UNIVERSITY INSTITUTE OF ENGINEERING AND TECHNOLOGY
PANJAB UNIVERSITY, CHANDIGARH - 160014, INDIA

Project report on

Handwritten Digit Recognition

Submitted By :

Name	Roll Number
Shruti	UE173098
Riya Aggarwal	UE173082
Visarika Vaidya	UE173119

Department of Computer Science and Engineering,
University Institute of Engineering and Technology,
Panjab University, Chandigarh-160014, India
November 2020

ABSTRACT

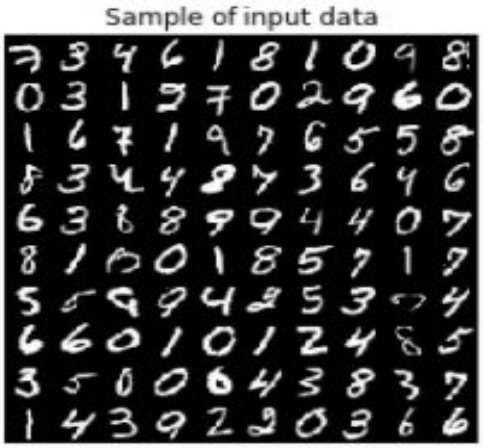
In recent times, with the increase of Artificial Neural Network (ANN), deep learning has brought a dramatic twist in the field of machine learning by making it more artificially intelligent. Deep learning is remarkably used in vast ranges of fields because of its diverse range of applications such as pattern recognition, speech recognition, face recognition, document analysis, handwritten digit recognition etc. Handwritten digit recognition is the ability of computers to recognize human handwritten digits. It is a hard task for the machine because handwritten digits are not perfect and can be made with many different flavors. The handwritten digit recognition is the solution to this problem which uses the image of a digit and recognizes the digit present in the image. In this project we aim to use CNN (Convolutional Neural Network) to train our model over the MNIST dataset. The library used will be Keras which is a very simple neural network library written over tensorflow.

CONTENTS

INTRODUCTION

Handwritten digit recognition is a classic problem in the field of image recognition.The shape of the digits and its features help identify the digit from the strokes and boundaries. There have been great achievements in recent years in the field of pattern recognition, particularly in the field of Handwritten digit recognition problem .Handwriting recognition is the ability of a device to take handwriting as input from sources. The handwriting taken as input can be used to verify signatures, used to interpret text and OCR (optical character recognition) to read the text and transform it into a form which can be manipulated by computer. The traditional machine learning algorithms are shallow learning algorithms and incapable of extracting multiple features. In the era of big data, deep learning algorithms have performed efficiently in digit recognition tasks on MNIST dataset.

Fig.1 shows the samples of digits in MNIST dataset. Neural networks and deep learning have proven to perform exceptionally well in recognising handwritten digits.



Artificial neural networks are machine learning algorithms that are an implementation of the neural structure of the brain. Similar to the brain, the neural networks take input and each input has a weight. The weight and bias along with the input are fed to hidden layers, and then based on activation values of the calculations the calculations are forwarded to the output layer. The neural networks with one hidden layer are called shallow neural networks. The shallow neural networks are incapable of training datasets that require multiple feature extraction. Hence, Deep neural networks were introduced. Deep neural networks are the neural networks with more than one hidden layer. In this each hidden layer learns a different feature. The state of art neural networks recently evolved into Deep learning algorithms which mimic the functions of human cerebral cortex in their implementation.

Proposed Methods :

Datasets :-

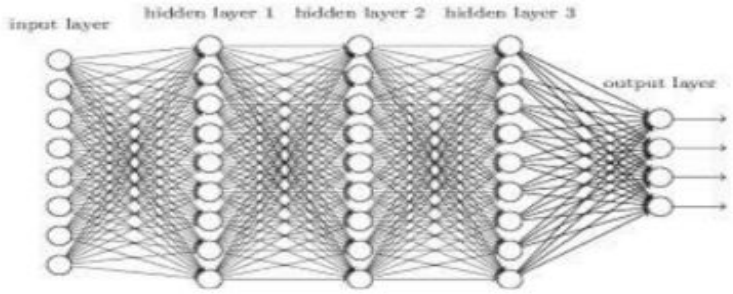
The handwritten digit recognition system uses the MNIST data set . It has 70,000 images that can be used to train and evaluate the system. The train set has 60,000 images and the test set has

