Handwritten Digit Recognition Using Standard Machine Learning and Deep Learning Algorithms

Shweta Tatiya (001814724), Radhika Gathia (001820710)

CSYE 7245-Big Data Systems and Intelligence Analytics
Under Guidance of Prof. Nicholas Brown
Github link - https://github.com/ShwetaTatiya/Big_Data_Project.git

Abstract- The ability of a device to take handwriting as an input is known as Handwriting recognition. In this paper, we deal with the recognition of handwritten digits by using certain image features that are used by many object recognition algorithms. A part of the extraction method is tested on Mixed National Institute of Standards and Technology database (MNIST) where 60000 images were taken for training and 10000 for testing. The classification models used are Logistic Regression, SVM Regression (Radial, Linear, Polynomial, Sigmoid), K-Nearest Neighbor(k-NN), Decision Tree, Multi-Layer Perceptron(MLP), Recurrent Neural Network(RNN), Deep Belief Network(DBN) and Convolutional Neural Network(CNN) with CNN achieving a highest success rate of 99.40%.

Index Terms- Logistic Regression, k-NN, Decision Tree, SVM, MLP, DBN, CNN, RNN

I. INTRODUCTION

The evolution of various Machine Learning, Deep Learning and Computer Vision algorithms has recently motivated researchers to dig deep into Handwritten Digit Recognition. Optical Character Recognition (OCR) is one of the techniques used for recognition. OCR will read text from scanned document and translate the images into a form that computer can manipulate it. Now-a-days many devices are available that take handwriting as an input such as smartphones, tablets and PDA to a touch screen through a stylus or finger.

Standard machine learning algorithms like K-Nearest Neighbors, SVM, Decision Trees and Logistic Regressions having accuracy of 96% are not enough for real world applications. This introduces the need for deep learning algorithms. In past few years, deep learning has become a trend for Image Processing as well as Handwritten digit and character recognition. Popular machine learning tools such as scikit-learn and Keras, Theano, Tensorflow by Google, TFLearn have made these applications more robust and accurate.

II. MNIST DATASET

The MNIST database of handwritten digits has 60,000 examples in its training set and 10,000 examples in its test set. The test and train images in the MNIST dataset consists an array of 28x28 values representing an image along with their labels. The data is stored in four files:

1. train-images-idx3-ubyte: training set images [1]
2. train-labels-idx1-ubyte: training set labels [1]
3. t10k-images-idx3-ubyte: test set images [1]
4. t10k-labels-idx1-ubyte: test set labels [1]

The ***Training Set Image*** file has the data in the following representation [1]:

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
....			
xxxx	unsigned byte	??	pixel

Pixels are organized row-wise and ranges from 0-255 where 0 means background (white), 255 means foreground (black).

The ***Training Set Label*** file has the data in the following representation [1]:

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
....			
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

The ***Test Set Image*** file has the data in following the representation [1]:

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	10000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
....			
xxxx	unsigned byte	??	pixel

The ***Test Set Label*** file has the data represented as follows [1]:

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  10000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label
```

The values of the labels are 0 to 9.

III. CLASSIFICATION USING MACHINE LEARNING

To measure the accuracy, we are using standard machine learning algorithms such as

1. Logistic Regression
2. Support Vector Machines(SVM)
3. Decision Trees
4. K-Nearest Neighbors(k-NN)

A. *Classification using Logistic Regression*

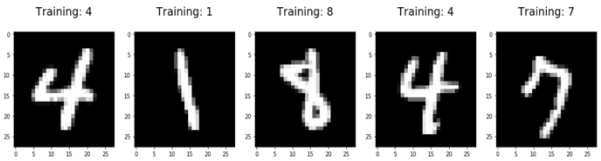

Fig. 1: MNIST Sample Data

Logistic regression is a probabilistic, linear classifier. The image above shows few training digits from the MNIST dataset whose category membership is known (labels 0–9). After using logistic regression to train the model, the model is tested to predict an image label (labels 0–9) given an image.

For MNIST Dataset Logistic Regression works as follows:

1. Loading the MNIST Dataset.
2. Splitting and labelling the data as Train and Test Image and Labels.
3. Randomizing the images and labels.
4. Applying Logistic Regression Classifier algorithm to train the model.
5. Testing the model to calculate the accuracy.

```
clf = LogisticRegressionCV()
trainingdataset=trainingdataset.reshape((trainingdataset.shape[0],trainingdataset.shape[1]*trainingdataset.shape[2]
print(trainingdataset.shape)
print(train_labels.shape)
A=t.time()
print('Training start time',t.time())
clf.fit(trainingdataset,train_labels)
print('Training End Time time',t.time())
print('Time taken',t.time()-A,'seconds')
A=t.time()
print('Prediction start time',t.time())
print('Train_Accuracy',accuracy_score(train_labels,clf.predict(trainingdataset)))
print('Prediction time taken',t.time()-A,'seconds')
```

```
(50000, 784)
(50000,)
Training start time 1522284388.9284632
Training End Time time 1522285082.638916
Time taken 693.7117240428925 seconds
Prediction start time 1522285082.640372
Train_Accuracy 0.92648
Prediction time taken 0.5263290405273438 seconds
```

Fig. 2: Logistic Regression MNIST Digit Recognition Train Accuracy

The above image shows how the model is trained using logistic regression classifier and the training accuracy of the model has come up to 92.64%.

