6th September 2021

MNIST Handwriting ANN Report

Introduction

One of the main applications of artificial intelligence (AI) is the recognition of faces, identification of images, and classification of handwriting. There is a common theme between all of these applications: the ability of an algorithm to identify edges, both straight and curved. AI is a powerful tool, as it allows one to statistically associate specific features with known information.

The USA's National Institute for Standards and Technology's modified handwriting dataset (MNIST) is made up of tens of thousands of handwritten single digit numbers (i.e., 0 - 9). Each image has a resolution of 28x28 pixels and is greyscale so each pixel has a range of 0 to 255.

There are 10,000 instances in the testing set. In this project, an Artificial Neural Network (ANN) will attempt to classify all 10,000 correctly. The training set was written by ~250 different individuals.

Dataset

The MNIST training and testing datasets have 60,000 and 10,000 instances, respectively. Each row on the dataset represents one 784-pixel image, with each cell representing one pixel of the image. The first number on each line is the label for that image.

A typical row for a single image in the data set would look like this:

LABEL	**PIXEL 1**	**PIXEL 2**	**PIXEL 4**	**...**	**PIXEL 784**

Pre-Processing the Dataset

Every image is grayscale, so each pixel holds a value between 0 and 255. In this project, these were mapped to a number in the range [0.01, 1]. This was done by multiplying each pixel by $\frac{0.99}{255}$, and then adding 0.01 to the product. Adding 0.01 to the product is necessary to remove zero values. For example, if the pixel is 0, then $0 * \frac{0.99}{255} = 0$. This would defeat the purpose of this process because zero-values can interfere with the computations of the ANN (for example, it might cause the weighted inputs of certain nodes to be 0):

```
fraction = 0.99/255;
TrainImgs = (np.asfarray(TrainData[:, 1:]) * fraction)
+ 0.01;
TestImgs = (np.asfarray(TestData[:, 1:]) * fraction) +
0.01;

TrainLabels = np.asfarray(TrainData[:, :1]);
TestLabels = np.asfarray(TestData[:, :1]);
```

One-Hot Representation:

After converting all images to this format, each label was converted into one-hot representation. One-Hot representation is when there is only a single bit (a bit is a single binary digit), which is “1”, all other bits are zero. Here, each label was converted to one-hot representation (with 1s and 0s), and then each one and zero was converted into 0.01 and 0.99 respectively, as this works better with the images.

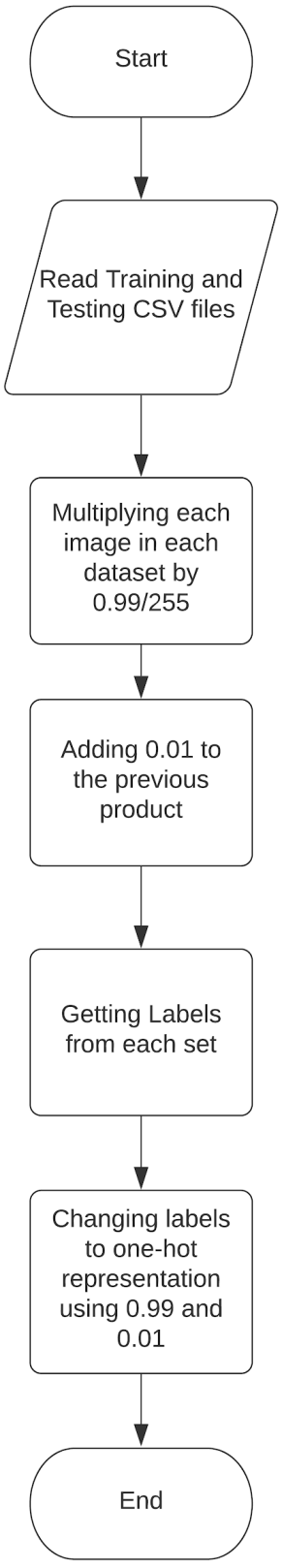
```
## One-Hot Rep
lr = np.arange(NoLabels);

# Changing labels into One-Hot form
TrainOneHot = (lr==TrainLabels).astype(np.float);
TestOneHot = (lr==TestLabels).astype(np.float);

# Using 0.01 & 0.99 instead of 1 and 0:
TrainOneHot[TrainOneHot == 0], TrainOneHot[TrainOneHot
== 1] = 0.01, 0.99;
TestOneHot[TestOneHot == 0], TestOneHot[TestOneHot ==
1] = 0.01, 0.99;
```

This was the last process that needed to be completed before the neural network could be programmed.

