**A PROJECT REPORT**

**ON**

**HAND WRITING DIGIT RECOGNITION**

Submitted in partial fulfillment for the requirement of the award of

TRAINING

IN

Data Analytics, Machine Learning and AI using Python



Submitted By

**Supriya G (**Ramaiah Institute of Technology ,Bangalore**)**

Under the guidance of

**Mr. BipulShahi**

**ACKNOWLEDGEMENT**

My sincere gratitude and thanks towards my project paper guide BipulShahi, Corporate Trainer, Developer, Traveller !!! IOT, Artificial Intelligence, Robotics, Cloud Computing, Android Apps !!!

It was only with his backing and support that I could complete the report. He provided me all sorts of help and corrected me if ever seemed to make mistakes. I have no such words to express my gratitude. I acknowledge my sincere gratitude to the HOD of Electronics and Telecommunication Department, Ramaiah Institute of Technology, Bangalore. She gave me the permission to do the project work. Without her support I couldn’t even start the work. So I am grateful to her. I acknowledge my sincere gratitude to the lecturers, research scholars and the lab technicians for their valuable guidance and helping attitude even in their very busy schedule. And at last but not the least, I acknowledge my dearest parents for being such a nice source of encouragement and moral support that helped me tremendously in this aspect. I also declare to the best of my knowledge and belief that the Project Work has not been submitted anywhere else.

**INTRODUCTION**

There is an ever-increasing amount of image data in the world, and the rate of growth itself is increasing. Info trends estimates that in 2016 still cameras and mobile devices captured more than 1.1 trillion images. According to the same estimate, in 2020 the figure will increase to 1.4 trillion. Many of these images are stored in cloud services or published on the Internet. In 2014, over 1.8 billion images were uploaded daily to the most popular platforms, such as Instagram and Facebook.

Going beyond consumer devices, there are cameras all over the world that capture images for automation purposes. Cars monitor the road, and traffic cameras monitor the same cars. Robots need to understand a visual scene in order to smartly build devices and sort waste. Imaging devices are used by engineers, doctors and space explorers alike.

To effectively manage all this data, we need to have some idea about its contents. Automated processing of image contents is useful for a wide variety of image-related tasks. For computer systems, this means crossing the so-called semantic gap between the pixel level information stored in the image files and the human understanding of the same images. Computer vision attempts to bridge this cap.

**Problem statement**

Digits contained in image files can be located and identified automatically. This is called object detection and is one of the basic problems of computer vision. As we will demonstrate, convolutional neural networks are currently the state-of-the-art solution for object detection. The main task of this thesis is to review and test convolutional object detection methods.

**Technology and Concepts**

**Machine Learining**

Learning algorithms are widely used in computer vision applications. Before considering image related tasks, we are going to have a brief look at basics of machine learning.

Machine learning has emerged as a useful tool for modelling problems that are otherwise difficult to formulate exactly. Classical computer programs are explicitly programmed by hand to perform a task. With machine learning, some portion of the human contribution is replaced by a learning algorithm. As availability of computational capacity and data has increased, machine learning has become more and more practical over the years, to the point of being almost ubiquitous.

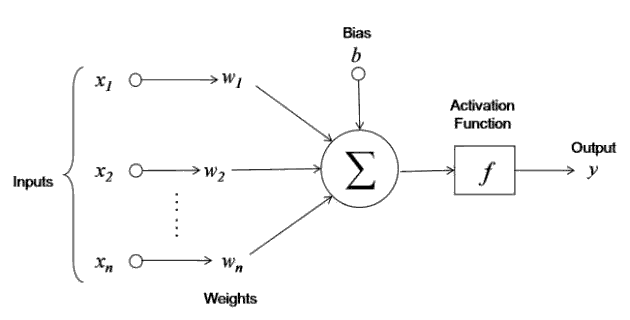
It can be used in two ways:

* Supervised Learning
* Unsupervised Learning

**Neural networks**

Neural networks are a popular type of machine learning model. A special case of a neural network called the convolutional neural network (CNN) is the primary focus of this thesis. Before discussing CNNs, we will discuss how regular neural networks work.