```
test_labels=pickle.load(Data)['test_labels']
print(test_labels.shape)
print(testingdata.shape)
testingdata=testingdata.reshape((testingdata.shape[0],testingdata.shape[1]*testingdata.shape[2]))
A=t.time()
print('Test set accuracy',accuracy_score(test_labels,Clf.predict(testingdata)))
print('Testing time',t.time()-A)
Data.close()
```

```
/Users/shwetatatiya/Desktop/Project_big_data
(900,)
(900, 28, 28)
Test set accuracy 0.92
Testing time 0.00689387321472168
```

Fig. 3: Logistic Regression MNIST Digit Recognition Test Accuracy

The above test image describes the testing accuracy of model which is 92% after the logistic regression is applied on MNIST dataset. However, the accuracy turns out to be very poor.

B. *Classification using Support Vector machines*

Support Vector machine is a discriminative classifier defined by a separating hyperplane. In simple terms, SVM is a supervised machine learning algorithm which when given a labelled training data, outputs a hyperplane that can categorize new data. In two-dimensional space, hyperplane is a straight line dividing the plan into two parts. Kernel is a function that maps the data to a higher dimension where the data is separable. SVM kernel projects the data up by k dimensions such that the data points are now separable in the higher dimensional plane [2].

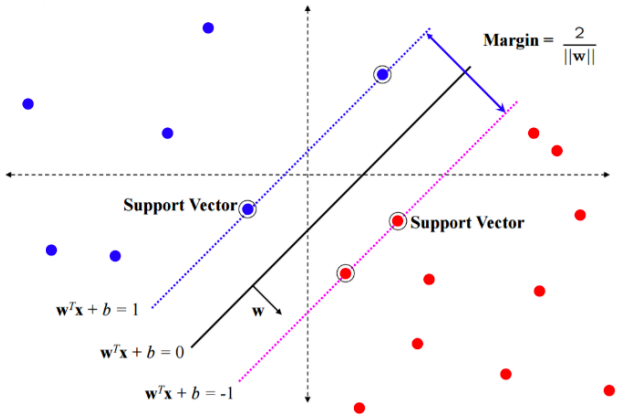

Fig. 4: Support Vector Machine

Working of SVM:

1. Loading the MNIST Dataset
2. Splitting and labelling the data as Train and Test Image and Labels.
3. Performing Principal Component Analysis(PCA) on the dataset.
4. Applying Support Vector Machine algorithm to train the model by fitting the training data to SVM classifier and fine tuning the parameters. SVM algorithm uses the existing labelled data to learn and then classifies the unlabeled data based on the learning.

5. The digit that is recognized using SVM is later matched with the provided Training Label.
6. The Test Image data is used to predict the labels of digits and is compared with the provided test labels to see for the accuracy of the algorithm.
7. The Classification Report and Confusion Matrix is printed which highlights the percentage of accuracy with which each digit has been recognized.

Results of SVM model using different types of kernel:

1. Gaussian Kernel (RBF-Radial Basis Function Kernel):

```
# Fitting the new dimensions.
testset = pca.transform(X_test)
print("Test-set dimensions after PCA")
print(testset.shape)
expected = y_test
predicted = classifier.predict(testset)

print("Results")
print("Classification report for classifier %s:\n%s\n"
      % (classifier, metrics.classification_report(expected, predicted)))
print("Confusion matrix:\n%s" % metrics.confusion_matrix(expected, predicted))
print('\n')
print("Accuracy Score")
accuracy_score(y_test, predicted, normalize = True)
```

Fig. 5.1.a: SVM-rbf kernel MNIST Digit Recognition Test Accuracy

```
Test-set dimensions after PCA
(10000, 90)
Results
Classification report for classifier SVC(C=3, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False):
             precision    recall  f1-score   support

        0.0       0.98      0.99      0.99       980
        1.0       0.99      0.99      0.99      1135
        2.0       0.98      0.98      0.98      1032
        3.0       0.98      0.98      0.98      1010
        4.0       0.98      0.98      0.98       982
        5.0       0.98      0.98      0.98       892
        6.0       0.99      0.98      0.99       958
        7.0       0.98      0.98      0.98      1028
        8.0       0.98      0.98      0.98       974
        9.0       0.98      0.97      0.97      1009

avg / total       0.98      0.98      0.98     10000


Confusion matrix:
[[ 973    0    1    0    0    3    0    1    2    0]
 [   0 1129    3    1    0    1    0    0    1    0]
 [   4    0 1015    0    1    0    1    7    4    0]
 [   0    0    2  993    0    4    0    4    5    2]
 [   0    0    5    0  964    0    3    0    1    9]
 [   2    0    0    6    0  877    3    0    3    1]
 [   6    2    0    0    2    3  942    0    3    0]
 [   1    5    9    2    1    0    0 1004    0    6]
 [   2    0    2    2    3    3    2    2  955    3]
 [   2    3    1    5   11    2    1    6    1  977]]


Accuracy Score

0.9829
```

Fig. 5.1.b: SVM-rbf kernel MNIST Digit Recognition Test Accuracy

The accuracy obtained after implementing Gaussian kernel on MNIST Dataset is 98.29%

2. Linear Kernel:

The accuracy obtained after implementing Linear kernel on MNIST Dataset is 94.42%

```
Test-set dimensions after PCA
(10000, 90)
Results
Classification report for classifier SVC(C=3, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False):
             precision    recall  f1-score   support

        0.0       0.96      0.99      0.97       980
        1.0       0.98      0.99      0.99      1135
        2.0       0.92      0.95      0.94      1032
        3.0       0.91      0.93      0.92      1010
        4.0       0.93      0.96      0.94       982
        5.0       0.92      0.90      0.91       892
        6.0       0.96      0.96      0.96       958
        7.0       0.96      0.94      0.95      1028
        8.0       0.94      0.90      0.92       974
        9.0       0.95      0.91      0.93      1009

avg / total       0.94      0.94      0.94     10000


Confusion matrix:
[[ 967    0    0    2    1    5    4    1    0    0]
 [   0 1124    4    1    0    1    2    1    2    0]
 [   6    1  984    7    9    2    6    4   12    1]
 [   4    1   19  944    0   15    1    9   14    3]
 [   1    0    8    2  938    1    5    2    3   23]
 [   9    4    5   35    7  800    9    0   19    4]
 [  11    1    9    2    2   11  919    2    1    0]
 [   1    6   25    6    8    1    0  964    0   17]
 [   4    3    8   27    6   31    8    6  879    2]
 [   4    6    3   10   33    6    0   19    5  923]]


