Homework 5

CS 559 - Neural Networks - Fall 2019

Matteo Corain 650088272

November 4, 2019

1 Question 1

1.1 Data parsing and preprocessing

Data have been parsed from the images and label files using a variation of the functions defined in the second homework for this purpose. Specifically:

- Function `idx_images_parse()` is used to parse the images files encoded in the IDX format; it performs the following actions:
  - It opens the given file in *read binary* mode;
  - It reads the magic number using the `fromfile()` function provided by the Numpy library, which allows to read a certain number of elements (in this case, a single one) from a file given their data type (in this case, big endian `>`, unsigned integer `u` on four bytes `4`) and checks its correctness (it must be equal to 2051);
  - In the same way, it reads the number of items and the dimensions of the input images into variables `n_samples`, `n_rows` and `n_cols`;
  - Using the list comprehension syntax and the `fromfile()` function provided by the Numpy library, it reads `n_samples` arrays of `n_rows * n_cols` elements stored as big endian, unsigned 8-bits integers (`>u1`), reshapes them to the correct format and stores them into the `samples` variable;
  - It returns variables `n_samples`, `n_rows`, `n_cols` and `samples`.
- Function `idx_labels_parse()` is used to parse the labels files encoded in the IDX format; it performs similar actions with respect to the previous one, the only differences being that in this case samples dimensions are not present and that data items are represented by single 8-bits elements.

With respect to the functions used for the second homework, some preprocessing is done on the data before feeding them to the network as training samples. In particular, the following preprocessing has been applied:

- Input pixel values have been normalized by subtracting them their mean value and by dividing each component by 255, so that in the end they are distributed in range $[-1, 1]$ (they are stored as 8-bits unsigned integers) with zero mean;
- Labels have been mapped on their binary vector representation, using the `label_to_array()` function from homework 2.

Scaling the inputs allows for avoiding the need of training large weights for the network, while label mapping is useful since it is not necessary to reconstruct the binary representation each time it is needed (it is computationally easier to extract the argmax, when necessary).

1.2 Hyperparameters selection

For the purpose of the exercise, the networks to be tested have been chosen according to the following parameters:

- *Network topology*: as for the network topology, all the considered networks are either 2-layers or 3-layers. The choice of not using more than 3 layers has been dictated both by the computational complexity required for the training of multiple layers and by the fact that, in any case, a 2-layers network is already proven to be capable of approximating any function with arbitrary precision (although the usage of multiple-layers networks may result in better training performance).
- *Digit representation*: classified digits have been represented as in homework 2; for this reason, all the tested networks present 10 neurons in the output layer, and the class prediction is obtained by extracting the argmax of the network output. This representation is handy because it allows also to assign a form of "confidence score" to the different predictions that the network returns (i.e. the network classifies the input pattern as i with a y_i confidence).
- *Activation functions*: as for activation functions, it has been chosen to utilize hyperbolic tangents for the hidden layers and sigmoid for the output layer. Hyperbolic tangents have been chosen for their well-known properties that make them suitable for the usage in neural networks (e.g. they are odd functions), while sigmoids have been used in order to saturate the network output in a positive range between 0 and 1, allowing to interpretate results as a sort of "probabilistic" confidence score (although they do not sum to 1).
- *Learning rates*: for the sake of simplicity, all neurons in the networks make use of the same learning rate η, initially set to $\eta = 0.01$. A dynamic update mechanism of the learning rate has also been implemented, decrementing the value of η by a factor of 0.9 each time it comes across an increase of the error function from an epoch to the following.
- *Energy function*: the mean squared error measure has been chosen as the energy function for all the tested networks, in which the error is evaluated in terms of distance from the binary representation of the expected label:

$$E = \frac{1}{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} ||d_i - y_{i,L}||^2$$

- Other implementation heuristics:
  - *Stochastic update*: online learning has been used for all training tasks;
  - *Inputs normalization*: inputs have been normalized to avoid saturation as previously described;
  - *Weights initialization*: weight matrices have been initialized using random values drawn from a normal distribution with $\mu = 0$ and $\sigma^2 = m^{-1}$, where m represents the number of inputs of each specific layer.