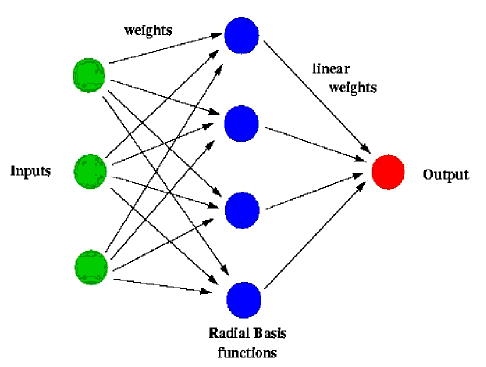
Neural networks were originally called artificial neural networks, because they were developed to mimic the neural function of the human brain.



The neuron is trained by carefully selecting the weights to produce a desired output for each input.

**Multi-layer networks**

A neural network is a combination of artificial neurons. The neurons are typically grouped into layers.



A multi-layer network typically includes three types of layers: an input layer, one or more hidden layers and an output layer. The input layer usually merely passes data along without modifying it. Most of the computation happens in the hidden layers. The output layer converts the hidden layer activations to an output, such as a classification. A multilayer feed-forward network with at least one hidden layer can function as a universal approximator.

**Computer vision**

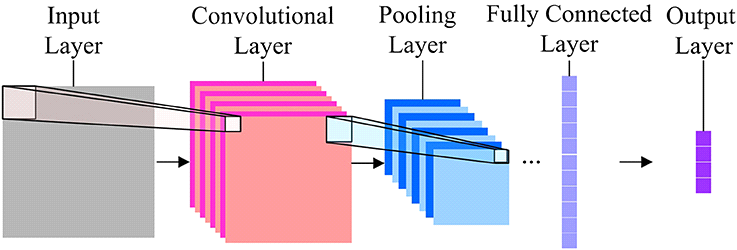
Computer vision deals with the extraction of meaningful information from the contents of digital images or video. This is distinct from mere image processing, which involves manipulating visual information on the pixel level. Applications of computer vision include image classification, visual detection, 3D scene reconstruction from 2D images, image retrieval, augmented reality, machine vision and traffic automation.

**Object detection**

Object detection is one of the classical problems of computer vision and is often described as a difficult task. In many respects, it is similar to other computer vision tasks, because it involves creating a solution that is invariant to deformation and changes in lighting and viewpoint. What makes object detection a distinct problem is that it involves both locating and classifying regions of an image. The locating part is not needed in, for example, whole image classification.

To detect an object, we need to have some idea where the object might be and how the image is segmented. This creates a type of chicken-and-egg problem, where, to recognize the shape (and class) of an object, we need to know its location, and to recognize the location of an object, we need to know its shape. Some visually dissimilar features, such as the clothes and face of a human being, may be parts of the same object, but it is difficult to know this without recognizing the object first. On the other hand, some objects stand out only slightly from the background, requiring separation before recognition.

**Convolution neural networks**



The basic idea of the CNN was inspired by a concept in biology called the receptive field. Receptive fields are a feature of the animal visual cortex. They act as detectors that are sensitive to certain types of stimulus, for example, edges. They are found across the visual field and overlap each other.

**Description of the dataset**

In this paper, we used the MNIST database consisting of offline handwritten digits ranging from 0-9. The database was constructed from Special Database 3 (SD-3) and Special Database 1 (SD-1) that contain binary images of handwritten digits. SD-3 was collected among Census Bureau employees, while SD-1 was collected among high-school students. For the results to be independent of both datasets, MNIST dataset was built by mixing NIST SD-1 and SD-3. The total number of digit image samples (70,000), the total number for training (60,000) and testing (10,000), and the subtotal number for each digit are shown in table 1. Each digit is a gray-level fixed-size image with a size of 28 x 28 (or 784 pixels) in total as the features.

|  |  |  |  |
| --- | --- | --- | --- |
| Dataset Digits | Training | Testing | Subtotal |
| 9 | 5949 | 1009 | 6958 |
| 8 | 5851 | 974 | 6825 |
| 7 | 6265 | 1028 | 7293 |
| 6 | 5918 | 958 | 6876 |
| 5 | 5421 | 892 | 6313 |
| 4 | 5842 | 982 | 6824 |
| 3 | 6131 | 1010 | 7141 |
| 2 | 5958 | 1032 | 6990 |
| 1 | 6742 | 1135 | 7877 |
| 0 | 5923 | 980 | 6903 |
| Total | 60,000 | 10,000 | 70,000 |

**Experimental setup**

We can develop this model with five key elements**.** They are the

* loading of the dataset
* the preparation of the dataset
* the definition of the model
* the evaluation of the model
* Finalize the model and make predictions.

