Adaptive Intelligence - Assignment 1

Aleksandra Kulbaka
The University of Sheffield, Sheffield, United Kingdom
awkulbaka1@sheffield.ac.uk

***Abstract*—This report describes the implementation of a supervised learning algorithm which trains a multi-layer feed-forward network to correctly classify a set of images. The network uses a rectified linear activation function (ReLU). Its performance is measured on the subset of the EMNIST data set [9] and compared with the results achieved by Cohen et al. in "EMNIST: an extension of MNIST to handwritten letters" [8].**

I. Introduction

Machine Learning is one of the fastest-growing research areas. It investigates how computers can learn and improve based on data. One of the main challenges in machine learning is classification. It aims to programme computers to automatically learn to recognise patterns from data and to use that knowledge to correctly assign test samples into specific categories [14]. Depending on a data set, different learning methods can be used for solving the classification problem.

Unsupervised learning uses data with no labels to discover hidden patterns which enable to classify samples into different clusters [15]. If only part of the data has labels, semi-supervised learning can be applied. Labelled data is used to learn class models and unlabelled data - to learn and improve the boundaries between different classes. Active learning is another technique that can be applied to a partially labelled data. In this approach, users directly take part in the learning process. They may be asked by a system to label an example that is either taken from the data set or generated during the learning process [14]. Finally, when labels for all the data are available, one can use supervised learning. This machine learning algorithm is applied to the neural network described in this report.

Supervised learning uses a big set of labelled data to update the system parameters such that the total system error is minimised. Error is computed as a distance between output vector x and the expected output vector y. The process of finding an optimal set of parameters is called training [2]. The trained system should correctly classify unseen samples from the same classes [12].

Training a classifier using supervised learning has a few disadvantages. It may be very resource-consuming. If some samples have wrong labels or there are many more samples from one class than from the others, the whole learning process may be biased or incorrect. Finally, supervised learning cannot train the neural network to correctly classify data to the classes not represented in the training data set. However, when used correctly, it is a very effective classifying method [15].

The supervised learning paradigm can be implemented in various different ways, e.g., by using linear or logistic regression, random forest, or k-nearest neighbour algorithm [15]. However, in this report, the focus is only on the neural network implementation.

II. Methods

Artificial neural networks are computing systems widely used in machine learning. They mimic the way that biological neurons communicate with each other in the human brain. They are comprised of the node layers: input layer that takes the input from the environment, hidden layers (none or multiple) which process the input, and the output layer returning the solution [15]. Example neural network is shown in Figure 1.

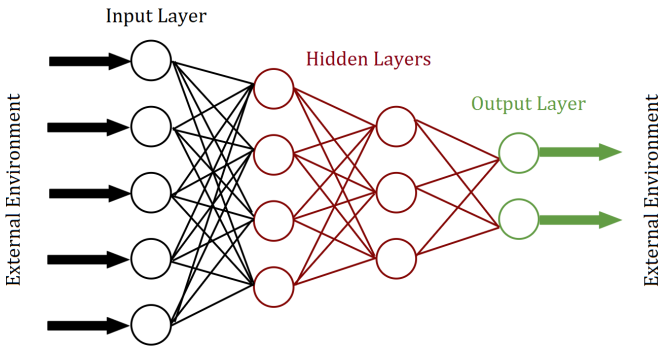

Fig. 1. Example of the neural network architecture

Each node(or artificial neuron) i in layer j consists of inputs ($h_i^{(j)}$), weights ($w_i^{(j)}$), a bias ($b_i^{(j)}$), and an output ($x_i^{(j)}$). The output is computed using the appropriate activation function. If the value exceeds the neuron's

threshold, the neuron is fired and it passes data to the next layer in the network [15]. Neural networks where the output from one layer is used as an input to the next layer (no loops in the network) are called feed-forward neural networks [19].

As previously stated, neural networks can be trained using supervised learning. The ultimate goal is to minimise the cost (also known as loss) function [19]. Algorithm 1 shows the steps of the supervised learning.

Algorithm 1: Supervised Learning [2]

```
Initialize weights and biases.
while error is not 0 do
    Compute neuron output from input pattern.
    Compute weights and biases change.
    Update weights and biases.
end
```

Each of these steps can be implemented in many different ways. In this report, the neural network is implemented as follows:

1) Preparing the data set: The neural network is trained on the subset of EMNIST data set, which contains 20 800 pictures handwritten letters from A to Z (800 pictures per class). The validation data set contains 200 pictures per class and the test set: 250 pictures per class. Each image is comprised of 784 pixels of intensity between 0 and 255. Before training, all the data is normalised such that every pixel has value from 0 to 1. Normalisation is a very beneficial step to take when working with machine learning models. Changing all the values to a common scale speeds up the process of finding optimal weights and biases by making gradient descents converge more quickly [17].

2) Initialising the network: Choice of initialization method affects the whole process of training and can determine whether the algorithm converges or not. A few different initialisation methods has been applied (inc. Xavier and He initialization) [7], [20]. However, initialising biases and weights as random float values from a univariate Gaussian distribution of mean 0 and variance 1 gave the best results [19].