1.3 Network construction and training

In order to properly represent the different networks, a `Network` class has been defined. Its constructor takes as arguments three lists, the first one containing the initial weights matrices (biases included) of each layer of the network, the second containing the function pointers corresponding to the activation functions of each layer of the network and the third the function pointers of the derivatives thereof. The class presents the following methods:

- `feedforward()`: it feeds the network with the input $x = y_0$, computing for each layer the corresponding value of the local field v and of the output y:

$$v_i = W_i \begin{bmatrix} 1 \\ y_{i-1} \end{bmatrix}$$
$$y_i = \phi_i(v_i)$$

- `backpropagate()`: it estimates the value of the gradient of the energy function ∇E through the backpropagation algorithm, for an input pattern x with desired output d; this is performed in three steps:
  - In the first step, the v and y vectors of the induced fields and outputs of each neuron are computed through the `feedforward()` method;
  - In the second step, the delta signals are calculated, in reverse order of i:

$$\delta_L = (d - y_L) \cdot \phi'_L(v_L)$$
$$\delta_i = \underline{W_{i+1}^T} \delta_{i+1} \cdot \phi'_i(v_i), i = L-1, \ldots, 1$$

    Where the underlining denotes the weight matrix in which the first row (the one related to the bias term) has been removed.
  - In the third step, delta signals are used to compute the effective components of the desired gradient:

$$\frac{\partial E}{\partial W_i} = -\delta_i \begin{bmatrix} 1 \\ y_{i-1} \end{bmatrix}$$

- `error()`: it calculates the value of the error function for the entire training set (mean squared error), averaging the squared value of the norm of the difference between the desired outputs and the effective outputs (computed through the `feedforward()` method):

$$E = \frac{1}{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} ||d_i - y_{i,L}||^2$$

- `test()`: it calculates the accuracy of the network on the test set, by summing a value of 1 each time the classification is correct (that is, $\operatorname{argmax} y_{i,L} = \operatorname{argmax} d_i$) and dividing by the number of test samples:

$$A = \frac{1}{|\mathcal{T}|} \sum_{i=1}^{|\mathcal{T}|} \alpha_i, \ \alpha_i = \begin{cases} 1 & \text{if } \operatorname{argmax} y_{i,L} = \operatorname{argmax} d_i \\ 0 & \text{otherwise} \end{cases}$$

- `train()`: it performs a gradient descent, backpropagation-based training of the network; it takes as parameters:
  - The training samples and labels;
  - The test samples and labels (to keep track of the test set accuracy for increasing epochs);
  - The initial learning rate η;
  - The target value of the error function ϵ;
  - The maximum number of training epochs;
  - A string suffix to be used for storing computed quantities in appropriate files for later usage (e.g. plotting) without having to fully recompute them.

  The procedure performs the following actions:
  - It initializes the `errors`, `trainacc` and `testacc` lists with the initial values of the error function and of the training and test sets accuracy;
  - It enters the training loop, whose stopping conditions are expressed on the number of epochs (it has to be higher than the configured limit) and on the value of the error function (it has to be lower than the configured limit);
  - It loops on the training samples, computing for each of them the value of ∇E through the `backpropagate()` method, then updates the weight matrices of the different layers using the standard gradient descent rule;
  - It registers the value of the error function and the accuracy on the training and test sets for the current weights;
  - If the value of the error function increases among two consecutive epochs, it reduces the learning rate of a factor of 0.9;
  - When the training loop is computed, the final weights and the lists of errors and accuracy values are saved to file and returned.

Algorithm 1 Feed-forward procedure

function FEED-FORWARD($network, x_i$)

▷ Initialization

$v \leftarrow$ EMPTY-LIST
$y \leftarrow$ EMPTY-LIST
$y[0] \leftarrow x_i$

▷ Computation of v and y values

for $i = 1, \ldots, L$ **do**
$v[i] \leftarrow network.W[i] \begin{bmatrix} 1 & y[i] \end{bmatrix}^T$
$y[i+1] \leftarrow network.\phi[i](v[i])$
end for
return v, y
end function

Algorithm 2 Backpropagation procedure

function BACKPROPAGATE($network, x_i, d_i$)

▷ Initialization

$v, y \leftarrow network.$FEED-FORWARD(x_i)
$\delta \leftarrow$ EMPTY-LIST
$\nabla E \leftarrow$ EMPTY-LIST