**1. Load Dataset**

We know some things about the dataset. For example, we know that the images are all pre-aligned (e.g. each image only contains a hand-drawn digit), that the images all have the same square size of 28×28 pixels, and that the images are grayscale. Therefore, we can load the images and reshape the data arrays to have a single color channel.

We also know that there are 10 classes and that classes are represented as unique integers. We can, therefore, use a one hot encoding for the class element of each sample, transforming the integer into a 10 element binary vector with a 1 for the index of the class value, and 0 values for all other classes. We can achieve this with the **to\_categorical()** utility function.

Import the libraries that are required for the implementation of our model.

from numpy import mean

from numpy import std

from matplotlib import pyplot

from sklearn.model\_selection import KFold

from keras.datasets import mnist

from keras.utils import to\_categorical

from keras.models import Sequential

from keras.layers import Conv2D

from keras.layers import MaxPooling2D

from keras.layers import Dense

from keras.layers import Flatten

from keras.optimizers import SGD

The  load\_dataset() function implements these behaviors and can be used to load the dataset.

def load\_dataset():

# load dataset

(trainX, trainY), (testX, testY) = mnist.load\_data()

# reshape dataset to have a single channel

trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))

testX = testX.reshape((testX.shape[0], 28, 28, 1))

# one hot encode target values

trainY = to\_categorical(trainY)

testY = to\_categorical(testY)

return trainX, trainY, testX, testy

**2. Preparation of the dataset**

We know that the pixel values for each image in the dataset are unsigned integers in the range between black and white, or 0 and 255.We do not know the best way to scale the pixel values for modeling, but we know that some scaling will be required. A good starting point is to [**normalize the pixel values**](https://machinelearningmastery.com/how-to-normalize-center-and-standardize-images-with-the-imagedatagenerator-in-keras/) of grayscale images, e.g. rescale them to the range [0 ,1]. This involves first converting the data type from unsigned integers to floats, then dividing the pixel values by the maximum value.

The prep\_pixels() function below implements these behaviors and is provided with the pixel values for both the train and test datasets that will need to be scaled.

# scale pixels

def prep\_pixels(train, test):

# convert from integers to floats

train\_norm = train.astype('float32')

test\_norm = test.astype('float32')

# normalize to range 0-1

train\_norm = train\_norm / 255.0

test\_norm = test\_norm / 255.0

# return normalized images

return train\_norm, test\_norm

**3. Defining the model**

There are many ways to change the model configuration in order to explore improvements over the baseline model. Two common approaches involve changing the [capacity](https://machinelearningmastery.com/how-to-control-neural-network-model-capacity-with-nodes-and-layers/) of the feature extraction part of the model or changing the capacity or function of the classifier part of the model. Perhaps the point of biggest influence is a change to the feature extractor.

We can increase the depth of the feature extractor part of the model, following a [VGG-like pattern](https://machinelearningmastery.com/how-to-implement-major-architecture-innovations-for-convolutional-neural-networks/) of adding more convolutional and pooling layers with the same sized filter, while increasing the number of filters. In this case, we will add a double convolutional layer with 64 filters each, followed by another max pooling layer.

The updated version of the define\_model() function with this change is listed below.

# define cnn model

def define\_model():

model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', kernel\_initializer='he\_uniform', input\_shape=(28, 28, 1)))

model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu', kernel\_initializer='he\_uniform'))

model.add(Conv2D(64, (3, 3), activation='relu', kernel\_initializer='he\_uniform'))

model.add(MaxPooling2D((2, 2)))

model.add(Flatten())

model.add(Dense(100, activation='relu', kernel\_initializer='he\_uniform'))

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

# compile model

opt = SGD(lr=0.01, momentum=0.9)

model.compile(optimizer=opt, loss='categorical\_crossentropy', metrics=['accuracy'])

return model

**4. Evaluation of the model**

After the model is defined, we need to evaluate it. The model will be evaluated using [five-fold cross-validation](https://machinelearningmastery.com/k-fold-cross-validation/). The value of k=5 was chosen to provide a baseline for both repeated evaluation and to not be so large as to require a long running time. Each test set will be 20% of the training dataset, or about 12,000 examples, close to the size of the actual test set for this problem. The training dataset is shuffled prior to being split, and the sample shuffling is performed each time, so that any model we evaluate will have the same train and test datasets in each fold, providing an apples-to-apples comparison between models. We will train the model for a modest 10 training epochs with a default batch size of 32 examples. The test set for each fold will be used to evaluate the model both during each epoch of the training run, so that we can later create learning curves, and at the end of the run, so that we can estimate the performance of the model. As such, we will keep track of the resulting history from each run, as well as the classification accuracy of the fold.

