Machine Learning and Data Mining

Mini Project 3 Report

Instructor: Prof. Shahryar Rahnamayan

Deadline: Monday, March. 24, 11:59 PM, 2022

Members:

Name: Iliya Karac
Student Number: 100703933

Name: Tegveer Singh
Student Number: 100730432

Name: Nicholas Kvrgic
Student Number: 100613386

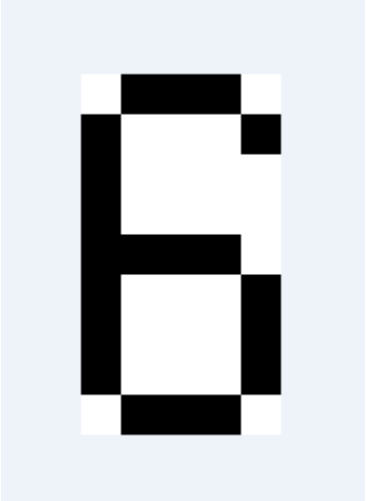
GitHub repository with code:

https://github.com/TegSingh/Bitmap-Recognition-Neural-Network

NOTE: A readme has been added to github repo for the reference. The code to run is main.py

Introduction

In this mini project we were tasked with creating and training a three layer artificial neural and testing to see how well it can recognise modified numbers. The first layer is the input layer. All the nodes in this layer are logically simple and listen to the raw input given to them. This information is a 5 by 9 bitmap that has a number drawn in it. See image below.



Every pixel represents a bit since there are 5x9 pixels in the image it is easily mapped onto an array that has 9 arrays representing the rows. Each row array inside the main one contains 5 bits representing each column within that row. In the row arrays the bits can be 1 or 0, 1 means the pixel is black and 0 means the pixel is white. A representation of the bitmap can be seen below.

```
60
61   value6 = [[0, 1, 1, 1, 0],
62   [1, 0, 0, 0, 1],
63   [1, 0, 0, 0, 0],
64   [1, 0, 0, 0, 0],
65   [1, 1, 1, 1, 0],
66   [1, 0, 0, 0, 1],
67   [1, 0, 0, 0, 1],
68   [1, 0, 0, 0, 1],
69   [0, 1, 1, 1, 0]]
70
```

Since there are 45 (5x9) bits in an image there has to be 45 nodes in the input layer to listen to each bit. In the second layer, the hidden layer, there are 10 nodes. The reason our artificial neural network only has 1 hidden layer and so few nodes is because the input layer has a relatively small amount of values and the set of values is crisp, the value can only be 1 or 0. For this reason we do not need as many hidden nodes to calculate the values and send their conclusion to the output layer. The 10 nodes in the middle layer represent a pattern that can be found in the number for example 4s and 7s have a prominent diagonal line that starts somewhere in the top right corner and ends in the bottom left corner. If a sufficient number of nodes from the input layer fire off to signify this type of pattern the network becomes more confident that the bive bitmap represents a 4 or 7. Based on which other nodes in this layer activate they will send signals to the output layer to decide what the given number was. This leaves the last layer, the output. By this time the data has already been interpreted and the output layer has received the final result. The output layer is composed of 10 nodes, 1 node for every digit. When the neural network finishes interpreting the inputs each output node receives a value. These output values are the certainty. The neural network can output the certainty values for every number but it is set to only return 1 guess (the number with the highest certainty value).

Training the artificial neural network

The training data is formatted as an ideal bitmap for each number value and the expected value for that bitmap. These values are fed to the neural network and the intal weights are set at random for the hidden layer. When the bitmaps are interpreted and the certainty value for the

given number is compared to 1 if the difference between these values is greater than the set toleration threshold we start back propagation. In this process we use the error percentage and correct the weights for all the relationships between the input and middle layer as well as the middle and output layer. This is done for a specified amount of epochs. If there are too little epochs the weights will not be fine tuned enough and the predictions will not be accurate. If there are too many epochs the neural network will be over trained and will not recognize numbers that slightly deviate from the ideal bitmap representing that number.