Pre-Processing Flow Chart

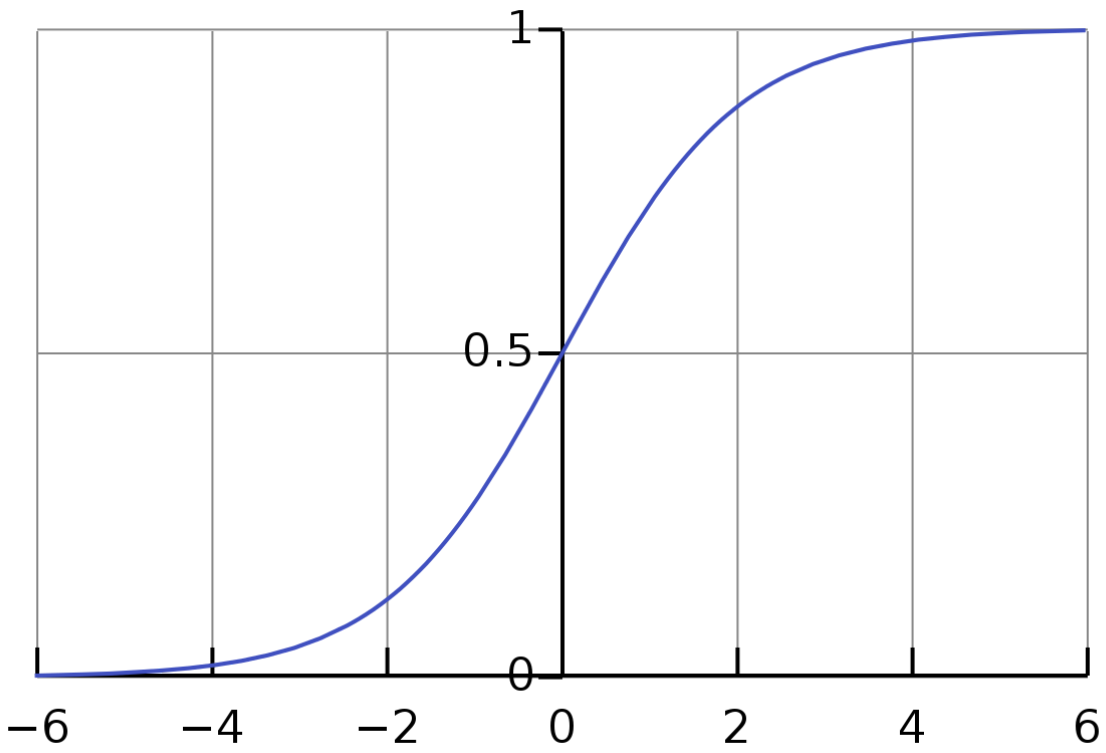


The ANN Algorithm

First, the Sigmoid Function was programmed, which will act as an "activation function" for a node. The Sigmoid Function is often used in ANNs when the output values are probabilities. Both the Sigmoid Function and probabilities only exist between $0 \leq S(x) < 1$, so Sigmoid is the correct choice of activation function.

$S(x) = \frac{1}{1+e^{-x}}$ (Sigmoid Function)

Graph of the Sigmoid Function:




```
@np.vectorize
def sigmoid(x):
    return 1 / (1 + np.e ** -x);
activationFunc = sigmoid;
```

The operation *@np.vectorise,* that is used above, is a NumPy method that tells the program to recursively repeat the function for every element in a given array each time that the function is called.

After that, the weight matrices were created in a method of neuralNetwork:

```
    def weightMatrices(self):
        """
        A method to initialize the weight
        matrices of the neural network
        """
        rad = 1 / np.sqrt(self.InNodes);
        x = truncatedNorm(mean = 0, sd = 1, low = -rad,
upp = rad);
        self.wih = x.rvs((self.HiddeNodes,
self.InNodes));
        rad = 1 / np.sqrt(self.HiddenNodes);
        x = truncatedNorm(mean=0, sd=1, low=-rad,
upp=rad);
        self.who = x.rvs((self.OutNodes,
self.HiddenNodes));
```

"rad = 1 / np.sqrt(self.InNodes);" calculates the bounds of the distribution (-rad points to the lower bound, +rad points to the upper bound).

The upper and lower bounds of the distribution were removed as they are very small and it will not affect the outcome of the ANN.

Using x.rvs, random numbers can be obtained using the number of hidden nodes, self.HiddenNodes, and the number of Input Nodes, self.InNodes, and assigning the random numbers to self.wih (the weight values of the synapses between input and the first layer of hidden nodes). With self.who, the same thing is done, except that the random numbers are being assigned to the weights of the synapses between the final hidden layer and the output layer.

Next, a method was written in order to train the ANN:
The method begins by converting the parameters of the function, InputVector (array) and TargetVector (array), into NumPy arrays. InputVector is the array of pixels for that image (with each pixel acting as an input node for every image).

Then, each input vector is multiplied by self.wih to produce a first set of output vectors. These output vectors are then "activated" using the Sigmoid function, creating the OutputHidden variable, which is the value of the last layer of hidden nodes.

These OutputHidden values are then multiplied with self.who weights and given as inputs to the output layers, where they are then activated again to give a final output.