The evaluate\_model() function below implements these behaviors, taking the training dataset as arguments and returning a list of accuracy scores and training histories that can be later summarized.

# evaluate a model using k-fold cross-validation

def evaluate\_model(dataX, dataY, n\_folds=5):

scores, histories = list(), list()

# prepare cross validation

kfold = KFold(n\_folds, shuffle=True, random\_state=1)

# enumerate splits

for train\_ix, test\_ix in kfold.split(dataX):

# define model

model = define\_model()

# select rows for train and test

trainX, trainY, testX, testY = dataX[train\_ix], dataY[train\_ix], dataX[test\_ix], dataY[test\_ix]

# fit model

history = model.fit(trainX, trainY, epochs=10, batch\_size=32, validation\_data=(testX, testY), verbose=0)

# evaluate model

\_, acc = model.evaluate(testX, testY, verbose=0)

print('> %.3f' % (acc \* 100.0))

# stores scores

scores.append(acc)

histories.append(history)

return scores, histories

def summarize\_diagnostics(histories):

for i in range(len(histories)):

# plot loss

pyplot.subplot(2, 1, 1)

pyplot.title('Cross Entropy Loss')

pyplot.plot(histories[i].history['loss'], color='blue', label='train')

pyplot.plot(histories[i].history['val\_loss'], color='orange', label='test')

# plot accuracy

pyplot.subplot(2, 1, 2)

pyplot.title('Classification Accuracy')

pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')

pyplot.plot(histories[i].history['val\_accuracy'], color='orange', label='test')

pyplot.show()

# summarize model performance

def summarize\_performance(scores):

# print summary

print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)\*100, std(scores)\*100, len(scores)))

# box and whisker plots of results

pyplot.boxplot(scores)

pyplot.show()

# run the test harness for evaluating a model

def run\_test\_harness():

# load dataset

trainX, trainY, testX, testY = load\_dataset()

# prepare pixel data

trainX, testX = prep\_pixels(trainX, testX)

**#** evaluate model

scores, histories = evaluate\_model(trainX, trainY)

# learning curves

summarize\_diagnostics(histories)

# summarize estimated performance

summarize\_performance(scores)

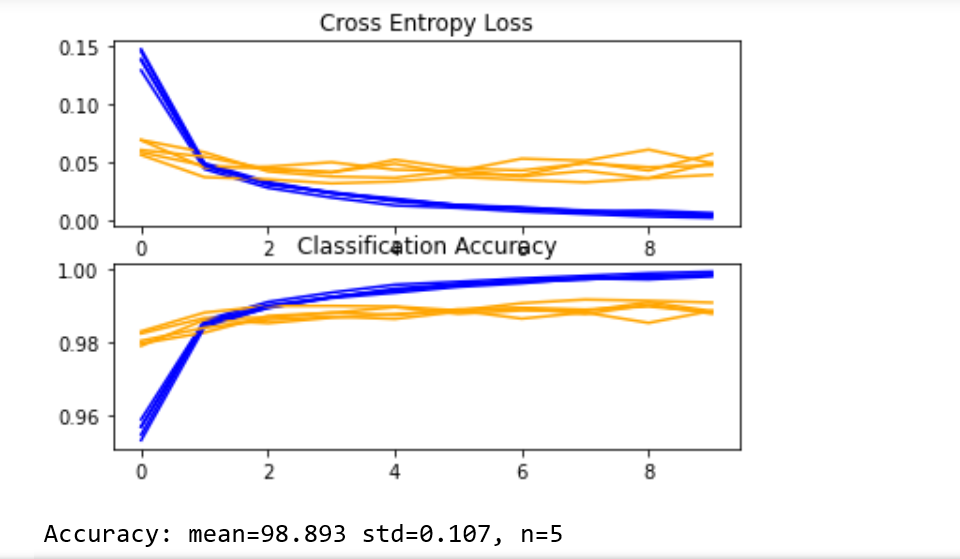
# entry point, run the test harness

run\_test\_harness()