Accuracy Score

0.9442000000000004
```

Fig. 5.2: SVM-linear kernel MNIST Digit Recognition Test Accuracy

3. Polynomial Kernel:

```
Test-set dimensions after PCA
(10000, 90)
Results
Classification report for classifier SVC(C=3, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='poly',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False):
             precision    recall  f1-score   support

        0.0       0.99      0.99      0.99       980
        1.0       0.99      0.99      0.99      1135
        2.0       0.97      0.97      0.97      1032
        3.0       0.98      0.98      0.98      1010
        4.0       0.98      0.98      0.98       982
        5.0       0.98      0.98      0.98       892
        6.0       0.99      0.98      0.98       958
        7.0       0.98      0.97      0.97      1028
        8.0       0.96      0.98      0.97       974
        9.0       0.96      0.96      0.96      1009

avg / total       0.98      0.98      0.98     10000


Confusion matrix:
[[ 968    0    1    0    0    5    3    1    2    0]
 [   0 1125    3    2    1    1    1    0    1    1]
 [   4    0 1005    0    1    0    2    7   13    0]
 [   0    0    1  988    0    4    0    7    7    3]
 [   0    0    1    0  963    0    3    0    2   13]
 [   2    0    1    7    0  873    2    1    5    1]
 [   3    2    3    0    3    4  938    0    5    0]
 [   0    4   14    0    0    0    0  993    2   15]
 [   1    0    1    4    1    4    0    2  958    3]
 [   1    4    1    6    9    4    1    5    6  972]]


Accuracy Score

0.9782999999999995
```

Fig. 5.3: SVM-poly kernel MNIST Digit Recognition Test Accuracy

The accuracy obtained after implementing Polynomial kernel on MNIST Dataset is 97.82%

4. Sigmoid Kernel:

```
Test-set dimensions after PCA
(10000, 90)
Results
Classification report for classifier SVC(C=3, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='sigmoid',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False):
             precision    recall  f1-score   support

        0.0       0.93      0.98      0.95       980
        1.0       0.96      0.98      0.97      1135
        2.0       0.90      0.89      0.90      1032
        3.0       0.86      0.89      0.87      1010
        4.0       0.90      0.93      0.92       982
        5.0       0.86      0.84      0.85       892
        6.0       0.95      0.94      0.94       958
        7.0       0.93      0.91      0.92      1028
        8.0       0.89      0.85      0.87       974
        9.0       0.91      0.86      0.89      1009

avg / total       0.91      0.91      0.91     10000


Confusion matrix:
[[ 959    0    3    3    0    9    4    1    0    1]
 [   0 1115    1    4    0    2    2    2    9    0]
 [  17    6  923   17   17    5   12    8   25    2]
 [   6    1   25  898    0   40    1   12   23    4]
 [   1    2   10    0  917    0    9    2    2   39]
 [  14    5    8   66    8  747   11    4   25    4]
 [  14    3   17    1    9   14  897    2    1    0]
 [   1   11   22    9   10    0    0  940    2   33]
 [  12   16   13   34    8   47   12    3  825    4]
 [   9    7    1   11   51    7    0   41   10  872]]


Accuracy Score

0.9093
```

Fig. 5.4: SVM-rbf kernel MNIST Digit Recognition Test Accuracy

The accuracy obtained after implementing Sigmoid kernel on MNIST Dataset is 90.93%

C. *Classification using K-Nearest Neighbors (KNN)*

The k-NN algorithm is among the simplest of all machine learning algorithms. In pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression where the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression. In k-NN classification, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. If k = 1, then the object is simply assigned to the class of that single nearest neighbor. A useful technique is to assign weight to the contributions of the neighbors, so that the nearer neighbors contribute more to the average compared to the distant ones. [3]

Working of k-NN:

1. Load the MNIST data.
2. Divide and label the data as Training and Testing Image and labels.
3. Training the k-NN Classifier using KNN algorithm. Provide training data and labels as input to train the classifier. KNN uses the difference of the distances between the actual points and the points provided to classify the digit in the image.
4. The digit that is recognized using KNN is later matched with the provided Training Labels.
5. The Test Image data is used to predict the labels of digits and is compared with the provided test labels to see for the accuracy of the algorithm.
6. The Confusion Matrix is printed that provides the accuracy predictions with which each digit is recognized.

```
In [26]: from sklearn.neighbors import KNeighborsClassifier
         knn = KNeighborsClassifier(n_neighbors=10, leaf_size=75)

In [27]: knn.fit(X_train,y_train)

Out[27]: KNeighborsClassifier(algorithm='auto', leaf_size=75, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=10, p=2,
                    weights='uniform')

In [28]: pred3 = knn.predict(X_test)

         Confusion Matrix

In [29]: from sklearn import metrics
         confusion_matrix = metrics.confusion_matrix(y_test, pred3)
         confusion_matrix

Out[29]: array([[ 972,    1,    1,    0,    0,    2,    3,    1,    0,    0],
                [   0, 1132,    2,    0,    0,    0,    1,    0,    0,    0],
                [  13,   12,  982,    2,    1,    0,    2,   17,    3,    0],
                [   0,    3,    3,  976,    1,   10,    1,    7,    6,    3],
                [   2,   11,    0,    0,  940,    0,    4,    1,    1,   23],
                [   4,    0,    0,   12,    1,  863,    6,    1,    1,    4],
                [   6,    4,    0,    0,    3,    2,  943,    0,    0,    0],
                [   0,   27,    4,    0,    2,    0,    0,  983,    0,   12],
                [   6,    4,    5,   11,    7,    9,    4,    7,  914,    7],
                [   7,    6,    3,    7,   10,    3,    1,   10,    2,  960]])