▷ Computation of δ_i values

$\delta[L] \leftarrow (d_i - y[L]) * network.\phi'[L](v[L])$
for $i = L-1, \ldots, 1$ **do**
$\delta[i] \leftarrow \left(network.\underline{W[i+1]}\right)^T \delta[i+1] * network.\phi'[i](v[i])$
end for

▷ Computation of ∇E components

for $i = 1, \ldots, L$ **do**
$\nabla E[i] = -\delta[i] \begin{bmatrix} 1 & y_{i-1} \end{bmatrix}^T$
end for
return v, y
end function

Algorithm 3 Error procedure

function ERROR($network, x, d$)
$e \leftarrow 0$
for $i = 1, \ldots, n$ **do**
$v, y \leftarrow network.$FEED-FORWARD$(x[i])$
$e \leftarrow e + (d[i] - y[L])^2$
end for
return e/n
end function

Algorithm 4 Test procedure

```
function TEST(network, x, d)
    c ← 0
    for i = 1, ..., n do
        v, y ← network.FEED-FORWARD(x[i])
        if argmax d[i] = argmax y[L] then
            c ← c + 1
        end if
    end for
    return c/n
end function
```

Algorithm 5 Training procedure

```
function TRAIN(network, x_train, d_train, x_test, d_test, η, ϵ, λ)
                                                        ▷ Initialization
    epochs ← 0
    errors ← EMPTY-LIST
    trainacc ← EMPTY-LIST
    testacc ← EMPTY-LIST
    errors[0] ← network.ERROR(x_train, d_train)
    trainacc[0] ← network.ERROR(x_train, d_train)
    testacc[0] ← network.ERROR(x_test, d_test)
                                                        ▷ Main weights update loop
    while epochs < λ ∧ errors[epochs] > ϵ do
        epochs ← epochs + 1
                                                        ▷ Compute update terms and update weights
        for i = 1, ..., n do
            ∇E ← network.BACKPROPAGATE(x_train[i], d_train[i])
            for j = 1, ..., L do
                network.W[j] ← network.W[j] − η∇E[j]
            end for
        end for
                                                        ▷ Error and accuracy registration
        errors[epochs] ← network.ERROR(x_train, d_train)
        trainacc[epochs] ← network.ERROR(x_train, d_train)
        testacc[epochs] ← network.ERROR(x_test, d_test)
                                                        ▷ Update of η
        if errors[epochs] > errors[epochs − 1] then
            η ← 0.9η
        end if
    end while
    return epochs, errors, trainacc, testacc
end function
```

1.4 Tested networks and results

For the sake of the exercise, six networks have been tested, whose main parameters are presented in table 1.

Network #	Layers	n_1	n_2	n_3	ϕ_1	ϕ_2	ϕ_3	η	ϵ	Epoch limit
1	2	10	10	-	tanh	sigmoid	-	0.01	0.01	100
2	2	100	10	-	tanh	sigmoid	-	0.01	0.01	100
3	2	200	10	-	tanh	sigmoid	-	0.01	0.01	100
4	3	10	10	10	tanh	tanh	sigmoid	0.01	0.01	100
5	3	100	10	10	tanh	tanh	sigmoid	0.01	0.01	100
6	3	100	100	10	tanh	tanh	sigmoid	0.01	0.01	100

Table 1: Parameters of the tested networks

For the selection of the number of neurons for each layer of each network, it has been chosen to start from the network used for this classification task in homework 2. In that case, we made use of a single-layer network with 10 neurons, capable of achieving an accuracy of about 85% when trained using the whole training set.

For this reason, a first try was performed, using a 2-layers network having the same number of neurons (10) for both the hidden and the output layer. Despite such network has not been able to converge to the configured maximum error value within the configured number of epochs, the training performances of such network already show a sensible improvement over the network used in homework 2 (93.79% accuracy for the training set, 91.79% accuracy for the test set).

Given that the precision of the approximation of a 2-layer network may be improved by introducing additional neurons, it has then been chosen to test two additional 2-layers networks, having respectively 100 and 200 neurons in their hidden layer. These networks performed much better, reaching convergence in 63 and 43 epochs respectively with around 99.5% training set accuracy and almost 98% test set accuracy. The downside for that has been the significant increase in training time, especially for the network with 200 elements in the hidden layer.