3) Computing Neuron Output: First, the input to each of the neurons of every layer needs to be computed. This is done by taking the outputs from the previous layer, multiplying them by the weights and adding biases. Below, this process is shown for neurons in layer 1:

$$h_i^{(1)} = \sum_{i=1}^{784} w_{ij}^{(1)} x_j^{(0)} + b_i^{(1)} \tag{1}$$

To compute the neurons' outputs x, the Rectified Linear Activation Function (ReLU) is used [4]:

$$f(h) = \begin{cases} h & \text{if } h > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

ReLU has been one of the most popular activation functions for many types of neural networks. This is mainly because it can be easily implemented and it significantly improves the convergence of the stochastic gradient descent compared to e.g., Sigmoid or TanH. ReLU is also a more suitable choice for networks with many layers because it overcomes the problem of vanishing gradients [4]. The problem with a multi-layer neural network is that the gradient decreases significantly when propagated backward so when it reaches the input layer, it may have little or no impact on the weights. Hence the whole training process becomes ineffective. This problem occurs with Sigmoid and TanH functions [5]. However, since the derivative of ReLU is 1 or 0, the update is either full or none - there are no vanishing gradients [13], [21].

Unfortunately, this method has also some drawbacks. Since every negative number becomes 0 in ReLU, backpropagating the large gradient could cause the "death" of many neurons - they would always output 0. During the training, a significant part of the network may "die" and the whole training process will stop decreasing the loss function. One way to deal with "dying ReLU" problem is to use Leaky ReLU which allows a small, positive gradient when the unit is not active [1]:

$$f(h) = \begin{cases} h & \text{if } h > 0 \\ 0.01h & \text{otherwise} \end{cases} \tag{3}$$

Having tried all the combinations of activation functions on the perceptron, ReLU gave the average accuracy of 48% and Leaky ReLU: 67%. The usage of Relu and Leaky Relu derivative gave the highest 67.7% accuracy on the test data, and this mix of activation functions is used for all the experiments.

4) Loss Function: Mean Squared Error (MSE) is used as a loss function in this report [12]:

$$E = \frac{1}{2N} \sum_{i=1}^{N} (y_i - x_i)^2 \tag{4}$$

5) Computing Updates: To find a set of weights and biases which minimise the cost function, the Mini-Batch Gradient Descent algorithm is used. It computes the gradient vector of E to update weights and biases [12]:

$$w\prime_{ij}^{(k)} = w_{ij}^{(k)} + \eta \frac{\partial E}{\partial w_{ij}^{(k)}} \tag{5}$$

$$b\prime_{i}^{(k)} = b_{i}^{(k)} + \eta \frac{\partial E}{\partial b_{i}^{(k)}} \tag{6}$$

Where η is the learning rate which controls how much the model is changed in response to the estimated error during every weights update [6]. In all the experiments, η is 0.05.

In order to compute the parameters updates, the derivative of Leaky ReLU needs to be defined [1]:

$$f'(h) = \begin{cases} 1 & \text{if } h > 0 \\ 0.01 & \text{otherwise} \end{cases} \tag{7}$$

The number of times the model parameters are updated depends on two variables. One of them is the number of epochs which states how many times during training the network will iterate thorough the complete training data set [3]. In the experiments, it is equal to 250, which enables the neural network to converge. The second variable is the batch size which defines how many samples the neural network will proceed before updating the weights and biases. If the batch size b is equal to the size of training set (N), the learning algorithm is called Batch Gradient Descent. It calculates the error and the gradient the most accurately, but it is quite slow. If $b = 1$, the algorithm is called Stochastic Gradient Descent (Online Update). It treats accuracy over time - error and gradient are inaccurate, but it converges quickly. Otherwise ($1 < b < N$), the model parameters are updated by Mini-Batch Gradient Descent [3]. Neural network implemented in this report uses the last algorithm with the batch size of 50, which means that the model parameters are updated every 50 samples.

III. Results and Discussion

This section summarises and draws the conclusions from all the experiments. First, the data set is used to train the perceptron with an input layer of size 784 and output layer of size 26. Next, the performance of multi-layer neural network and possible improvements to it (in form of L1 regularisation) are discussed. Finally, the set of experiments is conducted to find the size of the network which classifies the data samples from EMINST with the highest accuracy.

A. Single Layer Network

When training a multi-layer neural network, it is always useful to have a baseline performance to compare to [12]. That is why the data has first been used to train a perceptron. To ensure that the model has converged and it achieved the best (or close) possible performance, the average weight update matrix A_n has been implemented:

$$A_n = \begin{cases} \Delta w_1 & \text{for } n = 0 \\ A_{n-1}(1-\tau) + \tau \Delta w_n & \text{for } n > 1 \end{cases} \tag{8}$$

It computes the average weight update per epoch using an exponential moving average. Since in one epoch, the weights are updated multiple times (every 50 samples), the weight update has been averaged from all mini-batches. Figure 2 shows the sum of all the elements of the average matrix against the number of epochs.

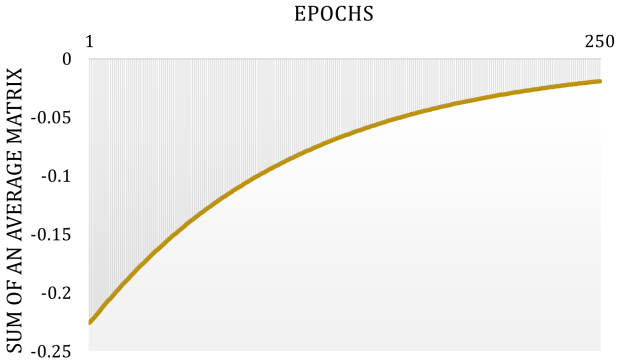

Fig. 2. Sum of the Average Matrix over the Epochs

At Epoch 250, that sum is equal to -0.01914, which means that the model has converged. The classification accuracy of that perceptron on the test set is around 67.7% with the error being around 0.258. The accuracy of Linear Classifier on the Letter EMNIST data set achieved by Cohen et al. is 55.78% [8].

