1 Setup

This assignment was written using pipenv for dependency management. Pipenv uses a Pipfile to store dependency information, however a requirements.txt file was also included for convenience.

1.1 Requirements

The setup assumes you have the following installed:

- python 3.6;
- Pipenv or venv.

The project also depends on the following libraries to run:

- numpy version 1.15;
- pandas version 0.23;
- matplotlib version 3.0.1.

1.2 Installing dependencies Using Pipenv

```
$ pipenv install # Install dependencies
$ pipenv shell # Activate Environment
```

1.3 Installing Dependencies Using Virtual Environment

```
$ python3 -m venv venv/ # create virtual Environment
$ source venv/bin/activate # Activate Environment
$ pip install -r # Install dependencies
```

2 Data Sets

2.1 Types of Test Data

The neural net was challenged with three different datasets:

- A simple problem where the output is (x[0], x[1], x[4])
- A normal complexity problem where the output is (x[0] | x[4], x[1] & x[2], x[4])
- A hard problem where the output is (x[0] | x[4], x[1] XOR x[2], x[3] & x[4])

2.2 Running the Data Generation Script

For convenience a script was included to automate the generation of test data called generate_test_data.py. Data generated will be located in the resources directory. Invoke the following command to generate the test data.

```
$ chmod +x generate_test_data.py
$ ./generate_test_data.py
```

Listing 1: Generating the test data

3 Neural Net

The python script neural_net contains a class that creates a neural network and implements the error back propagation algorithm.

3.0.1 Running the Neural Network

Listing 2 demonstrates how to run the neural network.

```
$ chmod +x neural_net.py # change permission to allow execution
$ ./neural_net.py -h # shows all available arguments.
$ ./neural_net.py --dataset simple_problem # trains the network for the simple_problem dataset for 1000 epochs
$ ./neural_net.py --dataset hard_problem --epochs 4000 # trains the network on the hard_problem dataset for 4000 epochs
```

Listing 2: Running the Neural Network

3.1 Results

For convenience the script outputs a confusion matrix with the associated statistics to measure the neural net's performance. The matrix is represented as follows:

$$\begin{bmatrix} TP & FN \\ FP & TN \end{bmatrix}$$

3.1.1 Simple Problem

Figure 1 shows the performance of the neural net on the simple problem dataset.

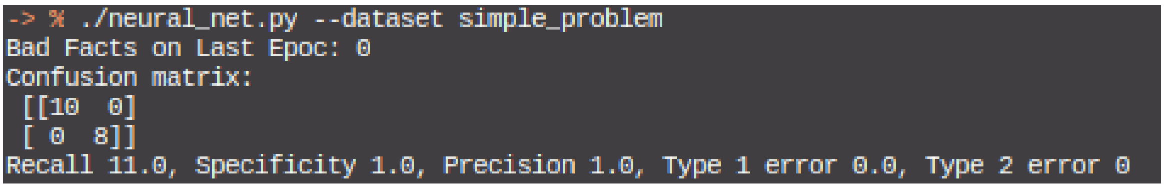


Figure 1: Simple problem performance.

Figure 2 shows a graph of the bad facts against epochs on the simple problem dataset.

