OPTICAL DIGIT RECOGNITION USING NEURAL NETWORKS

Samarth Mishra(130260018)
Katuri Sai Kiran(130050051)
Ravichandra Addanki (130050061)
Project Description

- Write a program which recognizes the numeric digit present in the images with a single hand-written digit on them using neural networks.
- Implement the training phase for the neural network to learn and a test phase for the same to predict on test samples.
- Calculate the percentage success rate on a collection of a big test sample.
Implementation Platform

- We chose to use Python despite its not so fast nature due to availability of wide range of easy-to-use libraries for implementing and trying out various algorithms in neural-net like PyBrain.
- We are planning to use PyBrain module’s neural net implementation for both training and testing phases of our program.
- The official documentation and relevant libraries of PyBrain for download could be found here => http://pybrain.org/
Algorithms Used

- Backward propagation algorithm to train the neural network using the training data
- Feed forward algorithm to calculate the final output from the given inputs and the trained neural network
- Various image processing algorithms to extract “features” out of the raw image data
- The first 2 have been implemented and made available in the PyBrain libraries.
We are planning to use the MNIST handwritten dataset for training and testing. Details about it can be found here.

It contains over 60,000 handwritten training samples, and over 10,000 test samples.

This database is open-sourced and created by a collaborative effort from Courant University, Google Labs, Microsoft Research Redmond.
FILE FORMAT FOR MNIST DATASETS

- The files in the MNIST data set are in the .idx format which we parse to output an image of the handwritten digit (we use the pygame library for this).
- For each of the phases, there are two files, one containing the pixel data, and the other containing the labels for the images. Both these files are in the idx format described above.
INPUT DATA FORMAT

- Each image is a grayscale image of size 28X28 pixels. Each pixel is of 1 byte. 0xFF means foreground (black) and 0x00 means background (white).
- There is also a label mapped with each image in a different file which can be used to get the digit that is associated with that image. This label is of 1 byte size and is between 0-9.
- We found further details about the IDX file format here => [http://yann.lecun.com/exdb/mnist/](http://yann.lecun.com/exdb/mnist/)
Examples of MNIST data
Libraries Used

- We use the PyBrain library for building and training the neural network for the OCR.
  - Library for loading and using the datasets – pybrain.datasets
  - Libraries for training – pybrain.supervised.trainers
- Graphic libraries for visualization – pygame
- Image processing libraries – Image, numpy
- Network Writer for storing the neural nets across different runs.
WORK & IMPLEMENTATION DESCRIPTION

- A parser to read the mnist dataset into an array format. This can be found in our file ‘convert.py’. It reads the 60,000 examples given in the mnist dataset file and populates the array with grayscale images and their corresponding labels.
- neural Network training : The code for this can be found in the file main.py. We either create a new neural network or load from an already trained neural net. The option for this is specified using a command line argument.
- Testing visualizer : A visualizer has been built to load a random input and test on it. This input is loaded from the test part of the mnist dataset. We also display the image visually.
Training Phase

- Since, our neural network contains 784 inputs and 60,000 training data samples, it took a lot of time for training of our neural net. Each iteration took about 100 seconds. Since, we trained it for about 5 hours.
- Since, 5 hours is a lot of time and we might lose the neural net at any case, hence after each iteration we are storing the entire neural net in an xml file so that if at any stage if we lose any data of neural net we can always get the backup from this xml file.
- Since, we also trained for a lot of cases like varying hidden layers, varying features etc..(graphs about these later), we did most of the training parallelly on 5 systems(3 PCs and 2 systems by using ssh).
Attempts made to improve accuracy

- Our first attempt was with a reduced feature vector, which had 10x10 pixels which was produced by an operation similar to a zoom out on the data-set images.
- We used a single output node which was supposed to output the decimal value of the predicted digit directly in the ideal case.
- Also we used the number of hidden nodes of 20 in this case and further we used only 2,000 training samples out of all 60,000 samples.
- Even though the distribution of 10 digits is uniform among these 2000 samples, after training all 60,000 samples, we got far better results.
- This, as it turned out, wasn’t very good at predicting the output, and the maximum accuracy that we could achieve was 40%.
Attempts made to improve accuracy

- We then tried to train for the full feature set of 28x28 pixels, trading-off training speed for more accuracy.
- We also made 10 output nodes, the $i^{th}$ of which will output 1 in an ideal case, and the others will output 0.
- The final predicted output of our neural net is thus, taken to be the output node index which outputs the highest value.
- This increased the accuracy a bit. We then used all 60,000 samples for training which increased accuracy a lot.
- We manipulated the number of nodes in the hidden layer and decided it was ideal to fix it around 30 considering the number of training samples and the time we have to train it. The graphs portion explains about this.
We train the neural net on the 60,000 examples one-by-one. We output the error after this training.

Then we sample around 50 test inputs for each of the digits (total of 500 test inputs) and we measure the output as predicted. We count the number of correct predictions and output this number.

This way we were able to measure the accuracy achieved after each training.

Also we write the neural network to stable storage after every training step. This is done to prevent the loss of training if the program suddenly stops for some reason.