There are a few reasons why the perceptron described in this report outperformed the data set authors' implementation. First of all, this model has been trained and tested on the small subset of EMNIST Letter data set - 1000 and 250 samples per class respectively, while the full data set contains 6000 samples per class for training and 6000 samples per class for testing. Another reason may be the implementation of the perceptron - mainly the choice of the activation function, learning rate or weights initialisation method.

B. Multi-Layer Network with one hidden layer

When training a multi-layer network, the overfitting problem has been observed. The model worked well on the training data (error around 0.11), but performed poorly on the test data (error of 0.198). The reason for this very common in ML problem is explained really well in a famous quote of John von Neumann: "With four parameters I can fit an elephant, and with five I can make him wiggle his trunk" [10]. Since the model has many parameters, it is able to learn the training data too well - it treats the noise or random fluctuations in the data as the concepts which need to be learnt. However, since these concepts do not fit the test data, the general performance of the system is worse [16].

One simple way to deal with overfitting is to use validation data to compute the model accuracy at the end of each epoch. Once the accuracy starts saturating, the training is stopped. Using validation set prevents overfitting on the test data. Other ways to deal with this problem include increasing the amount of training data, decreasing the size of the network (thus decreasing the number of model parameters), or using a dropout method which modifies the network itself by temporality deleting some neurons [19].

However, all these techniques fail with a fixed-size network and fixed data set available. In this case, the best solution is to use regularization techniques. The main idea behind them is penalizing the weight matrices of the nodes. The most common algorithms are L1 and L2 regularisation. They both update the loss function by adding the regularisation term [16]. L2 modifies the loss function as follows [19]:

$$E(\mu) = E_{MSE}(\mu) + \frac{\lambda}{N}\sum \mid w \mid^2 \tag{9}$$

While L1:

$$E(\mu) = E_{MSE}(\mu) + \frac{\lambda}{N}\sum \mid w \mid \tag{10}$$

Both methods penalize large weights. However, L1 shrinks them by a constant amount toward 0, while L2 shrink them by the amount which is proportional to w. Hence L2 decreases the big weights much more and small weights much less than L1. Another key difference is that L1 eliminates the insignificant features so it works like a feature selector [18].

In this report, L1 regularization is implemented. To do it, the cost function needs to be derived with respect to w:

$$\frac{\partial E(\mu)}{\partial w} = \frac{\partial E_{MSE}(\mu)}{\partial w} + \frac{\lambda}{N}\frac{\partial \mid w \mid}{\partial w} \tag{11}$$

Next, the derivative of the absolute value needs to be computed [19]:

$$\mid x \mid' = \frac{1}{2\sqrt{x^2}} * 2x = \frac{x}{\sqrt{x^2}} = \frac{x}{\mid x \mid} \tag{12}$$

It is worth noticing that the derivative of the absolute value is a signum function:

$$signum(x) = \begin{cases} 1 & \text{for } x > 0 \\ -1 & \text{otherwise} \end{cases} \tag{13}$$

With L1 implementation, the weight update looks like this:

$$w\prime_{ij}^{(k)} = w_{ij}^{(k)} + \eta\frac{\partial E}{\partial w_{ij}^{(k)}} - \eta\frac{\lambda}{N}signum(x) \tag{14}$$

The impact of L1 regularization depends on the penalty strength λ. Set of experiments has been conducted to find the optimal value of this parameter. Every value has been tested 5 times. In Figure 3 the average final error and accuracy of the neural network on the validation set have been plotted against the values of λ.

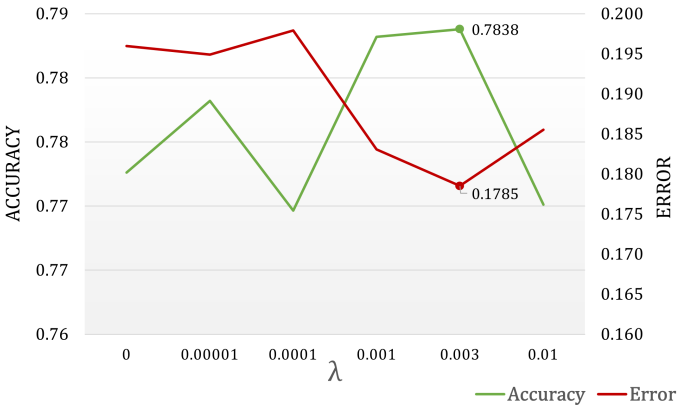

Fig. 3. Validation Error and Accuracy against penalty strength λ

All the values except 0.00001 significantly aided the learning process. The value of 0.003 resulted in the best network performance and it is used in the following experiments.