In [30]: from sklearn.metrics import accuracy_score
         accuracy_score(y_test, pred3)

Out[30]: 0.9665
```

Fig. 6: K-NN MNIST Digit Recognition Test Accuracy

The accuracy obtained after applying k-NN is 96.65%

D. *Classification using Decision Trees*

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label. The paths from root to leaf represent classification rules. Decision trees are commonly used in operations research, specifically in decision analysis, to help identify a strategy most likely to reach a goal but are also a popular tool in machine learning [4].

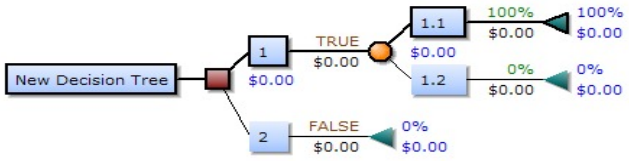

Fig. 7: Decision Tree

Working of Decision Trees:

1. Load the MNIST data.
2. Divide and label the data as Training and Testing Image and labels.
3. Training the model using decision tree algorithm and testing the test images against the trained model to calculate the accuracy. The Confusion Matrix is printed that provides the accuracy predictions with which each digit is recognized.

```
: from sklearn.tree import DecisionTreeClassifier
  dtc=DecisionTreeClassifier(random_state=42)

: dtc.fit(X_train,y_train)

: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                         max_features=None, max_leaf_nodes=None,
                         min_impurity_decrease=0.0, min_impurity_split=None,
                         min_samples_leaf=1, min_samples_split=2,
                         min_weight_fraction_leaf=0.0, presort=False, random_state=42,
                         splitter='best')

: y_pred=dtc.predict(X_test)

: from sklearn.model_selection import cross_val_score
  cross_val_score(dtc, X_train, y_train, cv=3, scoring="accuracy")

: array([0.85557888, 0.85739287, 0.86362954])

  Confusion Matrix

: from sklearn.metrics import confusion_matrix
  from sklearn.model_selection import cross_val_predict
  y_train_pred = cross_val_predict(dtc, X_train, y_train, cv=3)
  confusion_mx=confusion_matrix(y_train, y_train_pred)
  confusion_mx

: array([[5416,    7,   51,   60,   37,  105,  109,   23,   76,   39],
         [   3, 6387,   82,   49,   30,   30,   26,   30,   87,   18],
         [  92,   91, 4931,  185,  121,   59,   88,  138,  172,   81],
         [  49,   71,  179, 5034,   51,  277,   48,  109,  204,  109],
         [  36,   41,   88,   36, 4994,   70,   89,   81,   95,  312],
         [  83,   65,   57,  263,   78, 4416,  145,   34,  158,  122],
         [  76,   38,   86,   45,  101,  161, 5231,   11,  133,   36],
         [  24,   39,  144,   90,   86,   38,   10, 5618,   47,  169],
         [  78,  110,  177,  225,  145,  181,   94,   86, 4572,  183],
         [  42,   27,   69,  121,  278,  125,   26,  187,  141, 4933]])
```

Fig 8.1: Decision Tree MNIST Digit Recognition Confusion Matrix

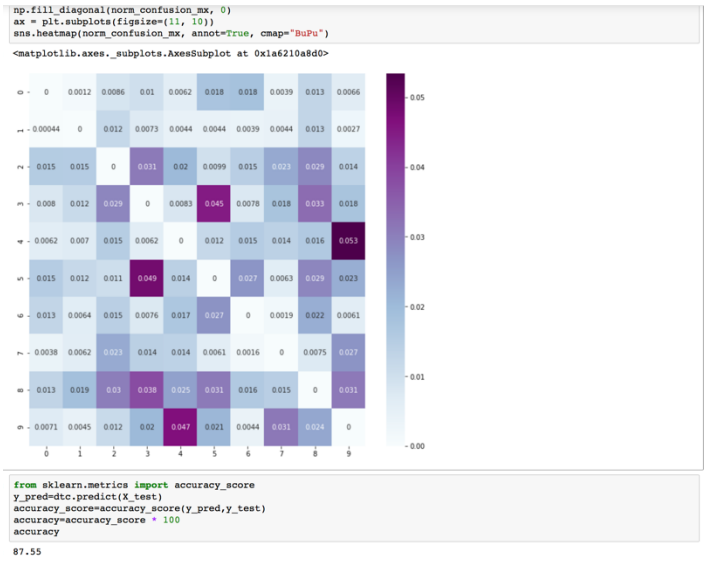

Fig 8.2: Decision Tree MNIST Digit Recognition Test Accuracy

The accuracy achieved with Decision Tree Classifier is 87.55%

IV. MACHINE LEARNING ACCURACY COMPARISON

After implementing the four standard machine learning algorithms, we can draw following conclusions:

1. Logistic regression recognized the digits with 92.64% accuracy.
2. Decision Tree classified the digits correctly with 97.55% accuracy.
3. Using the K Nearest Neighbors, we managed to achieve the accuracy of 96.65%.
4. SVM classifier recognized the digits with 98.29% accuracy using Gaussian kernel.

V. CLASSIFICATION OF DEEP LEARNING ALGORITHMS

For handwritten digit recognition we are implementing four deep learning algorithms. They are Convolutional Neural Networks(CNN), Deep belief Networks(DBN), Recurrent Neural Networks(RNN) and Multi-layer Perceptron(MLP).

A. *Convolutional Neural Network*

A Convolutional Neural Network(CNN) is a class of deep, feed-forward artificial neural networks that is successfully applied to analyze the visual imagery in machine learning.

Convolutional layers apply a convolution operation to the input, passing the result to the next layer. The convolution emulates the response of an individual neuron to visual stimuli. Each convolutional neuron processes data only for its receptive field.

Although fully connected feedforward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10000 weights for each neuron in the second layer. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. For instance, regardless of image size, tiling regions of size 5 x 5, each with the same shared weights, requires only 25 learnable parameters. In this way, it resolves the vanishing or exploding gradients problem in training traditional multi-layer neural networks with many layers by using backpropagation.

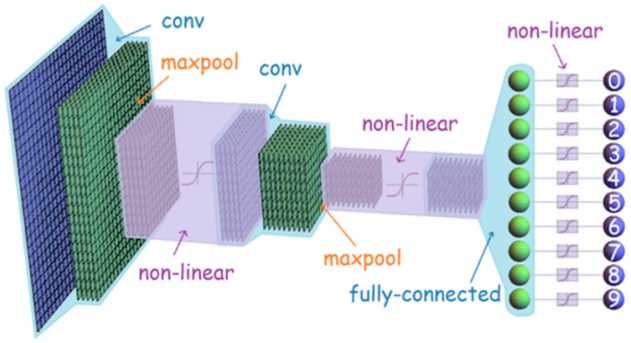

Fig 9: Convolutional Neural Network

Layers of Convolutional Neural Network:

The following algorithm consists of three main steps to build a CNN model.

Step 1: Data Loading:

The first step is to load the MNIST data. This dataset comprises of array with 784 pixels. We then convert the images into 28x28 matrix of pixels.

Step 2: Building Network Architecture:

In this step, we decide the models that are to be used for building a convolutional neural network.

a. Convolutional Layer:

In this layer, we take 32 convolutional filters that go as a window of size 5x5 over all the images of 28x28 matrix size and we try to get the pixels with most intensity value.

b. Pooling Layer:

This layer receives the data from the activation function (Eg: ReLu, Sigmoid) and down samples them in the 3-D tensor. In simple words, it pools all the pixels obtained from previous layers and again forms a new image matrix of a smaller size. These images are again input into the second set of layers and the process continues till we get to a smallest set of pixels from which we can classify the digits.

Step 3: Fully Connected Layer:

This layer connects previous layer to the next layers. It consists of 500 neurons. The layer returns a list of probabilities for each of the 10 class labels. The class label with the largest probability is chosen as the final classification from the network and shown in the output.