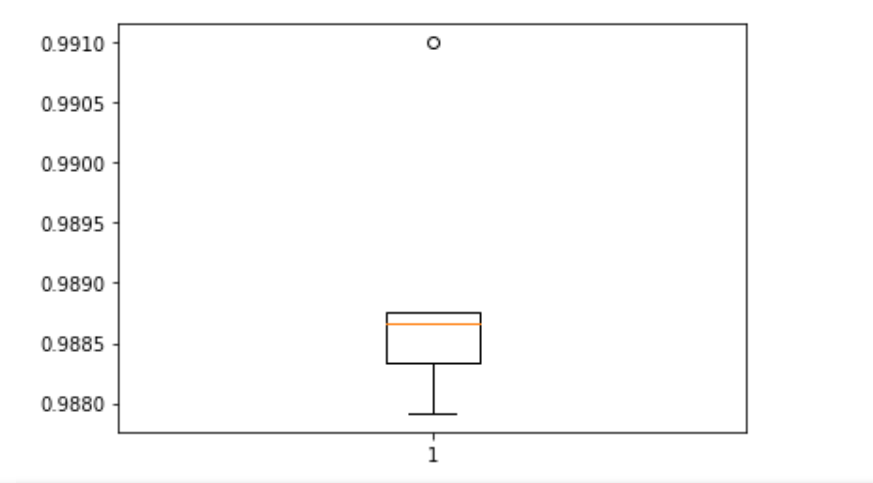
On running this model we get the Output as shown below.



A plot of the learning curves is created, in this case showing that the models still have a good fit on the problem, with no clear signs of overfitting. The plots may even suggest that further training epochs could be helpful.



Next, the estimated performance of the model is presented, showing a small improvement in performance as compared to the baseline from 98.677 to 98.893, with a small drop in the standard deviation as well.

****

**5. Finalize the model and make predictions**

A final model configuration must be chosen and adopted. In this case, we will choose the deeper model as our final model. First, we will finalize our model, but fitting a model on the entire training dataset and saving the model to file for later use. We will then load the model and evaluate its performance on the hold out test dataset to get an idea of how well the chosen model actually performs in practice. Finally, we will use the saved model to make a prediction on a single image.

### Save Final Model

A final model is typically fit on all available data, such as the combination of all train and test dataset. In this tutorial, we are intentionally holding back a test dataset so that we can estimate the performance of the final model, which can be a good idea in practice. As such, we will fit our model on the training dataset only. Once fit, we can save the final model to an H5 file by calling the save() function on the model and pass in the chosen filename. Note, saving and loading a keras model requires that the [h5py library](https://www.h5py.org/) is installed on your workstation.

Fit and Save the model using the save() method as shown below,

def run\_test\_harness():

# load dataset

trainX, trainY, testX, testY = load\_dataset()

# prepare pixel data

trainX, testX = prep\_pixels(trainX, testX)

# define model

model = define\_model()

# fit model

model.fit(trainX, trainY, epochs=10, batch\_size=32, verbose=0)

# save model

model.save('final\_model.h5')

# entry point, run the test harness

run\_test\_harness()

Now the model is saved to final\_model.h5.Which can be used in the future.

Now call the saved model final\_model.h5,

Then evaluate the model by calling the below functions shown,

def run\_test\_harness():

# load dataset

trainX, trainY, testX, testY = load\_dataset()

# prepare pixel data

trainX, testX = prep\_pixels(trainX, testX)

# load model

model = load\_model('final\_model.h5')

# evaluate model on test dataset

\_, acc = model.evaluate(testX, testY, verbose=0)

print('> %.3f' % (acc \* 100.0))

# entry point, run the test harness

run\_test\_harness()

Running the above saved model and evaluates the model on the hold out test dataset.

The classification accuracy for the model on the test dataset is calculated and printed. In this case, we can see that the model achieved an accuracy of 99.030%, or just less than 1%, which is not bad at all.

Ouput of this will be giving the accuracy of the model.



**Make Predictions**

We can use our saved model to make a prediction on new images. The model assumes that new images are grayscale, that they have been aligned so that one image contains one centered handwritten digit, and that the size of the image is square with the size 28×28 pixels.

We will pretend the input image as an entirely new and unseen image, prepared in the required way, and see how we might use our saved model to predict the integer that the image represents (e.g. we expect “2“).

First, we can load the image, force it to be in grayscale format, and force the size to be 28×28 pixels. The loaded image can then be resized to have a single channel and represent a single sample in a dataset. The load\_image() function implements this and will return the loaded image ready for classification.

