in the FC layer. Retain the dropouts in FC.

**Experiment-V:** Dropouts after conv layer, L2 in FC, use BN after convolutional layer

**Experiment-VI:** Add a **new convolutional layer** to the network. Note that by a 'convolutional layer', we are referring to a convolutional unit with two sets of Conv2D layers with 128 filters each.

**Experiment - VII:** Add more feature maps to the conv layers: from 32 to 64 and 64 to 128.

**Experiment – I**

model = Sequential()

# first CONV => RELU => CONV => RELU => POOL layer => Dropout

model.add(Conv2D(32, (3, 3), input\_shape = input\_shape, kernel\_initializer='he\_normal'))

model.add(Activation("relu"))

model.add(Conv2D(32, (3, 3), padding="same"))

model.add(Activation("relu"))

model.add(MaxPooling2D(pool\_size = (2, 2)))

model.add(Dropout(0.25))

# second CONV => RELU => CONV => RELU => POOL layer => Dropout

model.add(Conv2D(64, (3, 3), padding="same"))

model.add(Activation("relu"))

model.add(Conv2D(64, (3, 3), padding="same"))

model.add(Activation("relu"))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(units = 512, activation = 'relu'))

model.add(Dropout(0.5))

model.add(Dense(units = 10, activation = 'softmax'))

# summary

model.summary()

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_2 (Conv2D) (None, 26, 26, 32) 320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_1 (Activation) (None, 26, 26, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 26, 26, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_2 (Activation) (None, 26, 26, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_4 (Conv2D) (None, 13, 13, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_3 (Activation) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_5 (Conv2D) (None, 13, 13, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_4 (Activation) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 6, 6, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 6, 6, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 2304) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 512) 1180160

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_2 (Dropout) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 5130

=================================================================

Total params: 1,250,282

Trainable params: 1,250,282

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# compile the model

model.compile(loss='categorical\_crossentropy',

              optimizer='sgd',

              metrics=['accuracy'])

# train

history = model.fit(np.expand\_dims(x\_train, -1), y\_train,

              batch\_size=batch\_size,

              epochs=epochs,

              validation\_split=0.3,

              shuffle=True)

Epoch 1/25

329/329 [==============================] - 155s 473ms/step - loss: 0.7346 - accuracy: 0.7274 - val\_loss: 0.5854 - val\_accuracy: 0.7819

Epoch 2/25

329/329 [==============================] - 155s 471ms/step - loss: 0.6454 - accuracy: 0.7598 - val\_loss: 0.5855 - val\_accuracy: 0.7772

Epoch 3/25

329/329 [==============================] - 155s 471ms/step - loss: 0.6030 - accuracy: 0.7763 - val\_loss: 0.5199 - val\_accuracy: 0.8014

Epoch 4/25

329/329 [==============================] - 156s 473ms/step - loss: 0.5649 - accuracy: 0.7905 - val\_loss: 0.4820 - val\_accuracy: 0.8218

Epoch 5/25

329/329 [==============================] - 156s 474ms/step - loss: 0.5429 - accuracy: 0.8004 - val\_loss: 0.4636 - val\_accuracy: 0.8331

Epoch 6/25

329/329 [==============================] - 157s 477ms/step - loss: 0.5204 - accuracy: 0.8092 - val\_loss: 0.4576 - val\_accuracy: 0.8304

Epoch 7/25

329/329 [==============================] - 157s 477ms/step - loss: 0.5055 - accuracy: 0.8147 - val\_loss: 0.4341 - val\_accuracy: 0.8440

Epoch 8/25

329/329 [==============================] - 157s 476ms/step - loss: 0.4912 - accuracy: 0.8204 - val\_loss: 0.4205 - val\_accuracy: 0.8496

Epoch 9/25

329/329 [==============================] - 157s 476ms/step - loss: 0.4771 - accuracy: 0.8253 - val\_loss: 0.4142 - val\_accuracy: 0.8530

Epoch 10/25

329/329 [==============================] - 157s 478ms/step - loss: 0.4627 - accuracy: 0.8316 - val\_loss: 0.4366 - val\_accuracy: 0.8323

Epoch 11/25

329/329 [==============================] - 158s 481ms/step - loss: 0.4548 - accuracy: 0.8344 - val\_loss: 0.3894 - val\_accuracy: 0.8596

Epoch 12/25

329/329 [==============================] - 158s 481ms/step - loss: 0.4455 - accuracy: 0.8382 - val\_loss: 0.3856 - val\_accuracy: 0.8608

Epoch 13/25

329/329 [==============================] - 158s 481ms/step - loss: 0.4341 - accuracy: 0.8427 - val\_loss: 0.3994 - val\_accuracy: 0.8499

Epoch 14/25

329/329 [==============================] - 159s 482ms/step - loss: 0.4289 - accuracy: 0.8452 - val\_loss: 0.3719 - val\_accuracy: 0.8640

Epoch 15/25

329/329 [==============================] - 159s 483ms/step - loss: 0.4214 - accuracy: 0.8466 - val\_loss: 0.3687 - val\_accuracy: 0.8674

Epoch 16/25

329/329 [==============================] - 159s 484ms/step - loss: 0.4124 - accuracy: 0.8523 - val\_loss: 0.3699 - val\_accuracy: 0.8637

Epoch 17/25

329/329 [==============================] - 159s 484ms/step - loss: 0.4047 - accuracy: 0.8522 - val\_loss: 0.3705 - val\_accuracy: 0.8617

Epoch 18/25

329/329 [==============================] - 160s 486ms/step - loss: 0.4001 - accuracy: 0.8553 - val\_loss: 0.3499 - val\_accuracy: 0.8727

Epoch 19/25

329/329 [==============================] - 161s 489ms/step - loss: 0.3943 - accuracy: 0.8580 - val\_loss: 0.3530 - val\_accuracy: 0.8707

Epoch 20/25

329/329 [==============================] - 161s 489ms/step - loss: 0.3867 - accuracy: 0.8604 - val\_loss: 0.3362 - val\_accuracy: 0.8766

Epoch 21/25

329/329 [==============================] - 162s 491ms/step - loss: 0.3843 - accuracy: 0.8613 - val\_loss: 0.3521 - val\_accuracy: 0.8710

Epoch 22/25

329/329 [==============================] - 161s 490ms/step - loss: 0.3780 - accuracy: 0.8640 - val\_loss: 0.3624 - val\_accuracy: 0.8651