Next set of experiments was conducted to find the optimal number of neurons in the single hidden layer. Figure 4 shows the performance and error on the test data

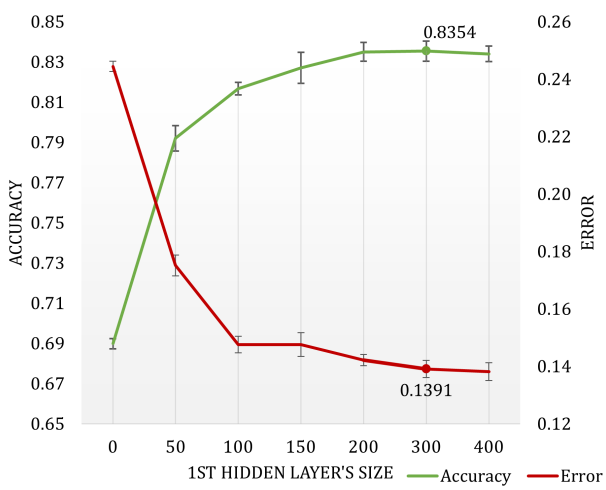

Fig. 4. Test Error and Accuracy against 1st hidden layer's size

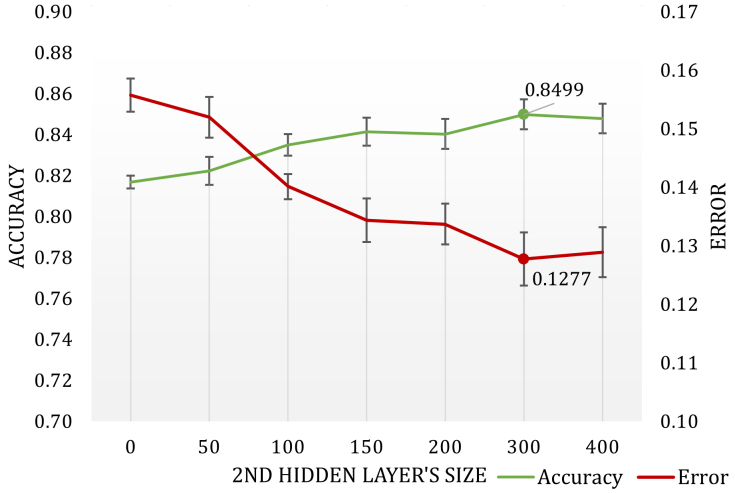

Fig. 5. Test Error and Accuracy against 2nd hidden layer's size

averaged over 10 runs against the numbers of neurons in the hidden layer.

The best performance (83.54%) was achieved with the hidden layer's size of 300. However, with the size of 200, neural network reached the accuracy of 83.51% and it worked much faster than with the bigger layer, thus $[784, 200, 26]$ should be considered as the optimal size of 3-layer network.

Cohen at al. achieved the accuracy above 80% only with hidden layer bigger or equal to 4000 and they beat the accuracy of 83.5% with the hidden layer size of 6000 and bigger. The best performance they achieved (with size 10000) was $85.15\% \pm 0.12\%$ [8]. Because of the lack of computer power, training the neural network of a similar size was not possible. However, the results achieved by smaller networks are comparable with those described in the paper [8].

C. Multi-Layer Network with two hidden layers

The goal of the last experiment was to find the optimal size of the second hidden layer with the first hidden layer of size 100. Figure 5 shows the performance and error on the test data averaged over 10 runs against the numbers of neurons in the second hidden layer.

The best average performance (84.99%) was achieved by the network with the second hidden layer of size 300. In one of the runs, the model classified 85.41% of test data correctly which outperforms the best performance of the neural network with 1 hidden layer developed by Cohen at al. [8].

It is worth noticing that the running time of training the neural network of size $[784, 100, 300, 26]$ was faster than training the neural network of size $[784, 300, 26]$. This might happen because the batch matrix operations on big matrices like 784×300 are much more resource-consuming than working on matrices of sizes 784×100 and 100×300.

IV. Conclusions

The paper described the details of supervised learning process and the implementation of that algorithm on the neural network with ReLU activation function. The ultimate goal was to find the optimal neural network size for classifying EMNIST data set.

The best performance was achieved on the neural network of size $[784, 100, 300, 26]$ with λ equal to 0.003. The accuracy of this system is similar to the best result achieved by the authors in "EMNIST: an extension of MNIST to handwritten letters" [8].

Based on the experiments, it can be derived that for this task the network with more smaller layers is better than the one with 1 larger layer in respect to both the accuracy and the running time.

Acknowledgement

The code has been based on the material from Lab1 and Lab2 of COM3240 Adaptive Intelligence created by Dr Matthew Ellis [12]. 2 methods has been inspired by the code described by Michael Nielsen in the book "Neural Networks and Deep Learning" and wrote in Python3 by Michal Daniel Dobrzanski [19], [11].

REFERENCES

[1] Commonly used activation functions. In *CS231n Convolutional Neural Networks for Visual Recognition*. Stanford University, 2020. Online Course.

[2] BONETTO, R., AND LATZKO, V. Chapter 8 - machine learning. In *Computing in Communication Networks*, F. H. Fitzek, F. Granelli, and P. Seeling, Eds. Academic Press, 2020, pp. 135–167.

[3] BROWNLEE, J. Difference between a batch and an epoch in a neural network, 2019. Online, Accessed: 12- Mar- 2020.

[4] BROWNLEE, J. A gentle introduction to the rectified linear unit (relu), 2020. Online, Accessed: 11- Mar- 2020.

[5] BROWNLEE, J. How to fix the vanishing gradients problem using the relu, 2020. Online, Accessed: 11- Mar- 2020.

[6] BROWNLEE, J. Understand the impact of learning rate on neural network performance, 2020. Online, Accessed: 12- Mar- 2020.