The bottom 12 lines of the code is where the actual "training" occurs. The values for the output of the ANN is subtracted from the values of the target vectors to give "OutputErrs". OutputErrs is then multiplied with the output of the ANN. This product is multiplied to the learning rate, the product of which is added to the weights between the last hidden layer and the output layer.

Next, the weights between the input layer and the first hidden layer are adjusted. This is done by multiplying self.who with OutputErrs to receive a value, HiddenErrs. HiddenErrs is then multiplied by the value of OutputHidden and 1 minus OutputHidden. The result of this is assigned to the variable *tmp*:

$$HiddenErrs = self.who * OutputErrs$$
$$tmp = HiddenErrs * OutputHidden * (1 - OutputHidden)$$

After that, self.wih is found by $LearningRate * (tmp * OutputHidden)$. Now, the weights are adjusted giving slightly more accurate values for the next image.

(Training method code is on the next page.)

```
    def train(self, InputVector, TargetVector):
        """
        InputVector and TargetVector can
        be tuple, list or ndarray
        """

        InputVector = np.array(InputVector, ndmin=2).T
        TargetVector = np.array(TargetVector,
ndmin=2).T

        OutputV1 = np.dot(self.wih, InputVector);
        OutputHidden = activationFunc(OutputV1);

        OutputV2 = np.dot(self.who, OutputHidden);
        OutputNet = activationFunc(OutputV2);

        OutputErrs = TargetVector - OutputNet;
```

```
        # update the weights:
        tmp = OutputErrs * OutputNet \
              * (1.0 - OutputNet);
        tmp = self.LearningRate  * np.dot(tmp,
OutputHidden.T);
        self.who += tmp;


        # calculate hidden errors:
        HiddenErrs = np.dot(self.who.T, OutputErrs);
        # update the weights:
        tmp = HiddenErrs * OutputHidden * \
              (1.0 - OutputHidden);
        self.wih += self.LearningRate \
                          * np.dot(tmp, InputVector.T);
```

Results

After running the ANN, with the run() method, I received the following results:
Training set accuracy: 94.96%
Testing set accuracy: 94.86%

For each label there was a precision of (2dp):
1: 98.06%
2: 98.33%
3: 93.70%
4: 96.24%
5: 93.18%
6: 92.15%
7: 96.35%
8: 89.79%
9: 94.35%
10: 95.94%
(N.B. in the program, the labels were given numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.)

This is the ratio/percentage of images that were actually a certain label to ones that were predicted to be the correct label.

Training Set Confusion Matrix

		PREDICTED VALUES										
		1	2	3	4	5	6	7	8	9	10	TOTAL ACTUALS/LABEL
ACTUAL VALUES	1	5785	1	36	12	11	28	28	13	15	15	5944
	2	0	6609	30	23	10	25	17	49	73	9	6845
	3	4	30	5622	72	21	14	7	58	14	6	5848
	4	8	30	68	5789	2	79	0	20	91	61	6148
	5	6	13	39	5	5385	23	10	38	19	65	5603
	6	6	4	2	53	1	5042	40	3	17	11	5179
	7	38	4	32	19	55	70	5762	9	26	4	6019
	8	1	7	33	29	3	2	0	5757	1	24	5857
	9	59	25	84	60	10	77	53	18	5504	34	5924
	10	16	19	12	69	344	61	1	300	91	5720	6633
	TOTAL PREDICTED/LABEL	5923	6742	5958	6131	5842	5421	5918	6265	5851	5949	60000

Testing Set Confusion Matrix

		PREDICTED VALUES										
		1	2	3	4	5	6	7	8	9	10	TOTAL ACTUALS/LABEL
ACTUAL VALUES	1	961	0	9	2	2	6	1	3	4	2	990
	2	0	1116	2	0	0	2	3	15	6	6	1150
	3	0	3	967	9	3	2	3	18	1	0	1006
	4	0	2	14	972	1	19	1	10	14	13	1046
	5	0	1	7	0	915	2	3	5	6	9	948
	6	3	1	0	8	0	822	10	0	3	2	849
	7	9	5	9	1	11	11	923	1	9	1	980
	8	1	1	6	4	1	2	0	923	2	2	942
	9	5	6	15	5	2	17	4	3	919	6	982
	10	1	0	3	9	47	9	1	50	10	968	1098
	TOTAL PREDICTED/LABEL	980	1135	1032	1010	982	892	949	1028	974	1009	9991

Conclusion

The initial purpose of this investigation and project was to determine whether an ANN would be able to classify various one-digit decimal numbers, and whether the algorithm that was used classified each image accurately. The two independent variables were the authors of the handwritten numbers and the different numbers that were written. For the ANN to be effective in the real world, it must be able to detect the fact that even if two numbers have different handwritings, they are still the same number. To allow the algorithm to account for various handwriting styles, various authors were used to write numbers for both the training and the testing sets. In this analysis, the ANN had a high accuracy of 94.86% when it came to labelling handwritten numbers that were between 0 and 9.

In conclusion, I believe that this ANN was successful at determining and labelling handwritten numbers due to its high accuracy.