We try to interpret the statistics obtained in the next slides.
Accuracy vs Number of Epochs Trained For

The data is the accuracy which we get by testing the neural net on 500 test samples (50 for each digit) after a certain number of epochs of training which we mention on the X-axis. As can be seen, the accuracy improves with training and saturates at around 85%.
**Training Error vs Number of Epochs Trained For**

The data is the training error we get at the end of a certain number of epochs of training which we mention on the X-axis. The training error reduces in the expected manner.
Training Error vs Number of Epochs Trained For

The data is the test error we get at the end of a certain number of epochs of training for the test conducted on 500 test samples, 50 for each digit. As can be seen, the test error reduces and the neural net becomes a better predictor with training.
Accuracy after 20 epochs of training vs number of hidden layer nodes

The larger the number of hidden layer nodes, the slower is the neural network to learn, since more number of weights have to be adjusted from the initial random values assigned to them. Though there is a difference in accuracy, it is not very huge.


The data for the adjoining graph is collected by creating 3 additional neural networks and training them only for the desired number of sample points and then performing enough rounds to make sure that the training has saturated and measuring the accuracy after that.
**Visual Interface to test Neural Net**

- In-order to make things easy for demo and grading, we made a simple visual interface where we can visually see the image on screen and compare it with predicted value.
- The interface first reads the neural net from xml file which is stored using networkwriter of python.
- It then picks a random test input from the present 10,000 test inputs of MNIST data.
- It then queries neural net to predict it by giving its grey scale data.
- After printing the predicted result on terminal it also displays the image so that we can visually see the digit on screen and compare it with predicted value.
Where neural net went “wrong”

Expected value : 8

Predicted value : 9

Reason : Quite clearly the above image resembles 9. The bottom loop is quite small as compared to the upper loop, and also the body is tilted to the left which 9 usually is.
**Where neural net went “wrong”**

Expected value : 9

Predicted value : 1

Reason : Quite clearly the above image resembles 9. However on removing the top loop, 1 can be obtained. Due to this close similarity, the hidden neurons which fire when the features of 1 are detected also fired for this one, and this possibly led to the prediction of 1 instead of 9.
**Where neural net went “wrong”**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>g</td>
</tr>
<tr>
<td>8(9)</td>
<td>4(9)</td>
<td>4(9)</td>
<td>1(9)</td>
<td>8(9)</td>
<td></td>
</tr>
<tr>
<td>7(2)</td>
<td>1(7)</td>
<td>4(7)</td>
<td>3(5)</td>
<td>2(6)</td>
<td></td>
</tr>
</tbody>
</table>
Where neural net went “wrong”

Expected value : 5

Predicted value : 7

Reason : The 5 written in the above format and 7 written in its most common format have the following similarities : The line at the top and an almost straight line from top to the bottom. The neurons that trigger for these features in the case of 7 also triggered in this case for 5 and hence, there is a mis-match.
**Unexplainable Predictions**

Expected value : 5

Predicted value : 3

Comments : As one can see that this is the clearest form of 5 that a human can write. But, we were not able to explain why neural net is predicting it to be 3 as we can see that it is not longer closer to 3 except the bottom part. The only comment that we can make at this stage is it might be because of the bottom part which is similar to 3.
**Deliverables**

- A Python program that can recognize handwritten characters which are passed to it in an image format of that of the MNIST dataset.
- Random test generating interface which chooses a random handwritten digit from the MNIST test data set, and outputs its image and the prediction made by the neural net.
- Statistics of neural net learning:
  - Number of nodes in hidden layer vs accuracy (trained for given number of iterations).
  - Learning against Reduced feature vectors vs complete learning.
- After all the training phase we reached upto an accuracy of 88% on the test data of MNIST. Here accuracy means that out of 500 test samples that we query for, it is predicting correctly for 440 test samples.
What we learnt

- Learning in Neural networks is a slow process and one needs a lot of time to get to reasonable accuracies.
- How to extract features from different types of problems. For example, in this project we tried different types of features like resizing image to 10*10 pixels and finally we found out that 28*28 pixels are far better. We also tried to convert the greyscale into black-white scale which only contains 0 or 1 but this also didn’t improve any accuracy. Thus choosing the correct features is very important.
- We learnt that there is a trade off between no. of hidden layers and time to achieve desired accuracy. As we increase the no. of hidden layers, the more time it takes to reach desired accuracy. This along with the no. of varieties of features needed to be learnt dictates the no. of nodes in the hidden layers to be used in neural net.
Future Scope of Work

- The project can further be expanded to have the following feature: Convert normal pictures of hand-written numbers (sequence of digits), break them into single digits and convert them into MNIST data-set like inputs. Our program would then output the hand-written number.
- We could also explore into other kinds of features for images, which are more characteristic of a digit, and which could hence, allow us to increase the prediction accuracy.
- We know that python is much slower than C++. Hence, we could port this to C++ which makes the training phase to run much faster as compared to python’s.
REFERENCES

- http://yann.lecun.com/exdb/mnist/
- http://pybrain.org/