Process of Back Propagation

This process is carried out in the following steps:

1. Loop through the outermost layer(10). Note that the output has already been calculated through forward propagation
2. Calculate the cost by subtracting the expected output from the current layer output
3. Find the Root Mean Square of that value
4. Calculate the derivative of the RMS for later calculations
5. Loop through the hidden layer (10)
6. Get weight combinations for the weight matrix through i and j values

Calculating the Updated Weight Value

7. Calculate the following value:

$$sigmoid_{backward} = sigmoid_{forward} * (1 - sigmoid_{forward})$$

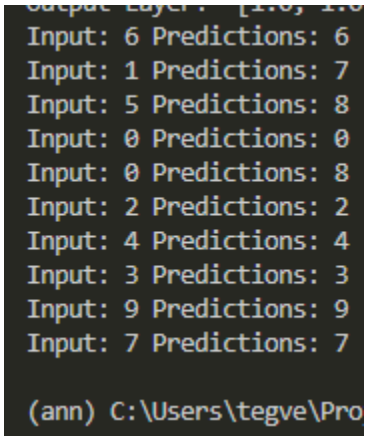
8. Multiply this value by the cost to get the gradient

$$Weightgradient = Cost * Sigmoid_{backward}$$

9. Multiply this by the input activation value calculated during forward propagation
10. Subtract the above calculated value from the original weight to get the updated value of weight
11. Repeat the same process for weights between the input layer and hidden layer by creating another nested for loop(45)
12. Repeat Steps 1-11 for all 30 test cases to train the model

Output:

Current Model shows 65% accuracy



The above output was received by using Tensorflow to create an artificial neural network. If our own neural network is used, the output has very low accuracy of about 25%. It is assumed that could be due to the following reasons:

1. Calculation errors in back propagation
2. Rounding of values to 1
3. Less number of hidden layers
4. Biases not included in the ANN

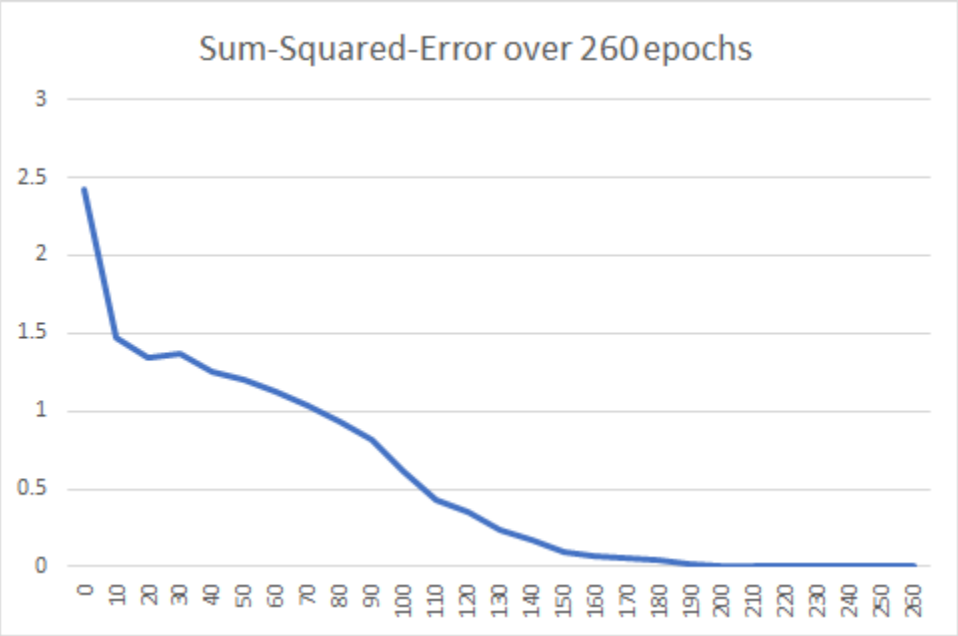
Did we come up with a more compact neural network?

Yes initially we started with 23 nodes in the middle layer. The thought process behind this decision is our experience with rule based expert systems. We thought that more “rules” would help the neural network in terms of accuracy. This was a mistake for 2 reasons.

1. The network would get stuck on local peaks. It would get stuck on a specific node that ultimately did a 50/50 guess on the possible outcome, every time the eight was re-distributed this node would either mess up the outcome or the weight would be so low that the node would become redundant.
2. This is a fallacy because rule based expert systems and artificial neural networks are fundamentally different. Rule based experts rely on rules for accuracy and cannot learn from previous experiences. Neural networks rely on weights for accuracy and learn from previous experiences. Networks make mistakes and sharpen their “rules” by shifting

weights to adjust for the error they made. Having more redundant nodes hinders this process as opposed to expert systems where more rules means higher accuracy usually. We shortened this down to 10 nodes in the middle layer, everyone of these nodes was set to recognise common patterns in order to predict each number. As for the input and output layer they are fixed at 45 nodes and 10 nodes because there are a guaranteed number of inputs and outputs each time.