Epoch 23/25

329/329 [==============================] - 161s 490ms/step - loss: 0.3717 - accuracy: 0.8634 - val\_loss: 0.3260 - val\_accuracy: 0.8811

Epoch 24/25

329/329 [==============================] - 161s 490ms/step - loss: 0.3667 - accuracy: 0.8656 - val\_loss: 0.3289 - val\_accuracy: 0.8798

Epoch 25/25

329/329 [==============================] - 161s 490ms/step - loss: 0.3654 - accuracy: 0.8664 - val\_loss: 0.3306 - val\_accuracy: 0.8782

plt.figure(figsize=(12, 8))

plt.subplot(2, 2, 1)

plt.plot(history.history['loss'], label='Loss')

plt.plot(history.history['val\_loss'], label='val\_Loss')

plt.legend()

plt.title('Loss evolution')

plt.subplot(2, 2, 2)

plt.plot(history.history['accuracy'], label='accuracy')

plt.plot(history.history['val\_accuracy'], label='val\_accuracy')

plt.legend()

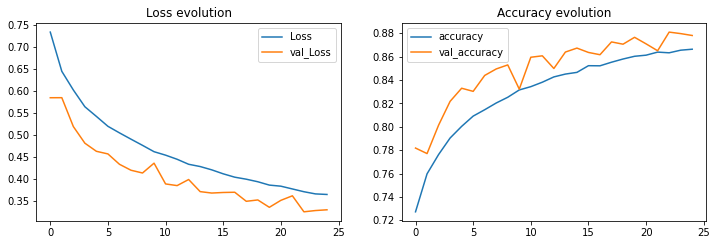
plt.title('Accuracy evolution')

evaluation = model.evaluate(np.expand\_dims(x\_test, -1), y\_test)

print(f'Test Accuracy : {evaluation[1]:.3f}')

313/313 [==============================] - 9s 29ms/step - loss: 0.3480 - accuracy: 0.8703

Test Accuracy : 0.870



**Experiment – II**

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input\_shape=input\_shape))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(32, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(64, (3, 3), padding='same'))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(512))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_classes))

model.add(Activation('softmax'))

# summary of the model

print(model.summary())

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_1 (Conv2D) (None, 28, 28, 32) 320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_1 (Activation) (None, 28, 28, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization (BatchNo (None, 28, 28, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 26, 26, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_2 (Activation) (None, 26, 26, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_1 (Batch (None, 26, 26, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 13, 13, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_3 (Activation) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_2 (Batch (None, 13, 13, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_4 (Conv2D) (None, 11, 11, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_4 (Activation) (None, 11, 11, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_3 (Batch (None, 11, 11, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 5, 5, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 1600) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 512) 819712

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_5 (Activation) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 5130

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_6 (Activation) (None, 10) 0

=================================================================

Total params: 890,602

Trainable params: 890,218

Non-trainable params: 384

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# compile

model.compile(loss='categorical\_crossentropy',

              optimizer='sgd',

              metrics=['accuracy'])

history = model.fit(np.expand\_dims(x\_train, -1), y\_train,

              batch\_size=batch\_size,

              epochs=epochs,

              validation\_split=0.3,

              shuffle=True)

Epoch 1/25

329/329 [==============================] - 162s 493ms/step - loss: 0.6676 - accuracy: 0.7697 - val\_loss: 1.4349 - val\_accuracy: 0.5177

Epoch 2/25

329/329 [==============================] - 161s 491ms/step - loss: 0.4271 - accuracy: 0.8452 - val\_loss: 0.4143 - val\_accuracy: 0.8526

Epoch 3/25

329/329 [==============================] - 162s 493ms/step - loss: 0.3672 - accuracy: 0.8684 - val\_loss: 0.4326 - val\_accuracy: 0.8296

Epoch 4/25

329/329 [==============================] - 162s 492ms/step - loss: 0.3350 - accuracy: 0.8775 - val\_loss: 0.3315 - val\_accuracy: 0.8753

Epoch 5/25

329/329 [==============================] - 162s 493ms/step - loss: 0.3081 - accuracy: 0.8890 - val\_loss: 0.3144 - val\_accuracy: 0.8819

Epoch 6/25

329/329 [==============================] - 162s 493ms/step - loss: 0.2889 - accuracy: 0.8955 - val\_loss: 0.2798 - val\_accuracy: 0.8932

Epoch 7/25

329/329 [==============================] - 162s 493ms/step - loss: 0.2741 - accuracy: 0.8993 - val\_loss: 0.2985 - val\_accuracy: 0.8891

Epoch 8/25

329/329 [==============================] - 162s 493ms/step - loss: 0.2604 - accuracy: 0.9040 - val\_loss: 0.3128 - val\_accuracy: 0.8877

Epoch 9/25

329/329 [==============================] - 162s 494ms/step - loss: 0.2509 - accuracy: 0.9089 - val\_loss: 0.2752 - val\_accuracy: 0.9014

Epoch 10/25

329/329 [==============================] - 162s 493ms/step - loss: 0.2374 - accuracy: 0.9118 - val\_loss: 0.3011 - val\_accuracy: 0.8868

Epoch 11/25

329/329 [==============================] - 163s 495ms/step - loss: 0.2322 - accuracy: 0.9161 - val\_loss: 0.2553 - val\_accuracy: 0.9062

Epoch 12/25

329/329 [==============================] - 162s 492ms/step - loss: 0.2244 - accuracy: 0.9182 - val\_loss: 0.3045 - val\_accuracy: 0.8888

Epoch 13/25

329/329 [==============================] - 162s 491ms/step - loss: 0.2134 - accuracy: 0.9220 - val\_loss: 0.2575 - val\_accuracy: 0.9072

Epoch 14/25

329/329 [==============================] - 162s 492ms/step - loss: 0.2048 - accuracy: 0.9262 - val\_loss: 0.2647 - val\_accuracy: 0.9033

Epoch 15/25

329/329 [==============================] - 162s 492ms/step - loss: 0.1998 - accuracy: 0.9272 - val\_loss: 0.3022 - val\_accuracy: 0.8899

Epoch 16/25

329/329 [==============================] - 161s 490ms/step - loss: 0.1940 - accuracy: 0.9290 - val\_loss: 0.2596 - val\_accuracy: 0.9082

Epoch 17/25

329/329 [==============================] - 162s 492ms/step - loss: 0.1914 - accuracy: 0.9296 - val\_loss: 0.2532 - val\_accuracy: 0.9084