10,000 images. It is the subset of NIST dataset (National institute of standards and technology) , having 28 x 28 size input images and 10 class labels from (0-9). Therefore, the size of the image is 28 x 28 pixel square i.e 784 pixels. The dataset is fed to 4 classification algorithms, namely : shallow neural networks, Deep neural networks (3-layer), Convolutional neural networks, and Recurrent neural networks.

Classifiers:-

1. **Deep Neural Networks:-**

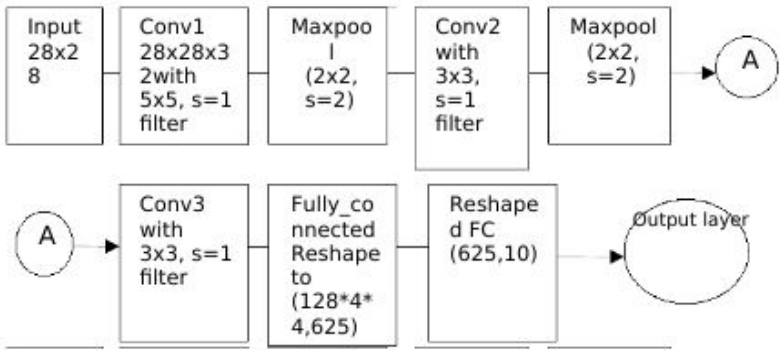
   The Deep neural networks are implemented in two ways: 4-layer. The 4-layer deep neural network uses a multilayer perceptron classifier or a deep neural network with 3 hidden layers and one output layer. The hyper parameters for the model are : number of neurons in hidden layers is 200,150 and 100 respectively, learning rate 0.005, batch size of 128 and number of epochs is 10. The number of neurons in the output layer is 10. The architecture uses Relu activation for the hidden layers and softmax activation for the output layer.



4-layer Neural network Architecture

2. **Convolutional Neural Networks:-**

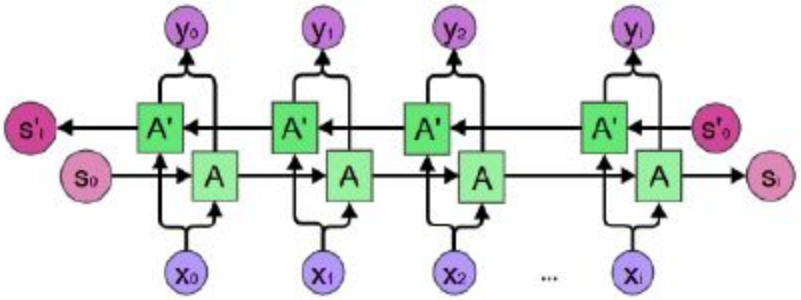
   Convolutional neural networks deal with image data, typically 2D data and uses convolution, pooling and fully connected layers to classify the data and produce the output. The three main features of CNN are : Local Receptive field, shared weight and biases and pooling. The convolution layers use the convolution operation between the input image and the filter or kernel. The filter or kernel is also a 2D matrix that is responsible for generating feature maps using the local receptive field. Local Receptive field is a small localised field of the input image connected to a single neuron in the feature map. The number of feature maps is dependent on the number of features to be classified. The kernel acts as a weight matrix and learns the weights after the feature map detects the features.



CNN Architecture

3. **Recurrent Neural Network :**

RNN allow the information to be continuous and lasting by using a loop. It processes the input one at a time in sequence and updates the state of the vector which has data about past elements. Traditionally, the neural networks give input simultaneously, are independent of one another and have different parameters. The recurrent neural networks process one input at a time, the weight and shared bias parameters are the same and dependent on one another. Here, we have used a bidirectional RNN which states that output will depend on previous and future data elements in sequence. The RNN run in opposite directions and the outputs of both are mixed. One executes the process in a direction and the other runs in the opposite direction. The architecture of the model has input number as 28, number of steps as 28, number of hidden neurons as128 and output labels as 10. The learning rate is 0.001, training iterations are 100000,batch size is 128 and display step is 10. The optimizer is Adam optimizer with default values. Two LSTM cells are defined in the model and the model is trained.