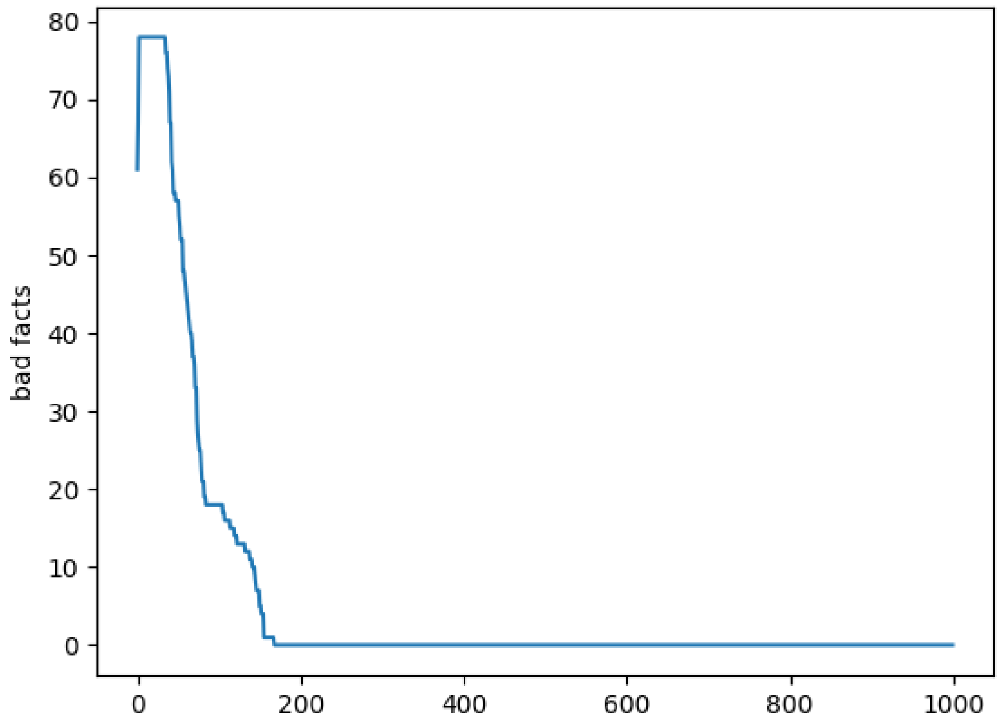
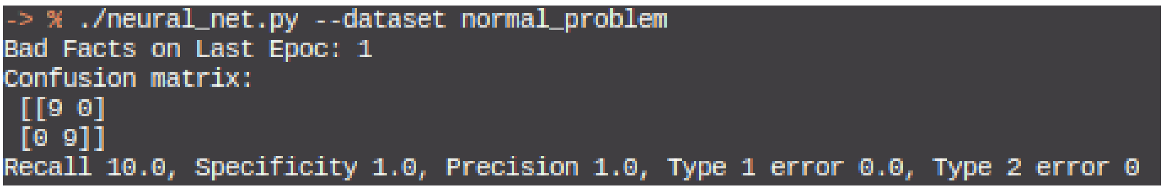


Figure 2: Graph of Bad facts vs Epoch.

3.1.2 Normal Problem

Figure 3 shows the performance of the neural net on the normal problem dataset.



```
-> % ./neural_net.py --dataset normal_problem
Bad Facts on Last Epoc: 1
Confusion matrix:
 [[9 0]
 [0 9]]
Recall 10.0, Specificity 1.0, Precision 1.0, Type 1 error 0.0, Type 2 error 0
```

Figure 3: Normal problem performance.

Figure 4 shows a graph of the bad facts against epochs on the normal problem dataset.

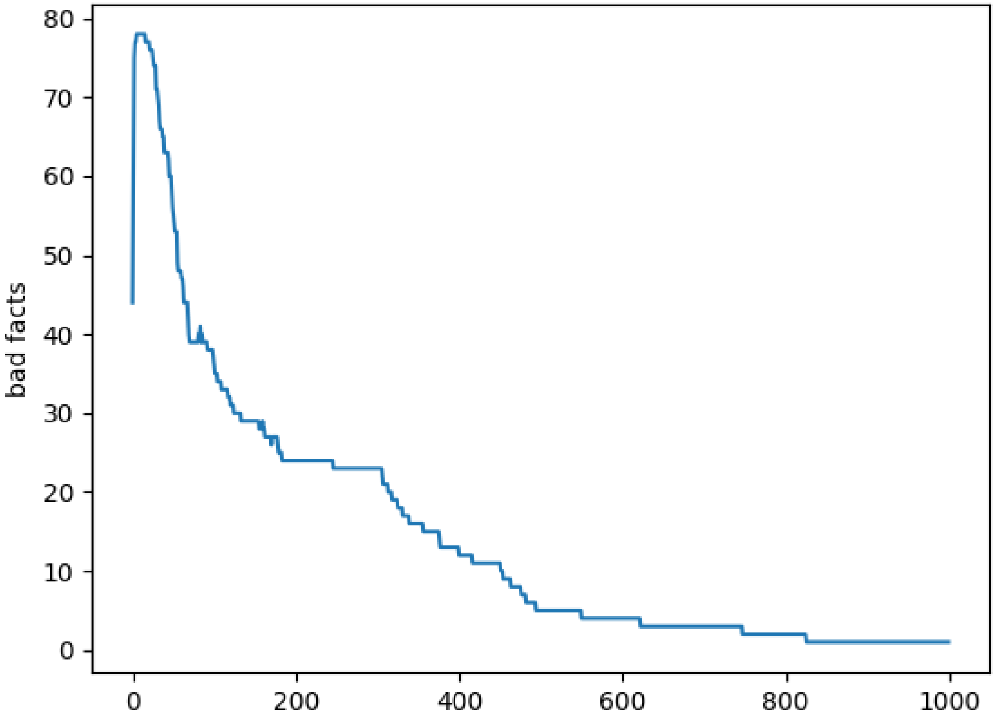


Figure 4: Graph of Bad facts vs Epoch for the normal dataset.