Epoch 18/25

329/329 [==============================] - 161s 491ms/step - loss: 0.1823 - accuracy: 0.9338 - val\_loss: 0.3335 - val\_accuracy: 0.8788

Epoch 19/25

329/329 [==============================] - 163s 494ms/step - loss: 0.1753 - accuracy: 0.9349 - val\_loss: 0.4178 - val\_accuracy: 0.8535

Epoch 20/25

329/329 [==============================] - 161s 489ms/step - loss: 0.1725 - accuracy: 0.9364 - val\_loss: 0.2374 - val\_accuracy: 0.9139

Epoch 21/25

329/329 [==============================] - 163s 496ms/step - loss: 0.1645 - accuracy: 0.9394 - val\_loss: 0.2459 - val\_accuracy: 0.9126

Epoch 22/25

329/329 [==============================] - 161s 489ms/step - loss: 0.1608 - accuracy: 0.9415 - val\_loss: 0.2737 - val\_accuracy: 0.9022

Epoch 23/25

329/329 [==============================] - 162s 492ms/step - loss: 0.1541 - accuracy: 0.9436 - val\_loss: 0.3845 - val\_accuracy: 0.8666

Epoch 24/25

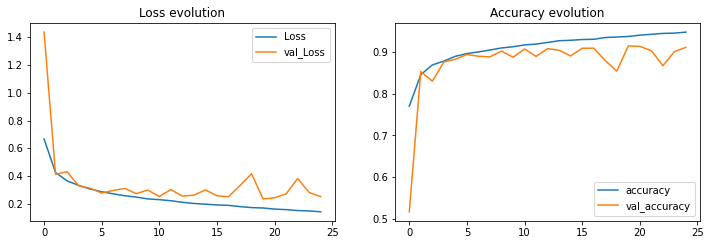
329/329 [==============================] - 162s 492ms/step - loss: 0.1513 - accuracy: 0.9443 - val\_loss: 0.2833 - val\_accuracy: 0.9001

Epoch 25/25

329/329 [==============================] - 163s 495ms/step - loss: 0.1450 - accuracy: 0.9467 - val\_loss: 0.2542 - val\_accuracy: 0.9104

313/313 [==============================] - 9s 30ms/step - loss: 0.2709 - accuracy: 0.9047

Test Accuracy : 0.905

****

**Experiment – III**

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input\_shape=input\_shape))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(32, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(512))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_classes))

model.add(Activation('softmax'))

# summary of the model

print(model.summary())

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_1 (Conv2D) (None, 28, 28, 32) 320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation (Activation) (None, 28, 28, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization (BatchNo (None, 28, 28, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 26, 26, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_1 (Activation) (None, 26, 26, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_1 (Batch (None, 26, 26, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 13, 13, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_2 (Activation) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_2 (Batch (None, 13, 13, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_4 (Conv2D) (None, 11, 11, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_3 (Activation) (None, 11, 11, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_3 (Batch (None, 11, 11, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 5, 5, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 5, 5, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 1600) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 512) 819712

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_4 (Activation) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_2 (Dropout) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 5130

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_5 (Activation) (None, 10) 0

=================================================================

Total params: 890,602

Trainable params: 890,218

Non-trainable params: 384

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# compile

model.compile(loss='categorical\_crossentropy',

              optimizer='sgd',

              metrics=['accuracy'])

x\_train = x\_train.astype('float32')

x\_test = x\_test.astype('float32')

# Training the model

history = model.fit(np.expand\_dims(x\_train, -1), y\_train,

              batch\_size=batch\_size,

              epochs=epochs,

              validation\_split=0.3,

              shuffle=True)

Epoch 1/25

329/329 [==============================] - 177s 538ms/step - loss: 0.8598 - accuracy: 0.7121 - val\_loss: 1.3963 - val\_accuracy: 0.6241

Epoch 2/25

329/329 [==============================] - 170s 516ms/step - loss: 0.5467 - accuracy: 0.7984 - val\_loss: 0.4396 - val\_accuracy: 0.8388

Epoch 3/25

329/329 [==============================] - 171s 519ms/step - loss: 0.4808 - accuracy: 0.8245 - val\_loss: 0.3891 - val\_accuracy: 0.8573

Epoch 4/25

329/329 [==============================] - 168s 510ms/step - loss: 0.4413 - accuracy: 0.8391 - val\_loss: 0.3543 - val\_accuracy: 0.8636

Epoch 5/25

329/329 [==============================] - 171s 521ms/step - loss: 0.4076 - accuracy: 0.8490 - val\_loss: 0.3412 - val\_accuracy: 0.8724

Epoch 6/25

329/329 [==============================] - 173s 526ms/step - loss: 0.3898 - accuracy: 0.8590 - val\_loss: 0.3507 - val\_accuracy: 0.8709

Epoch 7/25

329/329 [==============================] - 169s 515ms/step - loss: 0.3729 - accuracy: 0.8637 - val\_loss: 0.3426 - val\_accuracy: 0.8704

Epoch 8/25

329/329 [==============================] - 168s 510ms/step - loss: 0.3610 - accuracy: 0.8667 - val\_loss: 0.3011 - val\_accuracy: 0.8869

Epoch 9/25

329/329 [==============================] - 169s 512ms/step - loss: 0.3491 - accuracy: 0.8724 - val\_loss: 0.3135 - val\_accuracy: 0.8816

Epoch 10/25

329/329 [==============================] - 169s 513ms/step - loss: 0.3369 - accuracy: 0.8768 - val\_loss: 0.2986 - val\_accuracy: 0.8848

Epoch 11/25

329/329 [==============================] - 172s 524ms/step - loss: 0.3270 - accuracy: 0.8797 - val\_loss: 0.2796 - val\_accuracy: 0.8937

Epoch 12/25

329/329 [==============================] - 172s 522ms/step - loss: 0.3184 - accuracy: 0.8826 - val\_loss: 0.2814 - val\_accuracy: 0.8951

Epoch 13/25

329/329 [==============================] - 170s 517ms/step - loss: 0.3144 - accuracy: 0.8835 - val\_loss: 0.3344 - val\_accuracy: 0.8744

Epoch 14/25

329/329 [==============================] - 177s 537ms/step - loss: 0.3096 - accuracy: 0.8845 - val\_loss: 0.2722 - val\_accuracy: 0.8995