RNN Architecture

DNN, CNN and Bidirectional RNN are implemented on MNIST dataset with varying accuracy. The accuracy of the algorithms is tabulated below:

Algorithm	Accuracy
Deep Neural Networks (4-Layer)	97.74%
Convolutional neural networks	99.6%
Bidirectional Recurrent neural networks	99.2%

Hence, CNN performed has the best accuracy on MNIST dataset of 99.6%. Bidirectional RNN has the accuracy of 98.43% on training dataset and 99.2% on testing a dataset. The 4-layer DNN has the least accuracy of 97.4%. This is because the convolution neural network uses feature maps to learn the features from an image.

SPECIFICATION AND DESIGN

In this project, we designed a CNN model by using python language. We used an open-source software library called TensorFlow which is widely used for machine learning applications such as neural network and used Keras, which works as wrapper and it is a high-level neural network library that's built on top of TensorFlow.

CONVOLUTIONAL NEURAL NETWORK :

Convolutional neural network is a type of artificial neural network that uses multiple perceptrons that analyze image inputs and have learnable weights and bases to several parts of images and are able to segregate each other. One advantage of using Convolutional Neural Network is it leverages the use of local spatial coherence in the input images, which allow them to have fewer weights as some parameters are shared. This process is clearly efficient in terms of memory and complexity. The basic building blocks of convolutional neural networks are as follows:

a. Convolution Layer – In the convolutional layer, a matrix named kernel is passed over the input matrix to create a feature map for the next layer. We execute a mathematical operation called convolution by sliding the Kernel matrix over the input matrix. At every location, an element wise matrix multiplication is performed and sums the result onto the feature map. Convolution is a specialized kind of linear operation which is widely used in a variety of domains including image processing, statistics, physics.

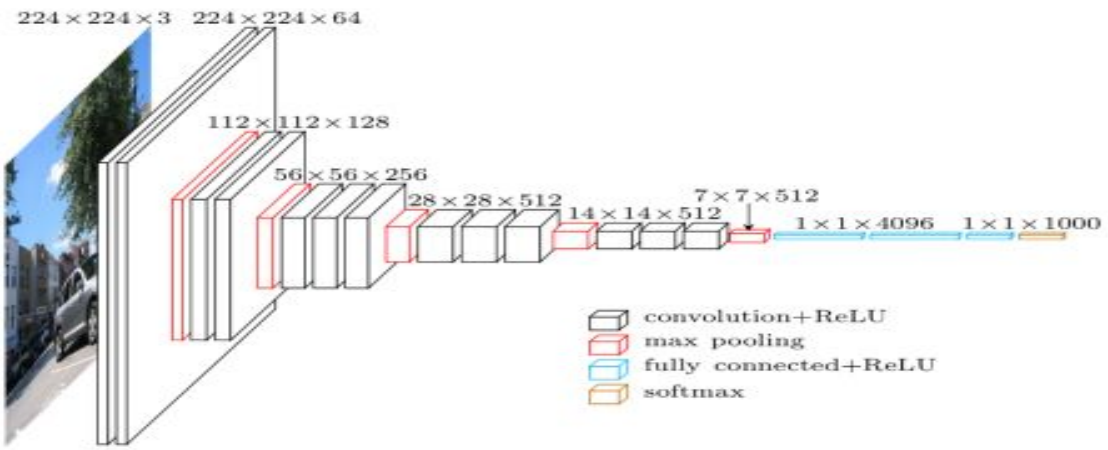
b. Non-linear activation functions (ReLU) – Activation function is a node that comes after the convolutional layer and the activation function is the nonlinear transformation that we do over the input signal. The rectified linear unit activation function (ReLU) is a piecewise linear function that will output the input if it is positive, otherwise it will output zero.

c. Pooling Layer – The drawback of the feature map output of convolutional layer is that it records the precise position of features in the input. This means during cropping, rotation or any other minor changes to the input image will completely result in a different feature map. To counter this problem, we approach down sampling of convolutional layers. Down sampling can be achieved by applying a pooling layer after nonlinearity layer. Pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.

d. Fully Connected Layer - At the end of a convolutional neural network, the output of the last Pooling Layer acts as input to the Fully Connected Layer. There can be one or more of these layers. Fully connected means that every node in the first layer is connected to every node in the second layer.