3.1.3 Hard Problem

Figure 5 shows the performance of the neural net on the hard problem dataset.

```
-> % ./neural_net.py --dataset hard_problem
Bad Facts on Last Epoc: 18
Confusion matrix:
 [[11  0]
 [ 3  4]]
Recall 12.0, Specificity 0.5714285714285714, Precision 0.7857142857142857, Type 1 error 0.42857142857142855, Type 2 error 0.42857142857142855
```

Figure 5: Hard problem performance.

Figure 6 shows a graph of the bad facts against epochs on the hard problem dataset.

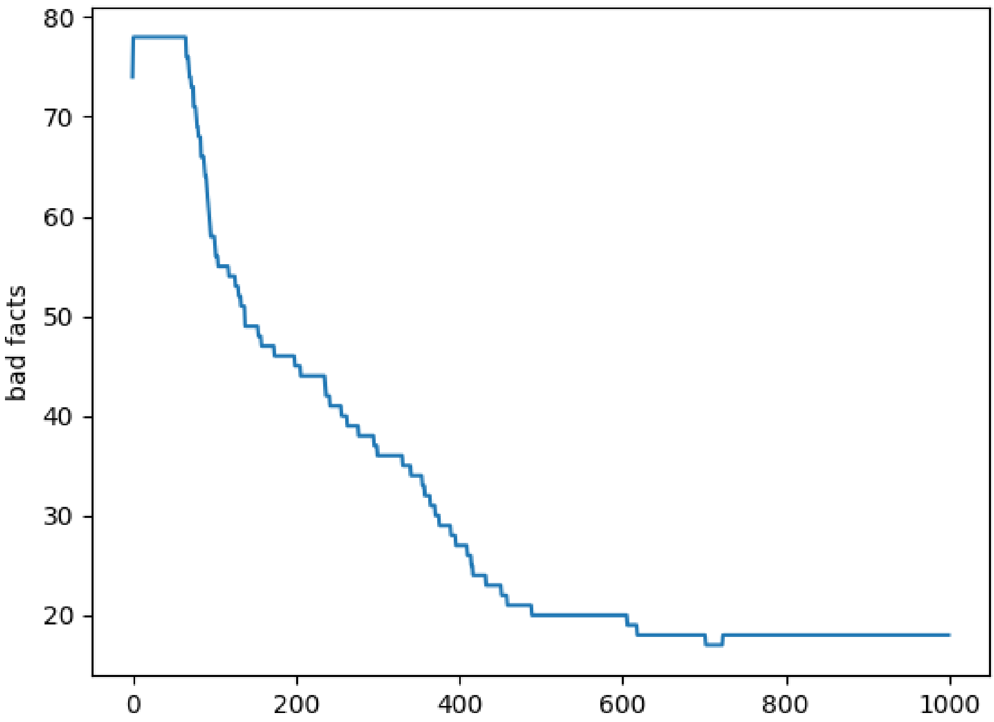


Figure 6: Graph of Bad facts vs Epoch for the hard dataset.

4 Code

4.1 Csv Utils Script

A class dedicated to manipulate CSV files was created called csv_utils.

```python
import numpy as np
import pandas as pd
import csv
import random
import os

resource_loc = 'resources'


def to_matrix(n: int):
    """
    A helper function that generates a matrix of bit combinations
    :param n: size of the matrix
    :return: A matrix with the input combinations
    """
    def gen(n: int):
        for i in range(1, 2 ** n - 1):
            yield '{:0{n}b}'.format(i, n=n)

    matrix = [[0 for i in range(n)]]
```

```python
    for perm in list(gen(n)):
        matrix.append([int(s) for s in perm])
    matrix.append([1 for i in range(n)])
    return matrix


def transformation(x):
    """
    A function that transforms 5 inputs into 3 outputs. (generates hard data set)
    :param x: A list to transform
    :return: The transformed list of size 3
    """
    return [x[0] | x[4], x[1] ^ x[2], x[3] & x[4]]


def generate_data_to_csv(matrix_size: int, file_name: str = 'hard_problem', transformation_function=transformation):
    """
    A helper function to aid in the generation of csv data
    :param transformation_function: The transformation function for generating the output bits
    :param matrix_size: The size of the input matrix
    :param file_name: The file name to produce
    :return: Input matrix and output matrix
    """
    input_array = np.asarray(to_matrix(matrix_size))
    output = np.apply_along_axis(transformation_function, 1, input_array)
    data_frame = pd.DataFrame(np.concatenate((input_array, output), axis=1))
    data_frame.to_csv(os.path.join('resources', f'{file_name}.csv'), header=None, index=None)
    return input_array, output


def import_data_from_csv_and_split(filename: str = 'hard_problem', input_range=5):
    """
    A function that parses a csv file with training data and converts it into a matrix
    :param filename: a string representation of the file name
    :param input_range: the size of input data
    :return: a tuple (input matrix, output matrix)
    """
    with open(os.path.join('resources', f'{filename}.csv')) as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        input_matrix = []
        output_matrix = []
        for row in csv_reader:
            input_matrix.append(row[:input_range])
            output_matrix.append(row[input_range:])
        return input_matrix, output_matrix