Epoch 15/25

329/329 [==============================] - 168s 510ms/step - loss: 0.3021 - accuracy: 0.8903 - val\_loss: 0.2998 - val\_accuracy: 0.8868

Epoch 16/25

329/329 [==============================] - 168s 512ms/step - loss: 0.2981 - accuracy: 0.8895 - val\_loss: 0.2928 - val\_accuracy: 0.8876

Epoch 17/25

329/329 [==============================] - 171s 520ms/step - loss: 0.2929 - accuracy: 0.8921 - val\_loss: 0.2672 - val\_accuracy: 0.8991

Epoch 18/25

329/329 [==============================] - 170s 517ms/step - loss: 0.2867 - accuracy: 0.8942 - val\_loss: 0.2660 - val\_accuracy: 0.9008

Epoch 19/25

329/329 [==============================] - 168s 509ms/step - loss: 0.2796 - accuracy: 0.8965 - val\_loss: 0.2649 - val\_accuracy: 0.9007

Epoch 20/25

329/329 [==============================] - 173s 527ms/step - loss: 0.2787 - accuracy: 0.8968 - val\_loss: 0.3004 - val\_accuracy: 0.8858

Epoch 21/25

329/329 [==============================] - 170s 517ms/step - loss: 0.2767 - accuracy: 0.8977 - val\_loss: 0.4569 - val\_accuracy: 0.8416

Epoch 22/25

329/329 [==============================] - 168s 509ms/step - loss: 0.2704 - accuracy: 0.8995 - val\_loss: 0.2560 - val\_accuracy: 0.9051

Epoch 23/25

329/329 [==============================] - 176s 534ms/step - loss: 0.2657 - accuracy: 0.9028 - val\_loss: 0.2749 - val\_accuracy: 0.8967

Epoch 24/25

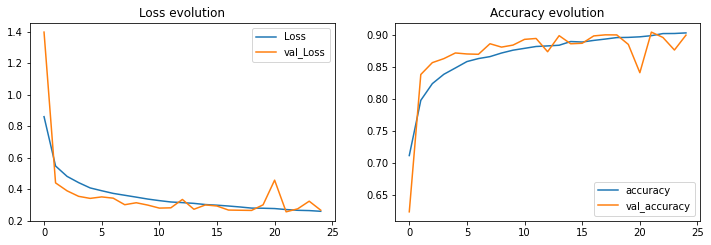
329/329 [==============================] - 169s 514ms/step - loss: 0.2638 - accuracy: 0.9029 - val\_loss: 0.3234 - val\_accuracy: 0.8771

Epoch 25/25

329/329 [==============================] - 171s 519ms/step - loss: 0.2593 - accuracy: 0.9038 - val\_loss: 0.2663 - val\_accuracy: 0.9003

313/313 [==============================] - 10s 31ms/step - loss: 0.2838 - accuracy: 0.8929

Test Accuracy : 0.893

****

**Experiment – IV**

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input\_shape=input\_shape))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(32, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(64, (3, 3), padding='same'))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(512,kernel\_regularizer=l2(0.01)))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_classes))

model.add(Activation('softmax'))

# summary of the model

print(model.summary())

Model: "sequential\_2"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_4 (Conv2D) (None, 28, 28, 32) 320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_4 (Activation) (None, 28, 28, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_4 (Batch (None, 28, 28, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_5 (Conv2D) (None, 26, 26, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_5 (Activation) (None, 26, 26, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_5 (Batch (None, 26, 26, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2 (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_6 (Conv2D) (None, 13, 13, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_6 (Activation) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_6 (Batch (None, 13, 13, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_7 (Conv2D) (None, 11, 11, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_7 (Activation) (None, 11, 11, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_7 (Batch (None, 11, 11, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_3 (MaxPooling2 (None, 5, 5, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 1600) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 512) 819712

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_8 (Activation) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 5130

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_9 (Activation) (None, 10) 0

=================================================================

Total params: 890,602

Trainable params: 890,218

Non-trainable params: 384

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# compile

model.compile(loss='categorical\_crossentropy',

              optimizer='sgd',

              metrics=['accuracy'])

# Training the model

history = model.fit(np.expand\_dims(x\_train, -1), y\_train,

              batch\_size=batch\_size,

              epochs=epochs,

              validation\_split=0.3,

              shuffle=True)

Epoch 1/25

329/329 [==============================] - 157s 476ms/step - loss: 7.9554 - accuracy: 0.7692 - val\_loss: 8.0477 - val\_accuracy: 0.5868

Epoch 2/25

329/329 [==============================] - 157s 476ms/step - loss: 6.7992 - accuracy: 0.8510 - val\_loss: 6.3606 - val\_accuracy: 0.8578

Epoch 3/25

329/329 [==============================] - 157s 476ms/step - loss: 5.9594 - accuracy: 0.8725 - val\_loss: 5.5806 - val\_accuracy: 0.8770

Epoch 4/25

329/329 [==============================] - 156s 475ms/step - loss: 5.2403 - accuracy: 0.8824 - val\_loss: 4.9168 - val\_accuracy: 0.8822

Epoch 5/25

329/329 [==============================] - 157s 477ms/step - loss: 4.6114 - accuracy: 0.8924 - val\_loss: 4.3264 - val\_accuracy: 0.8944

Epoch 6/25

329/329 [==============================] - 157s 476ms/step - loss: 4.0643 - accuracy: 0.8979 - val\_loss: 3.8254 - val\_accuracy: 0.8967

Epoch 7/25

329/329 [==============================] - 157s 478ms/step - loss: 3.5848 - accuracy: 0.9037 - val\_loss: 3.3722 - val\_accuracy: 0.9026

Epoch 8/25

329/329 [==============================] - 157s 477ms/step - loss: 3.1681 - accuracy: 0.9086 - val\_loss: 3.0205 - val\_accuracy: 0.8902

Epoch 9/25

329/329 [==============================] - 157s 477ms/step - loss: 2.8013 - accuracy: 0.9123 - val\_loss: 2.6825 - val\_accuracy: 0.8916

Epoch 10/25

329/329 [==============================] - 157s 477ms/step - loss: 2.4767 - accuracy: 0.9167 - val\_loss: 2.3625 - val\_accuracy: 0.9064

Epoch 11/25

329/329 [==============================] - 157s 477ms/step - loss: 2.1946 - accuracy: 0.9209 - val\_loss: 2.1547 - val\_accuracy: 0.8834