VGG-16 Model : The precise structure of the VGG-16 network shown in Figure. 7. is as follows:

- The first and second convolutional layers consist of 64 feature kernel filters and the size of the filter is 3×3. As the input image (RGB image with depth 3) passed into the first and second convolutional layer, dimensions change to 224x224x64. Then the resulting output is passed to the max pooling layer with a stride of 2.
- The third and fourth convolutional layers are of 124 feature kernel filters and size of filter is 3×3. These two layers are followed by a max pooling layer with stride 2 and the resulting output will be reduced to 56x56x128.
- The fifth, sixth and seventh layers are convolutional layers with kernel size 3×3. All three use 256 feature maps. These layers are followed by a max pooling layer with stride 2.
- Eighth to thirteen are two sets of convolutional layers with kernel size 3×3. All these sets of convolutional layers have 512 kernel filters. These layers are followed by a max pooling layer with stride of 1.
- Fourteen and fifteen layers are fully connected hidden layers of 4096 units followed by a softmax output layer (Sixteenth layer) of 1000 units.



PROGRESS TILL DATE ALONG WITH RESULTS

1. IMPORT THE LIBRARIES

First of all we imported all the modules that we are going to need for training our model. The Keras library already contains some datasets and MNIST is one of them.

```
[1]  import numpy as np
     import pandas as pd

[2]  from tensorflow.keras.datasets import mnist

[3]  (x_train,y_train),(x_test,y_test) = mnist.load_data()
```

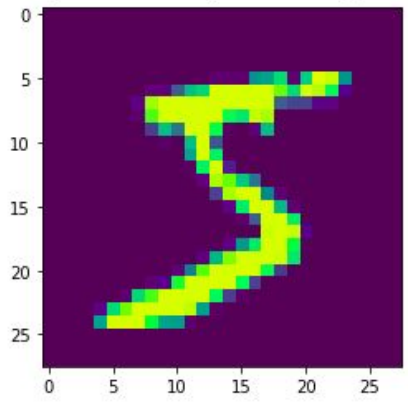
After fetching 70,000 handwritten digits from MNIST dataset using "*from keras.datasets import mnist*" module we segregated it into Training and Test set. The training set consists of 60,000 images and the test set consists the rest of the 10,000 images.
The mnist.load_data() method returns us the training data, its labels and also the testing data and its labels.

2. VISUALIZING THE IMAGE DATA

```
[6]
    import  matplotlib.pyplot as plt
    %matplotlib inline

[8]
    x_train.shape

[9]  single_image = x_train[0]

[10] single_image

[11] single_image.shape
```

The first two lines of code displays the image and the last line prints the value of the digit corresponding to that image.

```
<matplotlib.image.AxesImage at 0x7faad3e8e080>
```



3. PREPROCESSING THE DATA

The image data cannot be fed directly into the model so we performed some operations and processed the data to make it ready for our neural network. The dimension of the training data is (60000,28,28). The CNN model will require one more dimension so we reshaped the matrix to shape (60000,28,28,1). The first number is the number of images (Y_train -> 60000, Y_test -> 10000). Then comes the shape of each image i.e. (28, 28). The last number 1 signifies that the image is grayscale.