```

```python

def import_plain_data_from_csv(filename: str = 'hard_problem'):
    """
    A function that is responsible for importing data from csv
    :param filename: a string representing the file name
    :return: a list representing the csv
    """
    with open(os.path.join('resources', f'{filename}.csv')) as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        return [row for row in csv_reader]


def split_matrix_into_training_set(data: list, train_size=26) -> (list, list):
    """
    A method that splits a matrix into a training matrix a verification matrix
    :param data: the matrix
    :param train_size: the size of the training data
    :return: a tuple (training data, verification data)
    """
    data_copy = data
    random.shuffle(data_copy)
    return data_copy[:train_size], data_copy[train_size:]
```

4.2 Generate Data Script

An executable script to generate test data.

```python
#!/usr/bin/env python3
from utils.csv_utils import generate_data_to_csv, transformation

generate_data_to_csv(5, "simple_problem", lambda x: [x[0], x[1], x[4]])
generate_data_to_csv(5, 'normal_problem', lambda x: [x[0] | x[4], x[1] & x[2], x[4]])
generate_data_to_csv(5, 'hard_problem', transformation)
```

Listing 3: Generate test Data

4.3 Neural Network Script

An executable script that implements the error back propagation algorithm.

```python
#!/usr/bin/env python3

import numpy as np
import matplotlib.pyplot as plt
import argparse

from utils.csv_utils import split_matrix_into_training_set, import_plain_data_from_csv