[7] BROWNLEE, J. Why initialize a neural network with random weights?, 2020. Online, Accessed: 10- Mar- 2020.

[8] COHEN, G., AFSHAR, S., TAPSON, J., AND VAN SCHAIK, A. Emnist: an extension of mnist to handwritten letters, 2017.

[9] COHEN, G., AFSHAR, S., TAPSON, J., AND VAN SCHAIK, A. The emnist dataset, 2017. Online, Accessed: 10- Mar- 2020.

[10] COOK, J. How to fit an elephant, 2011. Online, Accessed: 11- Mar- 2020.

[11] DOBRZANSKI, M. D. Deeplearningpython, 2021. Online, Accessed: 28- Feb- 2020.

[12] ELLIS, M. Lecture 1.2: Categories of machine learning. In *COM3420 Adaptive Intelligence*. University of Sheffield, 2021. Course.

[13] GLOROT, X., BORDES, A., AND BENGIO, Y. Deep sparse rectifier neural networks. vol. 15.

[14] HAN, J., KAMBER, M., AND PEI, J. 1 - introduction. In *Data Mining (Third Edition)*, third edition ed., The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Boston, 2012, pp. 1–38.

[15] IBM. Supervised learning, 2020. Online, Accessed: 10- Mar- 2020.

[16] JAIN, S. An overview of regularization techniques in deep learning (with python code), 2018. Online, Accessed: 11- Mar- 2020.

[17] JAITLEY, U. Why data normalization is necessary for machine learning models, 2018. Online, Accessed: 10- Mar- 2020.

[18] NAGPAL, A. L1 and l2 regularization methods, 2017. Online, Accessed: 11- Mar- 2020.

[19] NIELSEN, M. Neural networks and deep learning, 2019. Online, Accessed: 28- Feb- 2020.

[20] VERSLOOT, C. He/xavier initialization & activation functions: choose wisely, 2019. Online, Accessed: 10- Mar- 2020.

[21] WALKER, T. How does rectilinear activation function solve the vanishing gradient problem in neural networks?, 2019. Online, Accessed: 12- Mar- 2020.

APPENDIX A - RESULTS

A. Detailed results to task 6

λ	Error	Accuracy
0	0.1938	0.7665
	0.1986	0.7715
	0.1987	0.7642
	0.1960	0.7806
	0.1928	0.7802
0.00001	0.1925	0.7838
	0.1932	0.7804
	0.1970	0.7729
	0.1926	0.7806
	0.1993	0.7733
0.0001	0.1922	0.7781
	0.1990	0.7633
	0.1974	0.7754
	0.1926	0.7777
	0.2084	0.7538
0.001	0.1806	0.7944
	0.1865	0.7750
	0.1851	0.7838
	0.1790	0.7819
	0.1843	0.7808
0.003	0.1821	0.7829
	0.1680	0.8035
	0.1777	0.7794
	0.1842	0.7725
	0.1806	0.7808
0.01	0.1844	0.7733
	0.1884	0.7640
	0.1882	0.7704
	0.1823	0.7790
	0.1844	0.7638

B. Detailed results to task 7

1st hidden layer	Error	Accuracy
0	0.2433	0.6898
	0.2434	0.6929
	0.2465	0.6894
	0.2464	0.6862
	0.2429	0.6914
50	0.1710	0.7988
	0.1732	0.7954
	0.1738	0.7954
	0.1798	0.7854
	0.1780	0.7854
100	0.1577	0.8135
	0.1589	0.8155
	0.1545	0.8148
	0.1516	0.8206
	0.1558	0.8197
150	0.1487	0.8258
	0.1520	0.8178
	0.1480	0.8275
	0.1486	0.8249
	0.1408	0.8392
200	0.1457	0.8315
	0.1419	0.8377
	0.1415	0.8349
	0.1414	0.8298
	0.1408	0.8414
300	0.1385	0.8414
	0.1398	0.8303
	0.1352	0.8371
	0.1384	0.8383
	0.1436	0.8302
400	0.1375	0.8363
	0.1329	0.8392
	0.1342	0.8362
	0.1413	0.8231
	0.1389	0.8318

C. Detailed results to task 8

2nd layer	Error	Accuracy
0	0.1577	0.8135
	0.1589	0.8155
	0.1545	0.8148
	0.1516	0.8206
	0.1558	0.8197
50	0.1499	0.8257
	0.1505	0.8294
	0.1577	0.8126
	0.1527	0.8180
	0.1489	0.8258
100	0.1399	0.8345
	0.1381	0.8408
	0.1400	0.8360
	0.1390	0.8372
	0.1437	0.8265
150	0.1385	0.8331
	0.1365	0.8397
	0.1332	0.8414
	0.1348	0.8406
	0.1287	0.8520
200	0.1310	0.8463
	0.1297	0.8483
	0.1385	0.8300
	0.1340	0.8374
	0.1354	0.8394
300	0.1248	0.8542
	0.1327	0.8442
	0.1297	0.8431
	0.1214	0.8606
	0.1302	0.8474
400	0.1292	0.8446
	0.1293	0.8531
	0.1291	0.8454
	0.1285	0.8482
	0.1283	0.8480