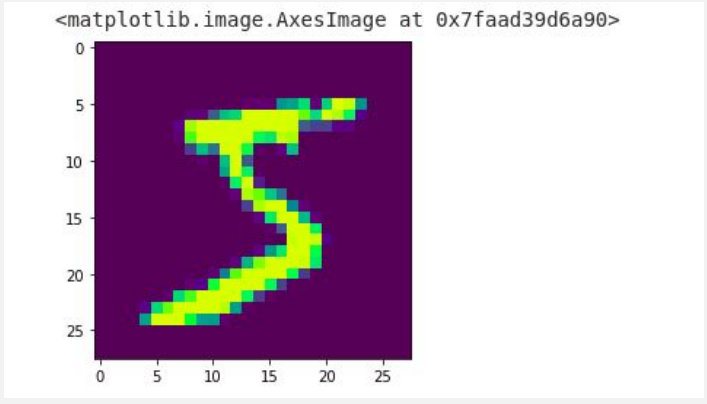
```
[36] y_train

     array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)

[37] y_test

     array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

4. ONE-HOT ENCODING

The final layer of our CNN model contains 10 nodes, each of them corresponding to the respective digit (first node -> 0, second node-> 1 and so on). So, when we will feed an image into the model, the model will return probabilities of that digit according to each node. So, at the end, the predicted digit will be corresponding to the node with highest probability. For example, if the first node has the highest probability, then the predicted digit is 0. For this whole operation, the model expects each of the labels to be in the form of an array of 10 elements in which only one of the elements = 1 (the element/node with highest probability) and rest = 0. So, for that, we need to hot encode our variables.

For example, if the image is of the number 6, then the label instead of being = 6, it will have a value 1 in column 7 and 0 in the rest of the columns, like [0,0,0,0,0,0,1,0,0]. Module used for this is *"from keras.utils import to_categorical"*.

```
[15] from tensorflow.keras.utils import to_categorical

[17] y_example = to_categorical(y_train)

[19] y_example.shape

[21] y_cat_test = to_categorical(y_test,num_classes = 10)

[22] y_cat_train = to_categorical(y_train,10)

[25] x_train = x_train/255
     x_test = x_test/255

[29] plt.imshow(scaled_single)
```



5. BUILDING THE MODEL

The model type that we used is Sequential. Here, comes the use of **"from keras.models import Sequential".**

Sequential is the easiest way to build a model in Keras. It allows us to build the model layer by layer. add() function is used for adding successive layers.
The first 2 layers are Conv2D layers. These are convolution layers that will deal with our input images, which are seen as 2D matrices.
Here, we used 32 nodes in the first layer and 64 nodes in the second layer. These numbers can be adjusted accordingly, depending on the size of the dataset. In this case, 32 and 64 seem to be working just fine.
Kernel Size is the size of the filter matrix for our convolution. So, kernel size 3 means that a 3x3 filter matrix is going to be used.
Activation is the activation function for the layer. The activation function here being used for the first 2 layers is the ReLU, or Rectified Linear Activation. This function outputs 0 if the input is a negative number and output the same input if the input is a positive number. In simple words, ReLU->max(0, input). This activation function is known for performing well in terms of speed and output in the neural nets.
Here, comes the use of "from keras.layers import Dense, Flatten" and "from keras.layers.convolutional import Conv2D"

Declare the model

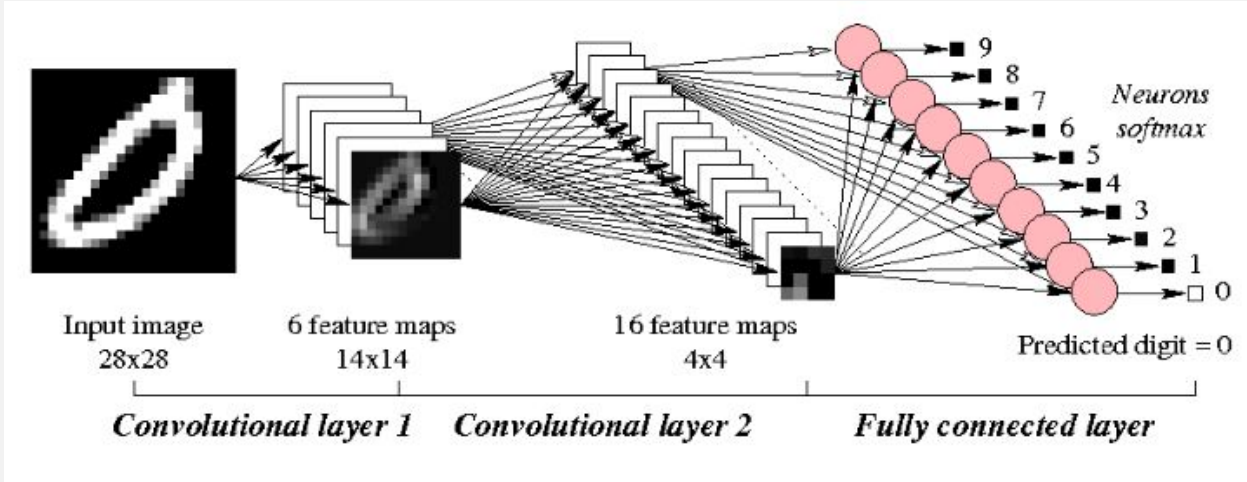
```
model = Sequential()
```

Declare the layers