```
# Train
history = model.fit(X_train, Y_train, epochs=10, batch_size=batch_size, shuffle=True, verbose=1)

Epoch 1/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.0322 - acc: 0.9906
Epoch 2/10
60000/60000 [==============================] - 86s 1ms/step - loss: 0.0322 - acc: 0.9909
Epoch 3/10
60000/60000 [==============================] - 80s 1ms/step - loss: 0.0295 - acc: 0.9919
Epoch 4/10
60000/60000 [==============================] - 79s 1ms/step - loss: 0.0310 - acc: 0.9920
Epoch 5/10
60000/60000 [==============================] - 87s 1ms/step - loss: 0.0302 - acc: 0.9911
Epoch 6/10
60000/60000 [==============================] - 88s 1ms/step - loss: 0.0303 - acc: 0.9913
Epoch 7/10
60000/60000 [==============================] - 90s 2ms/step - loss: 0.0285 - acc: 0.9919
Epoch 8/10
60000/60000 [==============================] - 89s 1ms/step - loss: 0.0303 - acc: 0.9916
Epoch 9/10
60000/60000 [==============================] - 81s 1ms/step - loss: 0.0326 - acc: 0.9919
Epoch 10/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.0292 - acc: 0.9920

# Evaluate
evaluation = model.evaluate(X_test, Y_test, batch_size=256, verbose=1)
print('Summary: Loss over the test dataset: %.2f, Accuracy: %.4f' % (evaluation[0], evaluation[1]))

10000/10000 [==============================] - 5s 481us/step
Summary: Loss over the test dataset: 0.02, Accuracy: 0.9940
```

Fig 10: CNN MNIST Digit Recognition Test Accuracy

The accuracy achieved by applying convolutional neural network is 99.40%

B. *Multi-Layer Perceptron*

Multi-layer Perceptron (MLP) is a supervised algorithm that learns a function:

$$f(\cdot) : R^m \rightarrow R^o$$

The function will learn by training on a dataset, where *m* is the number of dimensions for input and *o* is the number of dimensions for output. Given a set of features X = x1, x2, ..., xm and a target *y*, it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, call hidden layer [5]

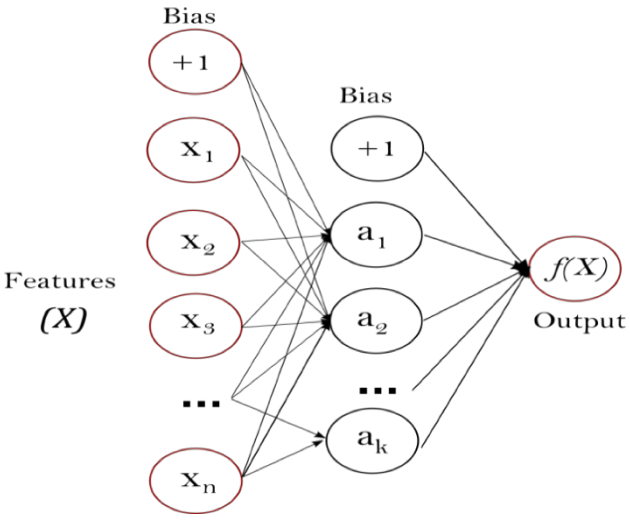

Fig 11: Multi-layer Perceptron

Working of MLP:

The MNIST dataset is loaded and then batch normalization technique is applied. Batch normalization is a technique used for improving the stability of neural networks. Activation functions such as sigmoid and ReLu (Rectified Linear Network) are used in the MLP model Implementation of MLP model with and without batch normalization is carried out on the training set. The test images are then passed through the model and test accuracy is measured. The results obtained showed that the combination of activation functions and presence and absence of batch normalization had different accuracies. The best accuracy is obtained by passing activation function ReLu and without performing any batch normalization technique on the dataset. The accuracy achieved is 97.60%.

ReLU activation function