Epoch 12/25

329/329 [==============================] - 158s 480ms/step - loss: 1.9504 - accuracy: 0.9213 - val\_loss: 1.9819 - val\_accuracy: 0.8648

Epoch 13/25

329/329 [==============================] - 157s 476ms/step - loss: 1.7301 - accuracy: 0.9260 - val\_loss: 1.6690 - val\_accuracy: 0.9122

Epoch 14/25

329/329 [==============================] - 157s 476ms/step - loss: 1.5387 - accuracy: 0.9300 - val\_loss: 1.7101 - val\_accuracy: 0.8422

Epoch 15/25

329/329 [==============================] - 157s 477ms/step - loss: 1.3721 - accuracy: 0.9324 - val\_loss: 1.3471 - val\_accuracy: 0.9132

Epoch 16/25

329/329 [==============================] - 157s 476ms/step - loss: 1.2211 - accuracy: 0.9357 - val\_loss: 1.2072 - val\_accuracy: 0.9148

Epoch 17/25

329/329 [==============================] - 157s 477ms/step - loss: 1.0930 - accuracy: 0.9371 - val\_loss: 1.0869 - val\_accuracy: 0.9180

Epoch 18/25

329/329 [==============================] - 157s 477ms/step - loss: 0.9761 - accuracy: 0.9415 - val\_loss: 1.0160 - val\_accuracy: 0.9059

Epoch 19/25

329/329 [==============================] - 156s 474ms/step - loss: 0.8765 - accuracy: 0.9415 - val\_loss: 0.8976 - val\_accuracy: 0.9198

Epoch 20/25

329/329 [==============================] - 156s 474ms/step - loss: 0.7874 - accuracy: 0.9449 - val\_loss: 0.8343 - val\_accuracy: 0.9132

Epoch 21/25

329/329 [==============================] - 156s 475ms/step - loss: 0.7118 - accuracy: 0.9465 - val\_loss: 0.7451 - val\_accuracy: 0.9209

Epoch 22/25

329/329 [==============================] - 157s 477ms/step - loss: 0.6385 - accuracy: 0.9511 - val\_loss: 0.7858 - val\_accuracy: 0.8873

Epoch 23/25

329/329 [==============================] - 156s 475ms/step - loss: 0.5771 - accuracy: 0.9537 - val\_loss: 0.7404 - val\_accuracy: 0.8824

Epoch 24/25

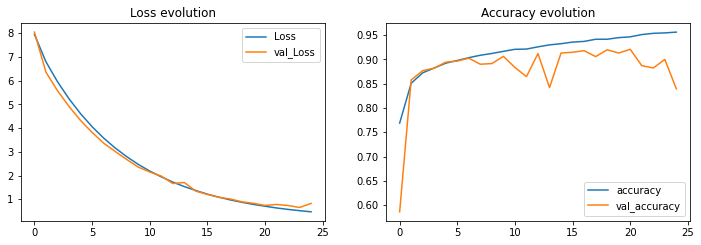
329/329 [==============================] - 156s 476ms/step - loss: 0.5236 - accuracy: 0.9546 - val\_loss: 0.6577 - val\_accuracy: 0.9001

Epoch 25/25

329/329 [==============================] - 157s 476ms/step - loss: 0.4758 - accuracy: 0.9562 - val\_loss: 0.8301 - val\_accuracy: 0.8397

313/313 [==============================] - 9s 29ms/step - loss: 0.8643 - accuracy: 0.8310

Test Accuracy : 0.831

****

**Experiment – V**

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input\_shape=input\_shape))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(32, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(512,kernel\_regularizer=l2(0.01)))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_classes))

model.add(Activation('softmax'))

# summary of the model

print(model.summary())

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_1 (Conv2D) (None, 28, 28, 32) 320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation (Activation) (None, 28, 28, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization (BatchNo (None, 28, 28, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 26, 26, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_1 (Activation) (None, 26, 26, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_1 (Batch (None, 26, 26, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 13, 13, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_2 (Activation) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_2 (Batch (None, 13, 13, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_4 (Conv2D) (None, 11, 11, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_3 (Activation) (None, 11, 11, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_3 (Batch (None, 11, 11, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 5, 5, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 5, 5, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 1600) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 512) 819712

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_4 (Activation) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_2 (Dropout) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 5130

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_5 (Activation) (None, 10) 0

=================================================================

Total params: 890,602

Trainable params: 890,218

Non-trainable params: 384

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# compile

model.compile(loss='categorical\_crossentropy',

              optimizer='sgd',

              metrics=['accuracy'])

# train

history = model.fit(np.expand\_dims(x\_train, -1), y\_train,

              batch\_size=batch\_size,

              epochs=epochs,

              validation\_split=0.3,

              shuffle=True)

Epoch 1/25

329/329 [==============================] - 166s 504ms/step - loss: 8.1147 - accuracy: 0.7170 - val\_loss: 8.2116 - val\_accuracy: 0.5819

Epoch 2/25

329/329 [==============================] - 166s 504ms/step - loss: 6.8914 - accuracy: 0.8068 - val\_loss: 6.4042 - val\_accuracy: 0.8332

Epoch 3/25

329/329 [==============================] - 165s 503ms/step - loss: 6.0458 - accuracy: 0.8288 - val\_loss: 5.5844 - val\_accuracy: 0.8637

Epoch 4/25

329/329 [==============================] - 166s 505ms/step - loss: 5.3163 - accuracy: 0.8447 - val\_loss: 4.9298 - val\_accuracy: 0.8683

Epoch 5/25

329/329 [==============================] - 166s 504ms/step - loss: 4.6863 - accuracy: 0.8557 - val\_loss: 4.4017 - val\_accuracy: 0.8542

Epoch 6/25

329/329 [==============================] - 166s 506ms/step - loss: 4.1381 - accuracy: 0.8634 - val\_loss: 3.8321 - val\_accuracy: 0.8858

Epoch 7/25

329/329 [==============================] - 167s 509ms/step - loss: 3.6580 - accuracy: 0.8705 - val\_loss: 3.3921 - val\_accuracy: 0.8886

Epoch 8/25

329/329 [==============================] - 166s 504ms/step - loss: 3.2383 - accuracy: 0.8752 - val\_loss: 3.0023 - val\_accuracy: 0.8934

Epoch 9/25

329/329 [==============================] - 167s 506ms/step - loss: 2.8702 - accuracy: 0.8818 - val\_loss: 2.6883 - val\_accuracy: 0.8842

