CIS3187 - Assignment
Neural Networks for Data Mining

Daniel Magro
3^{rd} year in Bachelor of Science in Information Technology (Honours) (Artificial Intelligence)

January 2019

Statement of Completion

Item	Completed
Create a Boolean Function with 5 inputs and 3 outputs	Yes
Implement a Neural Network with 5 input neurons, 4 hidden neurons and 3 output neurons	Yes
Implement the Error Back Propagation algorithm	Yes
Plot the Bad Facts vs. Epochs Graph	Yes
Network should Converge in less than 1000 Epochs	Yes

Listing of Binary Function

Table 1 shows the expected output for each input of the Binary Function. The function was generated as follows:

- out_1 = in_1 AND in_3
- out_2 = in_2 OR in_5
- out_3 = NOT(in_2) AND in_4

Table 1 includes both the training instances and the testing instances. These will be shuffled and split in a 26 training instances : 6 testing instances ratio by the python script.

i_1	i_2	i_3	i_4	i_5	o_1	o_2	o_3
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	0	0	1	0	0	0	1
0	0	0	1	1	0	1	1
0	0	1	0	0	0	0	0
0	0	1	0	1	0	1	0
0	0	1	1	0	0	0	1
0	0	1	1	1	0	1	1
0	1	0	0	0	0	1	0
0	1	0	0	1	0	1	0
0	1	0	1	0	0	1	0
0	1	0	1	1	0	1	0
0	1	1	0	0	0	1	0
0	1	1	0	1	0	1	0
0	1	1	1	0	0	1	0
0	1	1	1	1	0	1	0
1	0	0	0	0	0	0	0
1	0	0	0	1	0	1	0
1	0	0	1	0	0	0	1
1	0	0	1	1	0	1	1
1	0	1	0	0	1	0	0
1	0	1	0	1	1	1	0
1	0	1	1	0	1	0	1
1	0	1	1	1	1	1	1
1	1	0	0	0	0	1	0
1	1	0	0	1	0	1	0
1	1	0	1	0	0	1	0
1	1	0	1	1	0	1	0
1	1	1	0	0	1	1	0
1	1	1	0	1	1	1	0
1	1	1	1	0	1	1	0
1	1	1	1	1	1	1	0

Table 1: The Inputs and Outputs of the Binary Function to be learnt

Bad Facts against Epochs Graph

Figure 1 shows the graph of the Percentage of Bad Facts obtained during each Training Epoch.