```
In [17]: ## batch normalization parameter is given as false (without batch normalization)
model_relu = mlp(False, 'relu')
print_layers(model_relu)

bl_noBN = BatchLogger()

from keras.optimizers import Adam
model_relu.compile(loss='categorical_crossentropy', optimizer=Adam(lr=0.001), metrics=["accuracy"])

model_relu.fit(mnist.train.images, mnist.train.labels,
          batch_size=128, epochs=20, verbose=1, callbacks=[bl_noBN],
          validation_data=(mnist.test.images, mnist.test.labels))

('input_3', (None, 784), [])
('dense_13', (None, 128), [100352, 128])
('dense_14', (None, 128), [16384, 128])
('dense_15', (None, 128), [16384, 128])
('dense_16', (None, 128), [16384, 128])
('dense_17', (None, 128), [16384, 128])
('dense_18', (None, 10), [1280, 10])
Train on 55000 samples, validate on 10000 samples
Epoch 1/20
55000/55000 [==============================] - 3s 49us/step - loss: 0.3251 - acc: 0.9013 - val_loss: 0.1603 - val_acc
: 0.9499
Epoch 2/20
55000/55000 [==============================] - 2s 38us/step - loss: 0.1202 - acc: 0.9631 - val_loss: 0.1095 - val_acc
: 0.9649
Epoch 3/20
55000/55000 [==============================] - 2s 38us/step - loss: 0.0842 - acc: 0.9738 - val_loss: 0.1021 - val_acc
: 0.9707
Epoch 4/20
55000/55000 [==============================] - 2s 39us/step - loss: 0.0659 - acc: 0.9796 - val_loss: 0.0966 - val_acc
: 0.9708
Epoch 5/20
55000/55000 [==============================] - 2s 37us/step - loss: 0.0505 - acc: 0.9838 - val_loss: 0.0923 - val_acc
: 0.9741
Epoch 6/20
55000/55000 [==============================] - 2s 35us/step - loss: 0.0427 - acc: 0.9866 - val_loss: 0.0875 - val_acc
: 0.9752
Epoch 7/20
```

Fig 12.1: MLP MNIST Digit Recognition Test Accuracy

```
55000/55000 [==============================] - 2s 35us/step - loss: 0.0176 - acc: 0.9944 - val_loss: 0.0868 - val_acc
: 0.9804
Epoch 15/20
55000/55000 [==============================] - 2s 35us/step - loss: 0.0178 - acc: 0.9941 - val_loss: 0.1007 - val_acc
: 0.9774
Epoch 16/20
55000/55000 [==============================] - 2s 38us/step - loss: 0.0153 - acc: 0.9954 - val_loss: 0.1055 - val_acc
: 0.9750
Epoch 17/20
55000/55000 [==============================] - 2s 38us/step - loss: 0.0187 - acc: 0.9941 - val_loss: 0.0947 - val_acc
: 0.9781
Epoch 18/20
55000/55000 [==============================] - 2s 35us/step - loss: 0.0124 - acc: 0.9961 - val_loss: 0.1212 - val_acc
: 0.9730
Epoch 19/20
55000/55000 [==============================] - 2s 34us/step - loss: 0.0140 - acc: 0.9956 - val_loss: 0.1182 - val_acc
: 0.9751
Epoch 20/20
55000/55000 [==============================] - 2s 35us/step - loss: 0.0133 - acc: 0.9958 - val_loss: 0.1061 - val_acc
: 0.9760
Out[17]: <keras.callbacks.History at 0x1c1e530bd0>

In [18]: evaluation2 = model_relu.evaluate(mnist.test.images, mnist.test.labels, batch_size=256, verbose=1)
print('Summary: Loss over the test dataset: %.2f, Accuracy: %.4f' % (evaluation2[0], evaluation2[1]))

10000/10000 [==============================] - 0s 10us/step
Summary: Loss over the test dataset: 0.11, Accuracy: 0.9760
```

Fig 12.2: MLP MNIST Digit Recognition Test Accuracy

C. *Recurrent Neural Network(RNN)*

A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed graph along a sequence. RNNs can use their internal state (memory) to process sequences of inputs unlike feedforward neural networks. This makes RNN applicable to tasks such as connected handwriting recognition or speech recognition. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that

cannot be unrolled. Both finite impulse and infinite impulse recurrent networks can have additional stored state, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph, if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory and are part of Long short-term memory (LSTM) and gated recurrent units. [6]

Steps used in Handwritten recognition to build a RNN model is as follows:

1. Loading the MNIST dataset under tensorflow environment
2. Setting up various hyperparameters such as learning_rate, training_steps, batch_size, display_step, ninput, timesteps, hidden and num_of_classes.
3. Later, the RNN model is defined with LSTM cells
4. A graph model is then created using Adam optimizer, output function as softmax and loss function as cross entropy. In this step, the labels input on back propagation by default.
5. We now initialize the variables and then predict the training accuracy
6. Once we receive the training accuracy, testing is performed where an accuracy of 98.81% is achieved.

Running the model and predicting the accuracy

```
with tf.Session() as sess:

    sess.run(init)
    for step in range(1, training_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        batch_x = batch_x.reshape((batch_size, timesteps, ninput))
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:

            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + "{:.4f}".format(loss) + ", Training Accuracy= " + "{:.3f

    print("Completed")

    test_length = 10000
    test_data = mnist.test.images[:test_length].reshape((-1, timesteps, ninput))
    test_label = mnist.test.labels[:test_length]
    print("Testing Accuracy:", sess.run(accuracy, feed_dict={X: test_data, Y: test_label}))
```