```

```
def sigmoid(x: np.ndarray):
    """
    A function that works out sigmoid for x
    :param x: a numpy array or number to work out sigmoid
    :return:
    """
    return 1 / (1 + np.exp(-x))


def derivatives_sigmoid(x: np.ndarray):
    """
    A function that works out the sigmoid derivation for x
    :param x: a numpy array or number to work out sigmoid prime
    :return: the value of the sigmoid derivative
    """
    return x * (1 - x)


class NeuralNet(object):
    """
    A class that represents a neural net
    """
    def __init__(self, inputs: np.ndarray, hidden: int, outputs: np.ndarray, error_threshold: float = 0.2,
                 learning_rate: float = 0.2, wh=None, wo=None, activation=sigmoid,
                 activation_derivation=derivatives_sigmoid) -> None:
        """
        A constructor that initializes the Neural net with the required parameters.
        :param inputs: a numpy array representing the inputs
        :param hidden: an int representing the size of the hidden layer
        :param outputs: a numpy array representing the outputs
        :param error_threshold: a float representing the error threshold
        :param learning_rate: a float representing the learning rate
        :param wh: a numpy array representing the hidden layer weights
        :param wo: a numpy array representing the output layer weights
        :param activation: the activation function
        :param activation_derivation: the derivation of the activation function
        """
        self.activation = activation
        self.activation_derivation = activation_derivation

        self.inputs = inputs
        self.hidden = hidden
        self.targets = outputs

        self.error_threshold = error_threshold
        self.learning_rate = learning_rate

        if wh is None:
```

```python
            self.wh = np.random.random_sample((inputs[0].size, hidden))
        else:
            self.wh = wh
        if wo is None:
            self.wo = np.random.random_sample((hidden, outputs[0].size))
        else:
            self.wo = wo

    def __round_nn_output__(self, value: int):
        """
        A helper function that rounds a value using the error threshold as a midpoint
        :param value: the value to round
        :return: 0 IFF value < error threshold ELSE 1
        """
        return 0 if value <= self.error_threshold else 1

    def __calc_bad_facts__(self, errors: np.ndarray) -> int:
        """
        A helper function that calculates the amount of bad facts in an epoch.
        :param errors: a numpy array representing the errors
        :return: an int representing the bad facts
        """
        bad_facts = 0
        for error in errors:
            bad_facts += len(list(filter(lambda x: (abs(x) >= self.error_threshold), error)))
        return bad_facts

    def feedforward(self, inp: np.ndarray = None) -> (np.ndarray, np.ndarray):
        """
        A feed forward method that allows the neural net to 'think'.
        :param inp: a numpy array representing the inputs
        :return: a tuple representing the output of the hidden and final output
        """
        if inp is None:
            inp = self.inputs
        net_h = np.dot(inp, self.wh)
        out_h = self.activation(net_h)
        net_o = np.dot(out_h, self.wo)
        out_o = self.activation(net_o)
        return out_h, out_o

    def train(self, epoch: int = 1000) -> list:
        """
        A method responsible for training the neural network.
        :param epoch: an int representing the number of iterations over the training data set
        :return: a list of epochs vs bad facts
        """
```

```
        bad_facts = []
        for i in range(epoch):
            out_h, out_o = self.feedforward(self.inputs)
            error = self.targets - out_o
            d_output = error * self.activation_derivation(out_o)

            error_hidden_layer = d_output.dot(self.wo.T)
            d_hidden = self.activation_derivation(out_h) * error_hidden_layer

            layer1_adjustment = self.inputs.T.dot(d_hidden)
            layer2_adjustment = out_h.T.dot(d_output)

            self.wh += self.learning_rate * layer1_adjustment
            self.wo += self.learning_rate * layer2_adjustment

            bad_facts.append(self.__calc_bad_facts__(error))
        return bad_facts

    def get_confusion_matrix(self, x_verify: np.ndarray, y_verify: np.ndarray) -> (np.ndarray, tuple):
        """
        A helper function that feeds forward the inputs, verifies their output and returns the neural net's accuracy.
        :param x_verify: A numpy 2d array of the inputs
        :param y_verify: A numpy 2d array of the expected outputs
        :return: a numpy array representing the confusion matrix and a tuple (recall, specificity, type_one_error,
        type_two_error)
        """
        test_neural_net = NeuralNet(inputs=x_verify, hidden=self.hidden, outputs=y_verify, wh=self.wh, wo=self.wo)
        round_output = np.vectorize(self.__round_nn_output__)
        out_o = round_output(test_neural_net.feedforward(x_verify)[1])
        errors = int(np.sum(out_o == y_verify))
        true_pos = np.sum(np.logical_and(out_o == 1, y_verify == 1))
        true_negative = np.sum(np.logical_and(out_o == 0, y_verify == 0))
        false_positive = np.sum(np.logical_and(out_o == 1, y_verify == 0))
        false_negative = np.sum(np.logical_and(out_o == 0, y_verify == 1))
        recall = true_pos + true_pos / true_pos + false_negative
        specificity = true_negative / (false_positive + true_negative)
        precision = true_pos / (true_pos + false_positive)
        type_one_error = false_positive / (false_positive + true_negative)
        type_two_error = false_positive / (false_positive + true_negative) if false_positive != 0 else 0
        rates = (recall, specificity, precision, type_one_error, type_two_error)
        return np.array([[true_pos, false_negative], [false_positive, true_negative]], np.int32), rates

```

```python

if __name__ == '__main__':
    # Load boolean function dataset (simple_problem, normal_problem, hard_problem)
    parser = argparse.ArgumentParser()
    parser.add_argument('--dataset', default='normal_problem',
                        help='The data set to use: simple_problem, normal_problem, hard_problem. '
                             + 'Default: normal_problem')
    parser.add_argument('--epochs', default='1000',
                        help='The amount of epochs the training algorithm will iterate through. Default: 1000')
    args = parser.parse_args()

    training_data, verification_data = split_matrix_into_training_set(import_plain_data_from_csv(args.dataset))
    X_train = np.asarray([row[:5] for row in training_data], dtype=int)
    y_train = np.asarray([row[5:] for row in training_data], dtype=int)

    neural_net = NeuralNet(X_train, 4, y_train)
    # Neural net initialization
    total_bad_facts = neural_net.train(epoch=int(args.epochs))
    plt.plot(total_bad_facts)
    plt.ylabel('bad facts')
    plt.show()
    print(f"Bad Facts on Last Epoc: {total_bad_facts[len(total_bad_facts) - 1]}")

    # Verification
    X_verif = np.asarray([row[:5] for row in verification_data], dtype=int)
    Y_verif = np.asarray([row[5:] for row in verification_data], dtype=int)
    matrix, rates = neural_net.get_confusion_matrix(X_verif, Y_verif)
    print(f"Confusion matrix: \n {matrix}")
    print(f"Recall {rates[0]}, Specificity {rates[1]}, Precision {rates[2]},"
          + f" Type 1 error {rates[3]}, Type 2 error {rates[4]}")
```

Listing 4: Neural Network