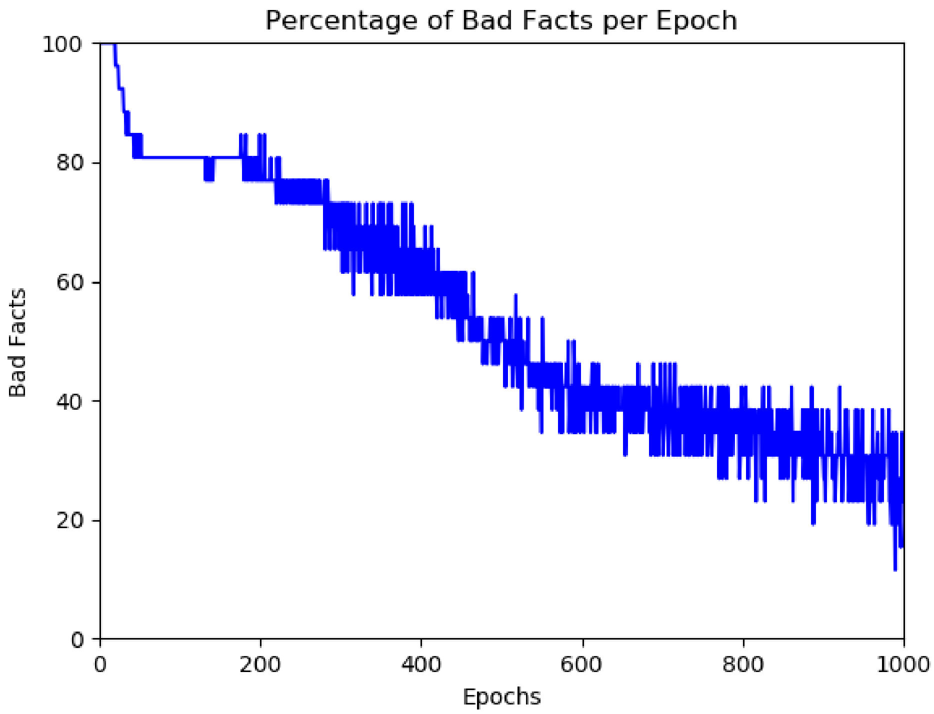

Figure 1: Bad Facts against Epochs Graph

Conclusions

As shown in Figure 1, the Neural Network doesn't converge (0 bad facts) within 1000 epochs.
However, it still manages to obtain an 83.33% performance, meaning it correctly predicts the output of 5/6 test instances.

To make the NN converge within 1000 epochs, the hyper parameters can be tweaked slightly.

The learning rate can be adjusted by changing the value in line 37 of the code.
The learning rate can be raised from 0.2 to 0.25 to make the NN converge within 900 Epochs, and still achieve the same Performance. This is shown in Figure 2.
Raising the learning rate to 0.5 makes the NN converge even faster, in less than 500 Epochs, again achieving the same performance.
Having an unreasonably high learning can, however, lead to overfitting.

Alternatively, the number of nodes in the hidden layer can also be changed. The number of nodes can be adjusted by changing the value in line 33 of the code.
By setting the number of nodes to 7, instead of 4, the number of Epochs needed for convergence dropped to less than 375, while the Performance shot up to 100%. This is shown in Figure 3.
While more nodes in the hidden layer very often results in better performance, the computational cost increases for each node added.

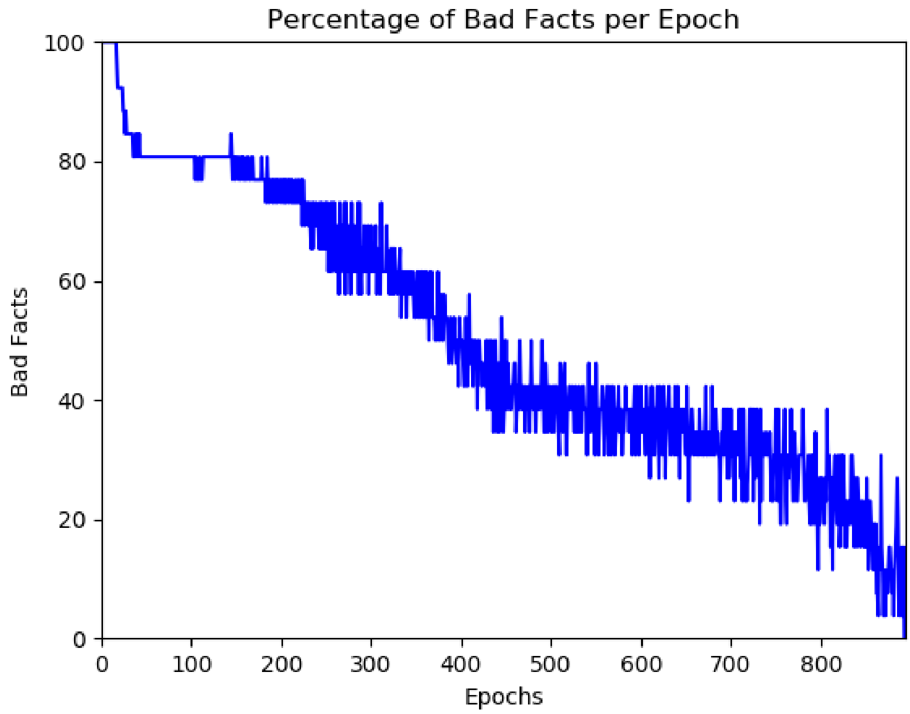

Figure 2: Bad Facts against Epochs Graph
increased learning rate

Notes & Implementation Details

The randomness is seeded in line 7 of the program. This can be commented out for randomness with each run.