```
Step 1, Minibatch Loss= 2.4056, Training Accuracy= 0.094
Step 200, Minibatch Loss= 0.2380, Training Accuracy= 0.922
Step 400, Minibatch Loss= 0.2756, Training Accuracy= 0.898
Step 600, Minibatch Loss= 0.1557, Training Accuracy= 0.961
Step 800, Minibatch Loss= 0.0710, Training Accuracy= 0.984
Step 1000, Minibatch Loss= 0.0556, Training Accuracy= 0.977
Step 1200, Minibatch Loss= 0.0648, Training Accuracy= 0.984
Step 1400, Minibatch Loss= 0.0358, Training Accuracy= 0.992
Step 1600, Minibatch Loss= 0.0622, Training Accuracy= 0.969
Step 1800, Minibatch Loss= 0.0546, Training Accuracy= 0.977
Step 2000, Minibatch Loss= 0.0514, Training Accuracy= 0.984
Step 2200, Minibatch Loss= 0.0770, Training Accuracy= 0.969
Step 2400, Minibatch Loss= 0.0394, Training Accuracy= 0.984
Step 2600, Minibatch Loss= 0.0485, Training Accuracy= 0.984
Step 2800, Minibatch Loss= 0.0481, Training Accuracy= 0.992
Step 3000, Minibatch Loss= 0.0268, Training Accuracy= 0.984
```

Fig 13.1: RNN MNIST Digit Recognition Test Accuracy