APPENDIX B - CODE

D. Code structure and how to reproduce the results?

The neural network has been implemented as a separate class inside the $neural\_network.py$ file. $test.py$ shows how to initialise it and how to repeat the experiments described above. To run the experiments described in different tasks in the assignment brief, please uncomment the appropriate piece of code. Code from both files is shown below.

```
#!/usr/bin/env python
""" Developed as part of Assignment 1 of COM3240.
The code implements Neural Network of varying size.
It can be used as perceptron or as multi-layer feed-forward network
depending on the invoked methods.
The code has been based on the material from Lab1 and Lab2
of COM3240 Adaptive Intelligence created by Dr Matthew Ellis.
2 methods has been inspired by the code described by Michael Nielsen
in the book "Neural Networks and Deep Learning"
and wrote in Python3 by Michal Daniel Dobrzanski.
 """

import math
import matplotlib.pyplot as plt
import numpy as np;
import numpy.matlib
import random

__author__ = "Aleksandra Kulbaka"
__credits__ = ["Dr Matthew Ellis", "Grant Sanderson", "Michael Nielsen", "Michal Daniel Dobrzanski"]
__version__ = "1.0.1"
__email__ = "awkulbaka1@sheffield.ac.uk"

class NeuralNetwork(object):

    def __init__(self, sizes, train_data, test_data, n_epoch, batch_size, eta, lambd, regularisation = False):
        """sizes - contains the number of neurons in the respective layers of the network,
        train and test data - lists of tuples of letters in form: (pixels, desired vector output),
        number of epochs, batch size, training rate eta, lambda,
        boolean value indicating applying or skipping L1 regularisation."""

        self.sizes = sizes
        self.train_data = train_data
        self.test_data = test_data
        self.n_epoch = n_epoch
        self.batch_size = batch_size
        self.eta = eta
        self.lambd = lambd
        self.regularisation = regularisation

        # generate random sets of weights and biases
        self.biases = [np.random.randn(y,) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) * np.sqrt(1/x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

        # shuffle train data and take 20% of it as a validate set
        random.shuffle(self.train_data)
        self.validate_data = self.train_data[:5200]
        self.train_data = self.train_data[5200:]

        self.n_train_samples = len(self.train_data)
        self.n_batches = int(math.ceil(self.n_train_samples/batch_size))

        # initialize lists to store training accuracy and error for each epoch
        self.errors = np.zeros((n_epoch,))
        self.accuracy = np.zeros((n_epoch,))

        # upzip train_data into array of pixels and array of desired vector outputs
        self.inputs_train, self.labels_train = zip(*self.train_data)


    def train_multilayer(self, layers, test, test_name):
        """ layers - number of layers in the neural network,
        test - set of data to test the accuracy: validate_data or test_data,
        test_name - name of the validation set"""
```

```
    for epoch in range(self.n_epoch):
        # divide the training data into 784 x 50 matrices representing mini batches
        mini_batches = [self.inputs_train[b : b + self.batch_size] for b in range(0, self.n_train_samples, self.batch_size)]
        desired_labels = [self.labels_train[b : b + self.batch_size] for b in range(0, self.n_train_samples, self.batch_size)]

        # x0: matrix of 50 initial inputs in the batch
        for index, x0 in enumerate(mini_batches):
            x0 = np.transpose(x0)

            # initialise matrices for storing biases and weights updates computed by the backpropagation
            biases_update = [np.zeros(b.shape) for b in self.biases]
            weights_update = [np.zeros(w.shape) for w in self.weights]

            # x1: input layer -> hidden layer,
            h1 = np.transpose(np.transpose(np.matmul(self.weights[0], x0)) + self.biases[0])
            x1 = relu(h1)

            # x2: hidden layer -> output layer or hidden layer1 -> hidden layer2 (for 2 hidden layers)
            h2 = np.transpose(np.transpose(np.matmul(self.weights[1], x1)) + self.biases[1])
            x2 = relu(h2)

            if layers == 4:
                # hidden layer2 -> output layer
                h3 = np.transpose(np.transpose(np.matmul(self.weights[2], x2)) + self.biases[2])
                x3 = relu(h3)

                error_signal = np.transpose(desired_labels[index]) - x3
            else:
                error_signal = np.transpose(desired_labels[index]) - x2

            if self.regularisation:
                # compute the L1 regularisation penalty
                L1_error = [self.lambd * np.sum(np.sqrt(np.square(weight))) / self.n_train_samples for weight in self.weights]
                L1_error = np.sum(L1_error)

                self.errors[epoch] = self.errors[epoch] + (0.5 * np.sum(np.square(error_signal)) / self.n_train_samples) + L1_error
            else:
                self.errors[epoch] = self.errors[epoch] + 0.5 * np.sum(np.square(error_signal)) / self.n_train_samples

            # backpropagation
            if layers == 4:
                # output layer -> hidden layer2
                delta3 = relu_dev(h3) * error_signal
                weights_update[2] =  np.matmul(delta3, np.transpose(x2))
                biases_update[2] = np.sum(delta3, axis=1)

                # hidden layer2 -> hidden layer1
                delta2 = relu_dev(h2) * np.matmul(self.weights[2].T, delta3)