The way that the instances of the binary function are split between the training set and the test set is random, and thus varies with each run, unless seeded.

The weights of the NN are also initialised randomly, to very small real numbers, unless seeded. After running the script a few times it became evident that this initialisation can have a significant effect on the performance of the NN, as sometimes the network would take up to 200 epochs more to converge, sometimes even achieving poorer performance than other runs.

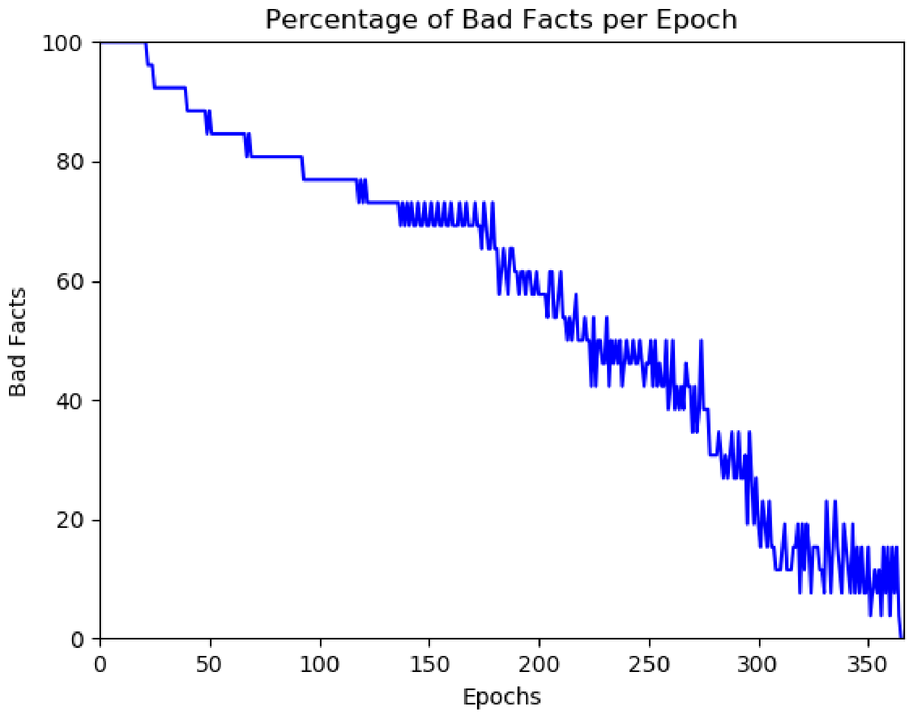

Figure 3: Bad Facts against Epochs Graph
increased number of nodes in the hidden layer

When testing the Neural Network, an error threshold of 0.5 is used rather than 0.2, i.e. values >= 0.5 become 1 and values < 0.5 become 0. This makes it possible to compare the values in the output of the NN (which will be floats) to the binary values in the target.