```
Step 2400, Minibatch Loss= 0.0394, Training Accuracy= 0.984
Step 2600, Minibatch Loss= 0.0485, Training Accuracy= 0.984
Step 2800, Minibatch Loss= 0.0481, Training Accuracy= 0.992
Step 3000, Minibatch Loss= 0.0268, Training Accuracy= 0.984
Step 3200, Minibatch Loss= 0.0080, Training Accuracy= 1.000
Step 3400, Minibatch Loss= 0.0256, Training Accuracy= 0.992
Step 3600, Minibatch Loss= 0.0211, Training Accuracy= 1.000
Step 3800, Minibatch Loss= 0.0840, Training Accuracy= 0.984
Step 4000, Minibatch Loss= 0.0197, Training Accuracy= 0.992
Step 4200, Minibatch Loss= 0.0968, Training Accuracy= 0.992
Step 4400, Minibatch Loss= 0.0126, Training Accuracy= 0.992
Step 4600, Minibatch Loss= 0.0031, Training Accuracy= 1.000
Step 4800, Minibatch Loss= 0.0049, Training Accuracy= 1.000
Step 5000, Minibatch Loss= 0.0082, Training Accuracy= 1.000
Step 5200, Minibatch Loss= 0.0025, Training Accuracy= 1.000
Step 5400, Minibatch Loss= 0.0114, Training Accuracy= 1.000
Step 5600, Minibatch Loss= 0.0073, Training Accuracy= 1.000
Step 5800, Minibatch Loss= 0.0131, Training Accuracy= 0.992
Step 6000, Minibatch Loss= 0.0031, Training Accuracy= 1.000
Step 6200, Minibatch Loss= 0.0177, Training Accuracy= 0.984
Step 6400, Minibatch Loss= 0.0625, Training Accuracy= 0.992
Step 6600, Minibatch Loss= 0.0331, Training Accuracy= 0.992
Step 6800, Minibatch Loss= 0.0033, Training Accuracy= 1.000
Step 7000, Minibatch Loss= 0.0064, Training Accuracy= 1.000
Step 7200, Minibatch Loss= 0.0003, Training Accuracy= 1.000
Step 7400, Minibatch Loss= 0.0018, Training Accuracy= 1.000
Step 7600, Minibatch Loss= 0.0196, Training Accuracy= 0.992
Step 7800, Minibatch Loss= 0.0005, Training Accuracy= 1.000
Step 8000, Minibatch Loss= 0.0135, Training Accuracy= 0.992
Step 8200, Minibatch Loss= 0.0022, Training Accuracy= 1.000
Step 8400, Minibatch Loss= 0.0268, Training Accuracy= 0.992
Step 8600, Minibatch Loss= 0.0036, Training Accuracy= 1.000
Step 8800, Minibatch Loss= 0.0018, Training Accuracy= 1.000
Step 9000, Minibatch Loss= 0.0020, Training Accuracy= 1.000
Step 9200, Minibatch Loss= 0.0129, Training Accuracy= 0.992
Step 9400, Minibatch Loss= 0.0005, Training Accuracy= 1.000
Step 9600, Minibatch Loss= 0.0051, Training Accuracy= 1.000
Step 9800, Minibatch Loss= 0.0028, Training Accuracy= 1.000
Step 10000, Minibatch Loss= 0.0054, Training Accuracy= 1.000
Completed
('Testing Accuracy:', 0.9881)
```

Fig 13.2: RNN MNIST Digit Recognition Test Accuracy

D. *Deep Belief Network(DBN)*

DBNs can be viewed as a composition of simple, unsupervised networks such as restricted Boltzmann machines(RBMs) or autoencoders, where each sub-network's hidden layer serves as the visible layer for the next. An RBM is an undirected, generative energy-based model with a "visible" input layer and a hidden layer and connections between but not within the layers. This composition leads to a fast, layer-by-layer unsupervised training procedure, where contrastive divergence is applied to each sub-network in turn, starting from the "lowest" pair of layers (the lowest visible layer is a training set) [7]

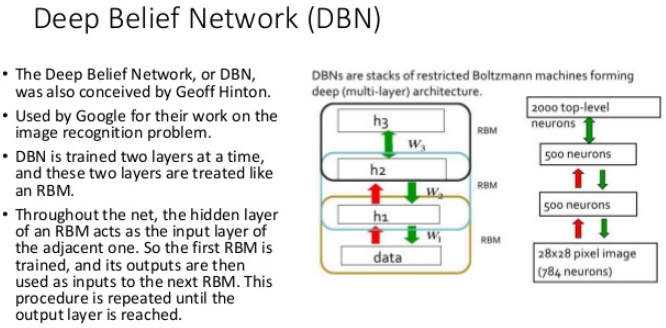

Fig 14: Deep Belief Network

Given a set of images of handwritten digits, we build a DBN classification model that maps these images into its corresponding numerical digit. After loading MNIST dataset, it is split into training and testing images and labels. DBN function parameters are defined by setting the hyperparameters.

The parameters tuned are as follows:

1. Learning rate-Learning rate is the size of the adjustments made to the weights with each iteration. A high learning rate makes a net traverse the errors cape quickly, but also makes it prone to overshoot the point of minimum error. A low learning rate is more likely to find the minimum, but it will do so very slowly, because it is taking small steps in adjusting the weights.
2. Learn_rate_decays - The number with which the learn_rate is multiplied after each epoch of fine-tuning.
3. output_act_funct (Output activation function) - In this case softmax function is used as activation function. Softmax activation function in the output layer of a deep neural net to represent a categorical distribution over class labels and obtaining the probabilities of each input element belonging to a label.
4. The next step that follows parameter tuning is accuracy prediction. [8]

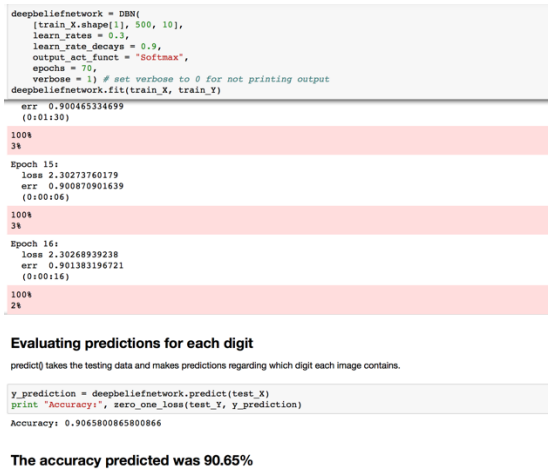

Fig 15: DBN MNIST Digit Recognition Test Accuracy

The model achieved the accuracy of 90.65%

VI. CONCLUSION OF DEEP LEARNING ALGORITHMS

Following accuracies are achieved after implementing various deep learning algorithms. They are as follows:

1. Deep Belief Network achieving an accuracy of 93.44%
2. Multi-Layer Perceptron recognized the digits with 97.60% accuracy
3. On the other hand, RNN received an accuracy of 98.81%
4. Finally, coming to CNN who successfully recognized the digits with 99.40% accuracy.

VII. RESULTS

The below snippet shows different cases of CNN model that is built using different hyperparameters and network architecture.

Sr No	ACTIVATION FUNCTION	COST FUNCTION (loss)	EPOCHS	GRADIENT ESTIMATION (optimizer)	NETWORK ARCHITECTURE	NETWORK INITIALIZATION (kernel_initializer)	Accuracy
1	relu	categorical_crossentropy	10	RMSprop	3 layers, kernelsize - 3 with 2 FC	he_normal	99.40%
2	relu	poisson	10	adagrad	3 layers, kernelsize - 3 with 2 FC	uniform	99.38%
3	linear	kullback_leibler_divergence	20	adamax	4 layers, kernelsize - 3 with 2 FC	uniform	99.34%
4	relu	categorical_crossentropy	50	rmsprop	3 layers, kernelsize - 3 with 2 FC	he_normal	99.01%

Fig 16: CNN Case Comparison

Similarly different cases are implemented for various other standard machine learning and deep learning algorithms to check which algorithm gives highest accuracy.The table show below describes the accuracy achieved for each and every case.

Algorithm + Cases	Test Accuracy
1. Logistic Regression	92.64%
2. SVM Regression – Gaussian Kernel	98.29%
3. SVM Regression – Linear Kernel	94.42%
4. SVM Regression – Polynomial Kernel	97.82%
5. SVM Regression – Sigmoid Kernel	90.93%
6. K-NN (neighbor=5)	90.38%
7. K-NN (neighbor=7)	95.83%
8. K-NN (neighbor=10)	96.65%
9. Decision Tree (only 1 parameter)	87.55%
10. Decision Tree (many parameters)	84.14%
11. Deep Belief Network	90.65%
12. Recurrent Neural Network	98.81%
13. MLP (sigmoid & no BN)	96.78%
14. MLP (sigmoid & BN)	97.52%
15. MLP (ReLu & no BN)	97.60%
16. MLP (ReLu & BN)	95.03%
17. Convolutional Neural Network	99.40%

Fig 17: Accuracy Comparison

VIII. CONCLUSION

Looking at the Accuracy Comparison table (Fig 17) we can finally conclude that from all the standard machine learning algorithms and deep learning algorithms, CNN turns out to be the best fit for the MNIST Handwritten Digit Recognition Dataset by giving an accuracy of 99.40%.

IX. DISCUSSION

We can conclude that as the number of epochs increase, the accuracy decreases after implementing various cases on the Convolutional model that received highest accuracy. However, the performance of the model not only depends on the number of epochs, but it also depends on various other parameters such as choice of optimizers, cost functions, activation functions as well as network architecture.

REFERENCES

[1] http://yann.lecun.com/exdb/mnist/
[2] https://codingmachinelearning.wordpress.com/2016/07/25/support-vector-machines-kernel-explained/https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
[3] https://en.wikipedia.org/wiki/Decision_tree
[4] https://arxiv.org/pdf/1702.00723.pdf
[5] https://en.wikipedia.org/wiki/Recurrent_neural_network
[6] https://en.wikipedia.org/wiki/Deep_belief_network
[7] https://deeplearning4j.org/mnist-for-beginner