Epoch 10/25

329/329 [==============================] - 166s 504ms/step - loss: 2.5517 - accuracy: 0.8837 - val\_loss: 2.3857 - val\_accuracy: 0.8895

Epoch 11/25

329/329 [==============================] - 165s 501ms/step - loss: 2.2635 - accuracy: 0.8880 - val\_loss: 2.1104 - val\_accuracy: 0.8981

Epoch 12/25

329/329 [==============================] - 165s 502ms/step - loss: 2.0179 - accuracy: 0.8932 - val\_loss: 1.8718 - val\_accuracy: 0.9031

Epoch 13/25

329/329 [==============================] - 171s 521ms/step - loss: 1.8011 - accuracy: 0.8952 - val\_loss: 1.6961 - val\_accuracy: 0.8959

Epoch 14/25

329/329 [==============================] - 166s 506ms/step - loss: 1.6106 - accuracy: 0.8972 - val\_loss: 1.5067 - val\_accuracy: 0.9019

Epoch 15/25

329/329 [==============================] - 167s 508ms/step - loss: 1.4430 - accuracy: 0.8993 - val\_loss: 1.3742 - val\_accuracy: 0.8941

Epoch 16/25

329/329 [==============================] - 167s 507ms/step - loss: 1.2975 - accuracy: 0.9027 - val\_loss: 1.2103 - val\_accuracy: 0.9103

Epoch 17/25

329/329 [==============================] - 172s 523ms/step - loss: 1.1681 - accuracy: 0.9042 - val\_loss: 1.0900 - val\_accuracy: 0.9099

Epoch 18/25

329/329 [==============================] - 166s 506ms/step - loss: 1.0555 - accuracy: 0.9056 - val\_loss: 0.9870 - val\_accuracy: 0.9128

Epoch 19/25

329/329 [==============================] - 167s 508ms/step - loss: 0.9559 - accuracy: 0.9084 - val\_loss: 0.9579 - val\_accuracy: 0.8904

Epoch 20/25

329/329 [==============================] - 167s 507ms/step - loss: 0.8681 - accuracy: 0.9092 - val\_loss: 0.8267 - val\_accuracy: 0.9084

Epoch 21/25

329/329 [==============================] - 167s 509ms/step - loss: 0.7909 - accuracy: 0.9110 - val\_loss: 0.7444 - val\_accuracy: 0.9141

Epoch 22/25

329/329 [==============================] - 168s 509ms/step - loss: 0.7235 - accuracy: 0.9124 - val\_loss: 0.7502 - val\_accuracy: 0.8918

Epoch 23/25

329/329 [==============================] - 168s 511ms/step - loss: 0.6664 - accuracy: 0.9128 - val\_loss: 0.6659 - val\_accuracy: 0.9028

Epoch 24/25

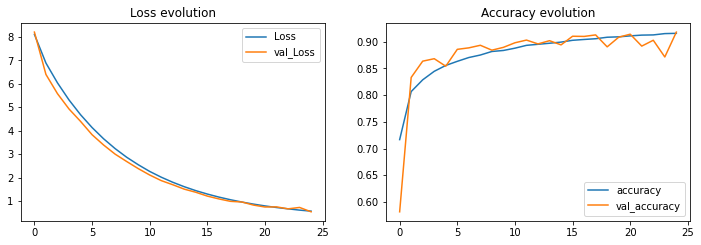
329/329 [==============================] - 168s 511ms/step - loss: 0.6118 - accuracy: 0.9152 - val\_loss: 0.7261 - val\_accuracy: 0.8715

Epoch 25/25

329/329 [==============================] - 168s 510ms/step - loss: 0.5685 - accuracy: 0.9157 - val\_loss: 0.5394 - val\_accuracy: 0.9180

313/313 [==============================] - 10s 31ms/step - loss: 0.5561 - accuracy: 0.9105

Test Accuracy : 0.910

****

**Experiment – VI**

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input\_shape=input\_shape))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(32, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), padding='same'))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(128, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(512,kernel\_regularizer=l2(0.01)))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_classes))

model.add(Activation('softmax'))

# summary of the model

print(model.summary())

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_1 (Conv2D) (None, 28, 28, 32) 320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation (Activation) (None, 28, 28, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization (BatchNo (None, 28, 28, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 26, 26, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_1 (Activation) (None, 26, 26, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_1 (Batch (None, 26, 26, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 13, 13, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_2 (Activation) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_2 (Batch (None, 13, 13, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_4 (Conv2D) (None, 11, 11, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_3 (Activation) (None, 11, 11, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_3 (Batch (None, 11, 11, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 5, 5, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 5, 5, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_5 (Conv2D) (None, 5, 5, 128) 73856

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_4 (Activation) (None, 5, 5, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_4 (Batch (None, 5, 5, 128) 512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_6 (Conv2D) (None, 3, 3, 128) 147584

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_5 (Activation) (None, 3, 3, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_5 (Batch (None, 3, 3, 128) 512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2 (None, 1, 1, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_2 (Dropout) (None, 1, 1, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 512) 66048

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_6 (Activation) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_3 (Dropout) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 5130

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_7 (Activation) (None, 10) 0

=================================================================

Total params: 359,402

Trainable params: 358,506

Non-trainable params: 896

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# compile

model.compile(loss='categorical\_crossentropy',

              optimizer='sgd',

              metrics=['accuracy'])

# Training the model

history = model.fit(np.expand\_dims(x\_train, -1), y\_train,

              batch\_size=batch\_size,

              epochs=epochs,

              validation\_split=0.3,

              shuffle=True)

Epoch 1/25

329/329 [==============================] - 200s 607ms/step - loss: 2.8932 - accuracy: 0.6532 - val\_loss: 3.9634 - val\_accuracy: 0.2551

Epoch 2/25

329/329 [==============================] - 198s 600ms/step - loss: 2.3172 - accuracy: 0.7659 - val\_loss: 2.1338 - val\_accuracy: 0.8034

Epoch 3/25

329/329 [==============================] - 199s 605ms/step - loss: 2.0241 - accuracy: 0.7978 - val\_loss: 1.8305 - val\_accuracy: 0.8354

Epoch 4/25

329/329 [==============================] - 196s 596ms/step - loss: 1.7845 - accuracy: 0.8243 - val\_loss: 1.6267 - val\_accuracy: 0.8532