Source Code Listing

```python
import numpy as np          # for matrix multiplication, e
import matplotlib
import matplotlib.pyplot as plt
from typing import List     # for type annotation

# seeds the randomness for repeatable results, this can be commented out
np.random.seed(0)


# method which computes sigmoid for any input value
def sigmoid(z: float) -> float:
    return 1 / (1 + np.exp(-z))


# read the binary function which is stored as a CSV File
with open('BinaryFunction.csv', 'r', encoding='utf-8') as f:
    data = [line.strip().split(',') for line in f.read().strip().split('\n')]
    # remove the first row, containing the header of each column
    data.pop(0)
    data = [[int(j) for j in i] for i in data]

# Randomising the order of the data
np.random.shuffle(data)
# Splitting the data into a training set (26) and a test set (6)
training_data = data[0:26]
test_data = data[26:32]


# Hyper Parameters
# maximum number of epochs the learning algorithm should run for (if convergence is not reached)
max_number_of_epochs: int = 1000
# number of nodes that will be used in the hidden layer
nodes_in_hidden_layer = 4
# the error threshold
error_threshold = 0.2
# the learning rate
learning_rate = 0.2

# initialising a matrix representing the weights of the hidden layer as a 5 x (no. of nodes in hidden layer) matrix
# to a matrix of small random numbers
```

```python
hidden_layer_weights = np.random.normal(0, 0.1, (5, nodes_in_hidden_layer))
# initialising a matrix representing the weights of the output layer as a (no. of nodes in hidden layer) x 3 matrix
# to a matrix of small random numbers
output_layer_weights = np.random.normal(0, 0.1, (nodes_in_hidden_layer, 3))

# storing the percentage of bad facts per epoch in a list of floats
percentage_of_bad_facts_per_epoch: List[float] = list()

number_of_epochs: int = 0
# Do the training process until the termination condition is met
# Termination condition: Run training until either The Number of Bad Facts is 0
# or some maximum number of epochs have been executed
while True:
    # store the number of bad facts encountered during this epoch
    number_of_bad_facts: int = 0

    # Run all the training instances through the neural net every epoch
    for j in range(len(training_data)):
        # separate the first 5 elements of a training instance (input) into a separate 1x5 matrix
        input = [training_data[j][0:5]]
        # separate the last 3 elements of a training instance (target) into a separate 1x3 matrix
        target = [training_data[j][5:8]]

        # calculate the net of the hidden layer as being the matrix multiplication of the input and the weights of the
        # hidden layer. result is a 1x4 matrix
        hidden_layer_net = np.matmul(input, hidden_layer_weights)
        # compute sigmoid on the net of the hidden layer. result remains a 1x4 matrix
        hidden_layer_output = np.zeros((1, nodes_in_hidden_layer))
        for k in range(len(hidden_layer_net[0])):
            hidden_layer_output[0][k] = sigmoid(hidden_layer_net[0][k])

        # calculate the net of the output layer as being the matrix multiplication of the output of the hidden layer
```

```python
        # and the weights of the output layer. result is a 1x3 matrix
        output_layer_net = np.matmul(hidden_layer_output, output_layer_weights)
        # compute sigmoid on the net of the output layer. result remains a 1x3 matrix
        output_layer_output = np.zeros((1, 3))
        for k in range(len(output_layer_net[0])):
            output_layer_output[0][k] = sigmoid(output_layer_net[0][k])

        # compute the error
        error = target - output_layer_output

        # check if the error threshold has been exceeded by any of the bits.
        threshold_exceeded: bool = False
        for k in error[0]:
            if abs(k) > error_threshold:
                threshold_exceeded = True
                # increment the number of bad facts
                number_of_bad_facts += 1
                break

        if threshold_exceeded:
            # Do Error Back Propagation

            # compute δ of the each neuron in the output layer. There should be 3 values
            output_layer_deltas = np.zeros((1, 3))
            for k in range(len(output_layer_deltas[0])):
                # set δ of node k in the output layer to:
                output_layer_deltas[0][k] = output_layer_output[0][k] \
                                            * (1 - output_layer_output[0][k]) \
                                            * (target[0][k] - output_layer_output[0][k])
            # Adjust the weights of the output layer
            for k in range(len(output_layer_weights)):
                for l in range(len(output_layer_weights[k])):
                    # set Δw_kl_o to the learning rate * δ of node l in the output layer
                    # * the output of node k in the hidden layer
                    delta_kl_o = learning_rate * output_layer_deltas[0][l] * hidden_layer_output[0][k]