Importantly, the pixel values are prepared in the same way as the pixel values were prepared for the training dataset when fitting the final model, in this case, normalized. Next, we can load the model as in the previous section and call the predict\_classes() function to predict the digit that the image represents.

The load\_image() and predict\_classes() functions are given below,

# make a prediction for a new image.

from keras.preprocessing.image import load\_img

from keras.preprocessing.image import img\_to\_array

from keras.models import load\_model

# load and prepare the image

def load\_image(filename):

# load the image

img = load\_img(filename, grayscale=True, target\_size=(28, 28))

# convert to array

img = img\_to\_array(img)

# reshape into a single sample with 1 channel

img = img.reshape(1, 28, 28, 1)

# prepare pixel data

img = img.astype('float32')

img = img / 255.0

return img

# load an image and predict the class

def run\_example():

# load the image

img = load\_image(r'C:\Users\Supriya G\Desktop\SUPRIYA B E\ML internship\Project\Hand\_written\_digit\sample\_image\_2.png')

# load model

model = load\_model('final\_model.h5')

# predict the class

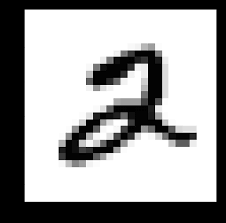
digit = model.predict\_classes(img)

print(digit[0])

# entry point, run the example

run\_example()

So here we are passing the input image as,

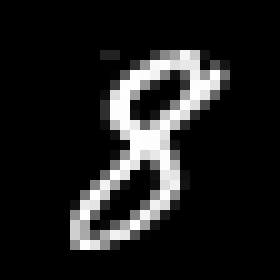


The predicted output will be,



Similarly we can test for some more cases,

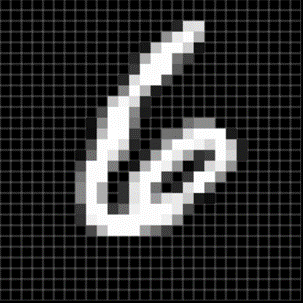
Input image:



Predicted Output:



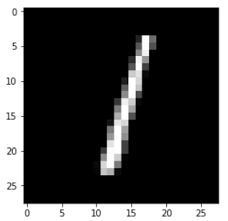
Input image:



Predicted output:



Input image:



Predicted output:



**Conclusion**

This project has practiced different machine learning technique and different models for data training attempting to discover a representation of isolated handwritten digits that allow their effective recognition and to achieve the highest accuracy of predicting handwritten numeral. Thus, this study settled on classifying a given handwritten digit image as the required digit using five different algorithms and consequently testing its accuracy. This study built handwritten recognizers evaluated their performances on MNIST (Mixed National Institute of Standards and Technology) dataset and then improved the training speed and the recognition performance. In addition to develop a system for word based handwriting recognition system and test the handwriting of a given word and detect the writer by selecting which is being recognized for most of the user for a given training sample. This study discusses in detail all advances in the area of handwritten character recognition. The most accurate solution provided in this area directly or indirectly depends upon the quality as well as the nature of the material to be read. The result of this study shows that accuracy is improved as the no of blocked are increased. Apart from that, a 4x4, 7x7 and 14x14 attribute reduction is performed separately to compare and find the optimal number of attributes that best represent the image. However, there is no single classifier that works best on all given problems. A result shows that probabilistic methods suit better for handwriting recognition. By varying the training and testing ratios (from 10% to 90%) we found that the larger training data size improves accuracy, but smaller testing dataset may also favor better accuracy. Preprocessing such as Attribute reduction (784 reduced to 196) reduce runtime and increase accuracy (from 98.9% to 99.030%). The proposed algorithm tries to address both the factors and well in terms of accuracy and time complexity. The overall highest accuracy 99.030% is achieved in the recognition process by Neural Network with the sacrifice of significantly extended runtime. This work is carried out as an initial attempt, and the aim of the paper is to facilitate for recognition of handwritten numeral without using any standard classification techniques. Image processing techniques Median filter, binary, Bweuler, and sharpening improve image quality.

**Bibliography**

* [https://www.an](https://www.analyticsvidhya.com/)[[https://www.kaggle.com](https://www.analyticsvidhya.com/)](https://www.kaggle.com/)
* [alyticsvidhya.com](https://www.analyticsvidhya.com/)
* [https://www.medium.com](https://www.medium.com/)