Epoch 5/25

329/329 [==============================] - 196s 596ms/step - loss: 1.5842 - accuracy: 0.8402 - val\_loss: 1.4865 - val\_accuracy: 0.8421

Epoch 6/25

329/329 [==============================] - 201s 612ms/step - loss: 1.4173 - accuracy: 0.8511 - val\_loss: 1.3745 - val\_accuracy: 0.8526

Epoch 7/25

329/329 [==============================] - 196s 596ms/step - loss: 1.2708 - accuracy: 0.8606 - val\_loss: 1.2016 - val\_accuracy: 0.8604

Epoch 8/25

329/329 [==============================] - 195s 593ms/step - loss: 1.1485 - accuracy: 0.8670 - val\_loss: 1.1901 - val\_accuracy: 0.8270

Epoch 9/25

329/329 [==============================] - 195s 594ms/step - loss: 1.0425 - accuracy: 0.8731 - val\_loss: 1.0742 - val\_accuracy: 0.8404

Epoch 10/25

329/329 [==============================] - 196s 595ms/step - loss: 0.9485 - accuracy: 0.8748 - val\_loss: 0.8974 - val\_accuracy: 0.8798

Epoch 11/25

329/329 [==============================] - 194s 590ms/step - loss: 0.8652 - accuracy: 0.8806 - val\_loss: 0.8411 - val\_accuracy: 0.8749

Epoch 12/25

329/329 [==============================] - 194s 590ms/step - loss: 0.7959 - accuracy: 0.8828 - val\_loss: 0.7470 - val\_accuracy: 0.8872

Epoch 13/25

329/329 [==============================] - 193s 588ms/step - loss: 0.7286 - accuracy: 0.8881 - val\_loss: 0.8770 - val\_accuracy: 0.8127

Epoch 14/25

329/329 [==============================] - 193s 587ms/step - loss: 0.6763 - accuracy: 0.8885 - val\_loss: 0.6317 - val\_accuracy: 0.8957

Epoch 15/25

329/329 [==============================] - 193s 587ms/step - loss: 0.6227 - accuracy: 0.8910 - val\_loss: 0.5846 - val\_accuracy: 0.8988

Epoch 16/25

329/329 [==============================] - 194s 589ms/step - loss: 0.5826 - accuracy: 0.8928 - val\_loss: 0.5332 - val\_accuracy: 0.9033

Epoch 17/25

329/329 [==============================] - 194s 590ms/step - loss: 0.5434 - accuracy: 0.8952 - val\_loss: 0.5877 - val\_accuracy: 0.8707

Epoch 18/25

329/329 [==============================] - 194s 590ms/step - loss: 0.5093 - accuracy: 0.8970 - val\_loss: 0.5167 - val\_accuracy: 0.8865

Epoch 19/25

329/329 [==============================] - 193s 587ms/step - loss: 0.4786 - accuracy: 0.8987 - val\_loss: 0.5936 - val\_accuracy: 0.8479

Epoch 20/25

329/329 [==============================] - 194s 589ms/step - loss: 0.4514 - accuracy: 0.9027 - val\_loss: 0.4424 - val\_accuracy: 0.8972

Epoch 21/25

329/329 [==============================] - 198s 603ms/step - loss: 0.4283 - accuracy: 0.9020 - val\_loss: 0.3949 - val\_accuracy: 0.9119

Epoch 22/25

329/329 [==============================] - 194s 591ms/step - loss: 0.4090 - accuracy: 0.9038 - val\_loss: 0.4194 - val\_accuracy: 0.8953

Epoch 23/25

329/329 [==============================] - 195s 592ms/step - loss: 0.3880 - accuracy: 0.9054 - val\_loss: 0.3657 - val\_accuracy: 0.9126

Epoch 24/25

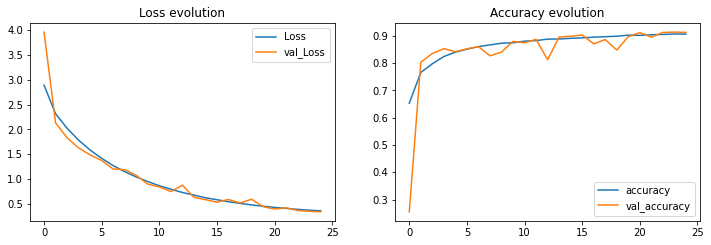
329/329 [==============================] - 194s 589ms/step - loss: 0.3725 - accuracy: 0.9068 - val\_loss: 0.3495 - val\_accuracy: 0.9138

Epoch 25/25

329/329 [==============================] - 199s 606ms/step - loss: 0.3590 - accuracy: 0.9065 - val\_loss: 0.3390 - val\_accuracy: 0.9128

313/313 [==============================] - 10s 32ms/step - loss: 0.3486 - accuracy: 0.9101

Test Accuracy : 0.910

****

**Experiment – VII**

model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input\_shape=input\_shape))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), padding='same'))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(Conv2D(128, (3, 3)))

model.add(Activation('relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(512,kernel\_regularizer=l2(0.01)))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_classes))

model.add(Activation('softmax'))

# summary of the model

print(model.summary())

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d (Conv2D) (None, 28, 28, 64) 640

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation (Activation) (None, 28, 28, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization (BatchNo (None, 28, 28, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_1 (Conv2D) (None, 26, 26, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_1 (Activation) (None, 26, 26, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_1 (Batch (None, 26, 26, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 13, 13, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 13, 13, 128) 73856

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_2 (Activation) (None, 13, 13, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_2 (Batch (None, 13, 13, 128) 512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 11, 11, 128) 147584

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_3 (Activation) (None, 11, 11, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_3 (Batch (None, 11, 11, 128) 512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 5, 5, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 5, 5, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 3200) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 512) 1638912

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_4 (Activation) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_2 (Dropout) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 5130

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation\_5 (Activation) (None, 10) 0

=================================================================

Total params: 1,904,586

Trainable params: 1,903,818

Non-trainable params: 768

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# compile

model.compile(loss='categorical\_crossentropy',

              optimizer='sgd',

              metrics=['accuracy'])

# Training the model

history = model.fit(np.expand\_dims(x\_train, -1), y\_train,

              batch\_size=batch\_size,

              epochs=epochs,

              validation\_split=0.3,

              shuffle=True)

Epoch 1/25

329/329 [==============================] - 463s 1s/step - loss: 9.0550 - accuracy: 0.7346 - val\_loss: 9.2667 - val\_accuracy: 0.5443