                    # Updating the weight of the connection
```

```python
from node k in the hidden layer
                    # to node l in the output layer
                    output_layer_weights[k][l] += delta_kl_o


            # compute δ of the each neuron in the hidden layer. There should be 4 values
            hidden_layer_deltas = np.zeros((1, nodes_in_hidden_layer))
            for k in range(len(hidden_layer_deltas[0])):
                # loop over the nodes that are downstream of the current node in the hidden layer to calculate sigma
                sigma: float = 0
                for l in range(len(output_layer_weights[k])):
                    sigma += output_layer_weights[k][l] * output_layer_deltas[0][l]

                # set δ of node k in the hidden layer to:
                hidden_layer_deltas[0][k] = hidden_layer_output[0][k] \
                                            * (1 - hidden_layer_output[0][k]) \
                                            * sigma
            # Adjust the weights of the hidden layer
            for k in range(len(hidden_layer_weights)):
                for l in range(len(hidden_layer_weights[k])):
                    # set Δw_kl_h to the learning rate * δ_l_h * the output of node k in the input layer (i.e. the input)
                    delta_kl_h = learning_rate * hidden_layer_deltas[0][l] * input[0][k]

                    # Updating the weight of the connection from node k in the input layer
                    # to node l in the hidden layer
                    hidden_layer_weights[k][l] += delta_kl_h

    # store the percentage of bad facts found during this epoch
    percentage_of_bad_facts_per_epoch.append((number_of_bad_facts / len(training_data)) * 100)
    # increment the number of epochs
    number_of_epochs += 1

    # Termination condition: Run training until either The Number of Bad Facts is 0
    # or some maximum number of epochs have been executed
    if number_of_epochs >= max_number_of_epochs:
        print("Neural Network has not converged after the max
```

```
 number of epochs (" + str(max_number_of_epochs) + " epochs), stopping training now.")
        break
    if percentage_of_bad_facts_per_epoch[-1] == 0:
        break

# Calculating test error (Performance)
test_good_facts: int = 0
for i in range(len(test_data)):
    # separate the first 5 elements of a test instance (input) into a separate 1x5 matrix
    input = [test_data[i][0:5]]
    # separate the last 3 elements of a test instance (target) into a separate 1x3 matrix
    target = [test_data[i][5:8]]

    # calculate the net of the hidden layer as being the matrix multiplication of the input
    # and the weights of the hidden layer. result is a 1x4 matrix
    hidden_layer_net = np.matmul(input, hidden_layer_weights)
    # compute sigmoid on the net of the hidden layer. result remains a 1x4 matrix
    hidden_layer_output = np.zeros((1, nodes_in_hidden_layer))
    for k in range(nodes_in_hidden_layer):
        hidden_layer_output[0][k] = sigmoid(hidden_layer_net[0][k])

    # calculate the net of the output layer as being the matrix multiplication of the output of the hidden layer
    # and the weights of the output layer. result is a 1x3 matrix
    output_layer_net = np.matmul(hidden_layer_output, output_layer_weights)
    # compute sigmoid on the net of the output layer. result remains a 1x3 matrix
    output_layer_output = np.zeros((1, 3))
    for k in range(len(output_layer_net[0])):
        output_layer_output[0][k] = sigmoid(output_layer_net[0][k])

    # since the target is made up of bits (0 or 1)
    # and the outputs of the Neural Net will be floats in the range [0,1]
    # values >= 0.5 in the output are set to 1, whereas values < 0.5 are set to 0.
    output_layer_output[output_layer_output >= 0.5] = 1
    output_layer_output[output_layer_output < 0.5] = 0
```

```python

    # the output is then compared to the target. If all bits are equal, it is counted as a good fact
    if np.array_equal(output_layer_output, target):
        test_good_facts += 1

# The performance is the number of good facts over the total number of test instances
print("Number of Epochs: " + str(number_of_epochs))
performance: float = (test_good_facts/len(test_data)) * 100
print(f"Performance: {performance.__format__('.2f')}%")

# Plot the Percentage of Bad Facts against Epochs Graph
# the x-values of each point are the epoch in which each percentage of bad facts was obtained. (0, 1, 2, 3, ...)
# the y-values are the percentage of bad facts during that epoch. (eg. 100%, 96%, 92%, ..., 4%, 0%)
x = np.arange(len(percentage_of_bad_facts_per_epoch))
plt.plot(x, percentage_of_bad_facts_per_epoch, color='blue', linestyle='-')
plt.title("Percentage of Bad Facts per Epoch")
plt.xlabel("Epochs")
plt.ylabel("Bad Facts")
plt.xlim(left=0)
plt.xlim(right=number_of_epochs)
plt.ylim(bottom=0)
plt.ylim(top=100)
plt.show()
```