Sum Squared Error vs Epochs



30 Test Cases

The code on github has a file called input_values.txt which contains bitmaps in the form of 9X5 matrices. These are all appended to a list making it a 3D array. This is fed to the main.py file which contains the generate_dataset method. This method updates the values by creating mutations. The number of test cases and number of mutations can be controlled by changing the following constants in the code.

```
TEST_CASES_PER_MAP = 3
NUM_MUTATIONS = 2
INPUT_LAYER_LENGTH = 45
HIDDEN_LAYER_LENGTH = 10
OUTPUT_LAYER_LENGTH = 10
LEARNING_RATE = 0.3
```

The 2D arrays are converted to 1D arrays by the generate_dataset method. In addition to this, the method performs mutations and adds them to the input_dataset variable.

Mutation: Each mutation switches a 1-bit to 0 and 0-bit to 1. These bits are randomly chosen

Output Generation: As mentioned above the output layer contains 10 values with a 0-1 bit indicating whether that value is indeed the expected output.

For instance, if the value is 3, the output dataset will be

[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

By following the above mentioned process, these methods were created

```
# Loop through the values
for i in range(10):
    # 1D layer to act as input
    input_layer = []

    # Loop through the 9 X 5 matrix for bitmap
    for j in range(9):
        for k in range(5):
            input_layer.append(values[i][j][k])

    self.input_dataset.append(input_layer)

    for p in range(TEST_CASES_PER_MAP - 1):
        # print("Original input layer: ", input_layer)
        self.input_dataset.append(self.perform_mutate(input_layer))

    output_layer = []
    for p in range(10):
        if p == i:
            output_layer.append(1)
        else:
            output_layer.append(0)

    for p in range(TEST_CASES_PER_MAP):
        self.output_dataset.append(output_layer)
```

Weight matrices: These matrices are just random floating point values between the numbers 0-1 and matrix size depends on the 2 layers these weights exist between. There are 2 weight matrices in our case

Weight1: Between Input Layer and Hidden Layer. Size 10 X 45

Weight2: Between Hidden Layer and Output Layer. Size 10 X 10

The above process leads to the generation of the following dataset:

```
(ann) C:\Users\tegve\Projects2022\Bitmap-Recognition>python main.py
Dataset:
Input Layers:
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
[0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1]
[0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0]
[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0]
[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0]
[1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1]
[1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0]
[0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]

Output Layers:
```

```
 Output Layers:
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

 Weights between Input layer and Hidden layer:
```

Problems faced

Some issues were encountered during the completion of this project. They were as follows:

1. *Python rounding:*

Python rounded the values that are really close to 1 to 1 due to which the values end up being rounded. The first sigmoid function during forward propagation leads to values of the format 0.99999 onwards. Another sigmoid applied to this leads to a value which is so close to 1 that python rounds it 1. This was solved by using Decimal class for calculations.

2. *Complex calculations for weights between input and hidden layer*

Calculating the weight updates between the input layer and the hidden layer

Conclusion

Despite some shortcomings, overall the project was successful. We made the neural network and remade it to be more compact. We learned about artificial neural networks and how they function from how each node calculates the value it will send to the next neuron to back propagation. How the network compares its prediction to the actual answer then adjusts the weightings for each relationship to fix the mistake. The mathematical equations used in this project were far more complex than the logic used in previous projects. Most of our errors came from misunderstanding the sigmoid function and using it incorrectly. Initially the weights would change at random and the neural network would not learn at all. Sometimes it would get stuck in a permanent feedback loop and always return the same prediction. Eventually we made a working model that would predict the numbers confidently and accurately. As an added bonus we added functionality where a user can draw a digit and the artificial neural network would try and estimate what number the image was showing. The problem with this is that drawing would give you a fuzzy set rather than a bitmap. We fixed this problem by setting a threshold to see the brightness of the pixel. This made the set back into a crisp bitmap.