                weights_update[1] = np.matmul(delta2, np.transpose(x1))
                biases_update[1] = np.sum(delta2,axis=1)

            else:
                # output layer -> hidden layer1
                delta2 = relu_dev(h2) * error_signal
                weights_update[1] = np.matmul(delta2, np.transpose(x1))
                biases_update[1] = np.sum(delta2,axis=1)

            # hidden layer -> input layer
            delta1 = relu_dev(h1) * np.matmul(self.weights[1].T, delta2)
            weights_update[0] = np.matmul(delta1, np.transpose(x0))
            biases_update[0] = np.sum(delta1,axis=1)

            # updating weights and biases
            if self.regularisation:
                self.weights = [weight + self.eta / self.batch_size * (update - self.lambd * signum(weight))
                for (weight, update) in zip(self.weights, weights_update)]
```

```
            else:
                self.weights = [weight + self.eta * update / self.batch_size
                for (weight, update) in zip(self.weights, weights_update)]

            self.biases = [bias + self.eta * update / self.batch_size
            for (bias, update) in zip(self.biases, biases_update)]

        # get accuracy and error of the testing data
        accuracy, error = self.evaluate(test)
        print( "Epoch {}. On {} data: Error: {}, Accuracy: {}".format(epoch + 1, test_name, error, accuracy))
    return "{} set: Error: {}, Accuracy: {}".format(test_name, error, accuracy)


def train_perceptron(self, test, test_name):
    """ test - set of data to test the accuracy: validate_data or test_data,
    test_name - name of the validation set """

    # initialise average matrix and constant tau
    average_matrix = np.zeros((self.n_epoch,))
    deltas = []
    TAU = 0.01

    for epoch in range(self.n_epoch):
        # divide the training data into 784 x 50 matrices representing mini batches
        mini_batches = [self.inputs_train[b : b + self.batch_size] for b in range(0, self.n_train_samples, self.batch_size)]
        desired_labels = [self.labels_train[b : b + self.batch_size] for b in range(0, self.n_train_samples, self.batch_size)]
        # initialise deltas to store the sum of weights updates
        deltas = [np.zeros(w.shape) for w in self.weights]

        # x0: matrix of 50 initial inputs in the batch
        for index, x0 in enumerate(mini_batches):
            x0 = np.transpose(x0)

            # initialise matrices for storing biases and weights updates computed by the backpropagation
            biases_update = [np.zeros(b.shape) for b in self.biases]
            weights_update = [np.zeros(w.shape) for w in self.weights]

            # x1: input layer -> output layer
            h1 = np.transpose(np.transpose(np.matmul(self.weights[0], x0)) + self.biases[0])
            x1 = relu(h1)

            error_signal = np.transpose(desired_labels[index]) - x1

            if self.regularisation:
                # compute the L1 regularisation penaltyy
                L1_error = [self.lambd * np.sum(np.sqrt(np.square(weight))) / self.n_train_samples for weight in self.weights]
                L1_error = np.sum(L1_error)

                self.errors[epoch] = self.errors[epoch] + (0.5 * np.sum(np.square(error_signal)) / self.n_train_samples) + L1_error
            else:
                self.errors[epoch] = self.errors[epoch] + 0.5 * np.sum(np.square(error_signal)) / self.n_train_samples

            # backpropagation: output layer -> input layer
            delta1 = relu_dev(h1) * error_signal
            weights_update[0] = np.matmul(delta1, np.transpose(x0))
            biases_update[0] = np.sum(delta1,axis=1)

            # updating weights and biases
            if self.regularisation:
                self.weights = [weight + self.eta / self.batch_size * (update - self.lambd * signum(weight))
                for (weight, update) in zip(self.weights, weights_update)]
            else:
                self.weights = [weight + self.eta * update / self.batch_size
                for (weight,update) in zip(self.weights, weights_update)]

            self.biases = [bias + self.eta * update / self.batch_size
```

```
            for (bias, update) in zip(self.biases, biases_update)]

            deltas[0] += (self.eta * weights_update[0] / self.batch_size)

        # compute average matrix
        if epoch == 0:
            average_matrix[epoch] = np.sum(deltas[0]) / self.n_batches
        else:
            average_matrix[epoch] = average_matrix[epoch - 1] * (1 - TAU) + TAU * np.sum(deltas[0] / self.n_batches)

        accuracy, error = self.evaluate(test)
        print( "Epoch {}. Average Matrix: {}.  On {} data: Error: {}, Accuracy: {}".format(epoch + 1,
        average_matrix[epoch], test_name, error, accuracy))

    # plot the average matrix value vs epochs
    plt.plot(average_matrix)
    plt.ylabel('Average Matrix')
    plt.xlabel('Epochs')
    plt.show()
    return "{} set: Error: {}, Accuracy: {}".format(test_name, error, accuracy)


# the code for the following 2 functions was inspired by the code from the book:
# http://neuralnetworksanddeeplearning.com/
# and the following repository:
# https://github.com/MichalDanielDobrzanski/DeepLearningPython
  def feed_forward(self, x):
      """Return the output of the network for x"""
      for b, w in zip(self.biases, self.weights):
          x = relu(np.dot(w, x) + b)
      return x

  def evaluate(self, test):
      """ Take test dataset as an argument
      return the accuracy and errors of neural network on this dataset """