```
layer_1 = Conv2D(32, kernel_size=3, activation='relu', input_shape=(28, 28, 1))
layer_2 = Conv2D(64, kernel_size=3, activation='relu')
layer_3 = Flatten()
layer_4 = Dense(10, activation='softmax')
```

##Flow of the model

- The first layer takes in an input of shape, here, being 28, 28, 1 where 1 signifies greyscale.
- In between, the Conv2D layers and the dense layer, there is a "Flatten" layer. Flatten serves as a connection between convolutional and dense layers.
- 'Dense' is the layer type which is being used for the output layer. Dense is a standard layer type that is used in many cases for neural networks.
- We will have 10 nodes in our output layer, one for each possible outcome (0–9)
- The activation function is 'softmax'. Softmax makes the output sum up tp 1, so that the output contains a series of probabilities.
- The model will predict the one with the highest probability.



Model architecture

6. COMPILING THE MODEL

Compiling the model takes three parameters:

- Optimizer — It controls the learning rate. We will be using an 'Adam' optimizer. It is a very good optimizer as it utilises the perks of both Stochastic gradient and RMSprop optimizers.
- Loss function — We will be using 'categorical_crossentropy' loss function. It is the most common choice for classification. A lower score corresponds to better performance.
- Metrics — To make things easier to interpret, we will be using 'accuracy' metrics to see the accuracy score on the validation set while training the model.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

7. TRAINING THE MODEL

Now we'll train the above model with specified characteristics.

So, the model will train on (X_train, y_train) and it will get validated on (X_test, y_test). Setting the number of epochs to 3. Number of epochs can be increased for better accuracy but here with even 3 epochs a remarkable accuracy gets achieved.

1 epoch -> One iteration/cycle of the dataset throughout the Neural Network

```
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3)
```

Output

Train on 60000 samples, validate on 10000 samples

Epoch 1/3

60000/60000 [==============================] - 87s 1ms/step - loss: 1.4928 - acc: 0.8784 - val_loss: 0.0714 - val_acc: 0.9774

Epoch 2/3

60000/60000 [==============================] - 60s 1ms/step - loss: 0.0637 - acc: 0.9813 - val_loss: 0.0730 - val_acc: 0.9789

Epoch 3/3
60000/60000 [==============================] - 61s 1ms/step - loss: 0.0456 - acc: 0.9859 - val_loss: 0.0709 - val_acc: 0.9792
We can see that our model achieved a remarkable validation accuracy of 97.92%.

FUTURE WORK PLAN

The application of this handwritten digit Recognition algorithm is extensive. Now-a-days recent advancement in technologies has pushed the limits further for man to get rid of older equipment which posed inconvenience in using. In our case that equipment is a keyboard.

There are many situations when using a keyboard is cumbersome like:

1. We don't get fluency with a keyboard as real word writing.
2. When any key on the keyboard is damaged.
3. Keyboards have scripts on its keys in only one language.
4. We have to find each digit on the keyboard which takes time.
5. In touch-enabled portable devices it is difficult to add a keyboard with much ease.