Epoch 2/25

329/329 [==============================] - 465s 1s/step - loss: 7.7520 - accuracy: 0.8191 - val\_loss: 7.2529 - val\_accuracy: 0.8233

Epoch 3/25

329/329 [==============================] - 464s 1s/step - loss: 6.7899 - accuracy: 0.8426 - val\_loss: 6.3181 - val\_accuracy: 0.8607

Epoch 4/25

329/329 [==============================] - 464s 1s/step - loss: 5.9678 - accuracy: 0.8584 - val\_loss: 5.5437 - val\_accuracy: 0.8811

Epoch 5/25

329/329 [==============================] - 464s 1s/step - loss: 5.2564 - accuracy: 0.8685 - val\_loss: 4.9241 - val\_accuracy: 0.8686

Epoch 6/25

329/329 [==============================] - 462s 1s/step - loss: 4.6361 - accuracy: 0.8739 - val\_loss: 4.3851 - val\_accuracy: 0.8626

Epoch 7/25

329/329 [==============================] - 462s 1s/step - loss: 4.0903 - accuracy: 0.8817 - val\_loss: 3.8983 - val\_accuracy: 0.8619

Epoch 8/25

329/329 [==============================] - 466s 1s/step - loss: 3.6113 - accuracy: 0.8884 - val\_loss: 3.3688 - val\_accuracy: 0.8974

Epoch 9/25

329/329 [==============================] - 464s 1s/step - loss: 3.1969 - accuracy: 0.8920 - val\_loss: 2.9908 - val\_accuracy: 0.8979

Epoch 10/25

329/329 [==============================] - 463s 1s/step - loss: 2.8354 - accuracy: 0.8940 - val\_loss: 2.7174 - val\_accuracy: 0.8787

Epoch 11/25

329/329 [==============================] - 462s 1s/step - loss: 2.5109 - accuracy: 0.9004 - val\_loss: 2.4855 - val\_accuracy: 0.8521

Epoch 12/25

329/329 [==============================] - 457s 1s/step - loss: 2.2327 - accuracy: 0.9027 - val\_loss: 2.0920 - val\_accuracy: 0.9069

Epoch 13/25

329/329 [==============================] - 457s 1s/step - loss: 1.9846 - accuracy: 0.9058 - val\_loss: 1.8858 - val\_accuracy: 0.9013

Epoch 14/25

329/329 [==============================] - 460s 1s/step - loss: 1.7694 - accuracy: 0.9081 - val\_loss: 1.8884 - val\_accuracy: 0.8473

Epoch 15/25

329/329 [==============================] - 459s 1s/step - loss: 1.5805 - accuracy: 0.9109 - val\_loss: 1.5384 - val\_accuracy: 0.8970

Epoch 16/25

329/329 [==============================] - 459s 1s/step - loss: 1.4112 - accuracy: 0.9142 - val\_loss: 1.3444 - val\_accuracy: 0.9115

Epoch 17/25

329/329 [==============================] - 459s 1s/step - loss: 1.2648 - accuracy: 0.9160 - val\_loss: 1.2179 - val\_accuracy: 0.9067

Epoch 18/25

329/329 [==============================] - 458s 1s/step - loss: 1.1382 - accuracy: 0.9170 - val\_loss: 1.0813 - val\_accuracy: 0.9168

Epoch 19/25

329/329 [==============================] - 459s 1s/step - loss: 1.0235 - accuracy: 0.9190 - val\_loss: 1.0360 - val\_accuracy: 0.8966

Epoch 20/25

329/329 [==============================] - 455s 1s/step - loss: 0.9240 - accuracy: 0.9212 - val\_loss: 0.8775 - val\_accuracy: 0.9228

Epoch 21/25

329/329 [==============================] - 453s 1s/step - loss: 0.8381 - accuracy: 0.9220 - val\_loss: 0.8099 - val\_accuracy: 0.9195

Epoch 22/25

329/329 [==============================] - 454s 1s/step - loss: 0.7596 - accuracy: 0.9248 - val\_loss: 0.7353 - val\_accuracy: 0.9227

Epoch 23/25

329/329 [==============================] - 455s 1s/step - loss: 0.6944 - accuracy: 0.9257 - val\_loss: 0.7989 - val\_accuracy: 0.8750

Epoch 24/25

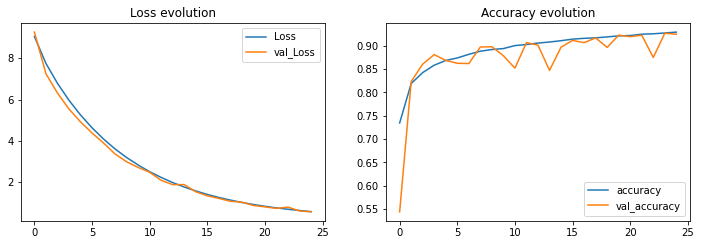
329/329 [==============================] - 455s 1s/step - loss: 0.6333 - accuracy: 0.9274 - val\_loss: 0.6125 - val\_accuracy: 0.9269

Epoch 25/25

329/329 [==============================] - 453s 1s/step - loss: 0.5804 - accuracy: 0.9293 - val\_loss: 0.5708 - val\_accuracy: 0.9248

313/313 [==============================] - 24s 76ms/step - loss: 0.5853 - accuracy: 0.9196

Test Accuracy : 0.920

****

**Results Table**

**Conditions:**

**Epochs = 25, Batch Size = 128, Loss = categorical\_crossentropy, optimizer = sgd, metrics = accuracy**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Experiment** | **Accuracy** | **Value Accuracy** | **Loss** | **Value Loss** | **Test Accuracy** |
| **I** | **0.8664** | **0.8782** | **0.365** | **0.330** | **0.87** |
| **II** | **0.9467** | **0.9104** | **0.145** | **0.2542** | **0.905** |
| **III** | **0.9038** | **0.9003** | **0.2593** | **0.2663** | **0.893** |
| **IV** | **0.9562** | **0.8397** | **0.4785** | **0.8301** | **0.831** |
| **V** | **0.9157** | **0.8715** | **0.5685** | **0.7261** | **0.91** |
| **VI** | **0.9065** | **0.9128** | **0.359** | **0.339** | **0.91** |
| **VII** | **0.9293** | **0.9248** | **0.5804** | **0.5708** | **0.92** |