      test_results = [(np.argmax(self.feed_forward(pixels)), np.argmax(label_vec)) for (pixels, label_vec) in test]
      accuracy = sum(int(output == label) for (output, label) in test_results) / len(test)

      errors = [(np.square(self.feed_forward(pixels) - label_vec)) for (pixels, label_vec) in test]
      errors = 0.5 * np.sum(errors) / len(test)

      return accuracy, errors


def relu(x):
  """The relu function."""
  return np.maximum(0, x)
  # return np.where(x > 0, x, 0.01*x)

def relu_dev(x):
  """Derivative of the relu function.
  Return 0.01 if x <= 0 instead of 0 to prevent the dying relu problem"""
  # https://cs231n.github.io/neural-networks-1/#actfun
  return np.where(x > 0, 1, 0.01)

def signum(x):
  """The signum function"""
  return np.where(x > 0, 1, -1)
```

```
#!/usr/bin/env python
""" Developed as part of Assignment 1 of COM3240.
The code tests Neural Network of varying size.
It can be used as perceptron or as multi-layer feed-forward network
depending on the invoked methods.
The code has been based on the material from Lab1 and Lab2
of COM3240 Adaptive Intelligence created by Dr Matthew Ellis.
 """

import csv
import math
import matplotlib.pyplot as plt
import numpy as np;
import numpy.matlib
from scipy.io import loadmat
import neural_network

__author__ = "Aleksandra Kulbaka"
__credits__ = ["Dr Matthew Ellis", "Grant Sanderson", "Michael Nielsen", "Michal Daniel Dobrzanski"]
__version__ = "1.0.1"
__email__ = "awkulbaka1@sheffield.ac.uk"

# read EMNIST train and test datasets
# divide the input in train and test sets by 255 to normalize data so each pixel has value from 0 to 1
emnist = loadmat('emnist-letters-1k')
x_train = emnist['train_images'] / 255
train_labels = emnist['train_labels']
x_test = emnist['test_images'] /255
test_labels = emnist['test_labels']

# dataset contains letter A - Z so 26 different labels
n_labels = 26
# create labels vectors. e.g., label 0 corresponds to a vector [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
y_train = np.zeros((train_labels.shape[0], n_labels))
y_test  = np.zeros((test_labels.shape[0], n_labels))
for i in range(train_labels.shape[0]):
    y_train[i, train_labels[i].astype(int)] = 1
for i in range(test_labels.shape[0]):
    y_test[i, test_labels[i].astype(int)] = 1

# global parameters constant during all the tests: number of epochs, size of batch, learning rate eta
N_EPOCH = 250
BATCH_SIZE = 50
ETA = 0.05


# create list of tuples (pixels, desired output vector)
# train_data as list of tuples is useful for e.g. easy random split of data for training and validation
train_data = list(zip(x_train, y_train))
test_data = list(zip(x_test, y_test))

# parameters changed during experiments: lambda for L1, sizes of hidden layers, parameter indicating applying or skipping L1
regularisation
lambd = 0.003
hidden_1 = 100
hidden_2 = 100
if_L1 = True

# create NeuralNetwork object by giving arguments: layers' sizes,
# tuples of train and test data, number of epochs, batch size, eta,
# lambda and boolean value indicating applying or skipping L1 regularisation
# net = neural_network.NeuralNetwork([784, hidden_1, hidden_2, 26], train_data, test_data, N_EPOCH, BATCH_SIZE, ETA,
lambd, if_L1)
# call train_multilayer method with arguments: size of neural network,
# test set (test_data or validate_data) and name of the test data
# print(net.train_multilayer(4, net.test_data, "Test "))
```

```
# Below there is a code for all the experiments I've done in my report. Uncomment to make it work.

# # TASK 3, 4: single layer perceptron with average weight update matrix, on a test data
# if_L1 = False
# net = neural_network.NeuralNetwork([784, 26], train_data, test_data, N_EPOCH, BATCH_SIZE, ETA, lambd, if_L1)
# print(net.train_perceptron(net.test_data, "Test "))


# # TASK 6:
# hidden_1 = 50
# if_L1 = True
# # vary lambda
# lambd = 0.003

# # creates or add to the existing file the result of 5 runs of neural network
# with open("task6.txt", "a") as f:
#     f.write("Lambda value = {}".format(lambd))
#     f.write("\n")
#     for i in range(5):
#         net = neural_network.NeuralNetwork([784, hidden_1, 26], train_data, test_data, N_EPOCH, BATCH_SIZE, ETA, lambd,
if_L1)
#         result = net.train_multilayer(3, net.validate_data, "Validation ") + "\n"
#         f.write(result)
#         f.write("\n")
#         print(result)

# # TASK 7:
# lambd = 0.003
# if_L1 = True
# # vary hidden layer's size
# hidden_1 = 50

# # creates or add to the existing file the result of 10 runs of neural network
# with open("task7.txt", "a") as f:
#     f.write("1st Hidden layer = {}".format(hidden_1))
#     f.write("\n")
#     for i in range(10):
#         net = neural_network.NeuralNetwork([784, hidden_1, 26], train_data, test_data, N_EPOCH, BATCH_SIZE, ETA, lambd,
if_L1)
#         result = net.train_multilayer(3, net.test_data, "Test ") + "\n"
#         f.write(result)
#         f.write("\n")
#         print(result)

# # TASK 8:
# lambd = 0.003
# hidden_1 = 100
# if_L1 = True
# #vary second hidden layer's size
# hidden_2 = 100

# # creates or add to the existing file the result of 10 runs of neural network
# with open("task8.txt", "a") as f:
#     f.write("2nd hidden layer = {}".format(hidden_2))
#     f.write("\n")
#     for i in range(10):
#         net = neural_network.NeuralNetwork([784, hidden_1, hidden_2, 26], train_data, test_data, N_EPOCH, BATCH_SIZE, ETA,
lambd, if_L1)
#         result = net.train_multilayer(4, net.test_data, "Test ") + "\n"
#         f.write(result)
#         f.write("\n")
#         print(result)
```