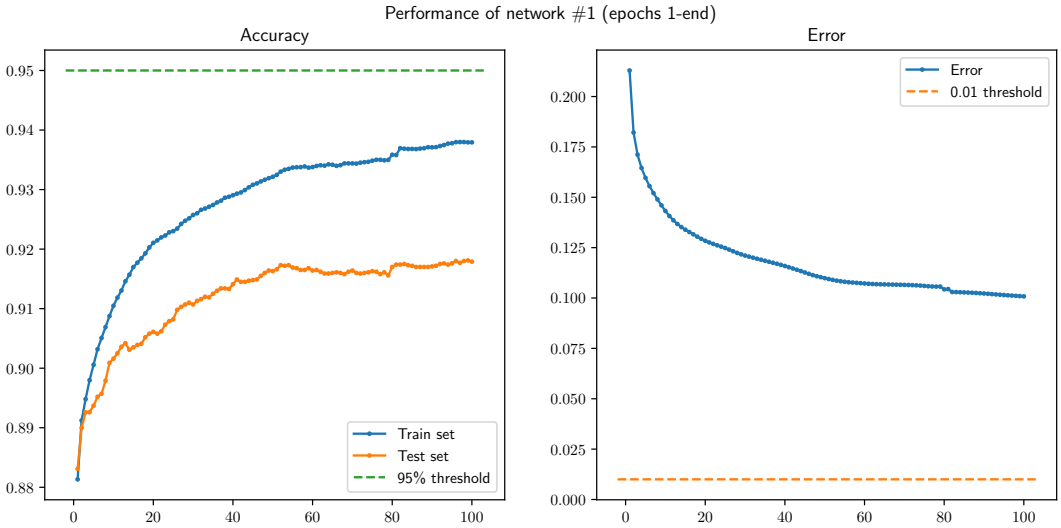
For the 3-layers network, a similar process has been followed. Initially, it has been chosen to test the performance of a network having 10 neurons in both hidden layers, the training of which again reached the configured epoch limit; as expected, the performance of such network are in any case better than the ones of the initial 2-layers network due to the increased abstraction capabilities introduced by the additional hidden layer, but still not in line with the 95% test set accuracy target (95.44% for the training set, 92.87% for the test set).

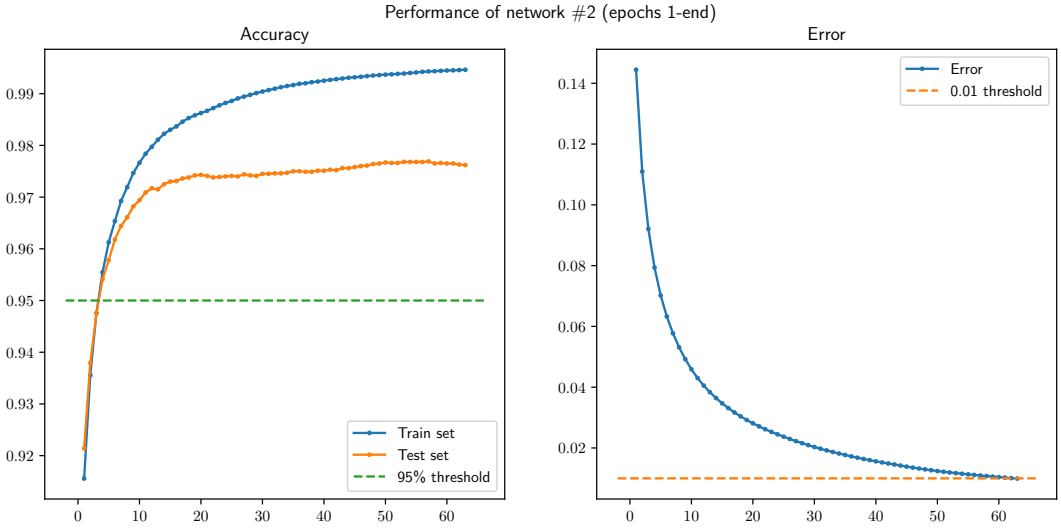
Again, two additional 3-layers networks have been tested, increasing the number of neurons to 100 respectively in the first and in both hidden layers. Those networks were the ones that yielded the best performances in terms of training epochs and time overall, reaching accuracy levels comparable to the ones of 2-layers networks, but reducing the number of training epochs by half and the training time by three to four times.

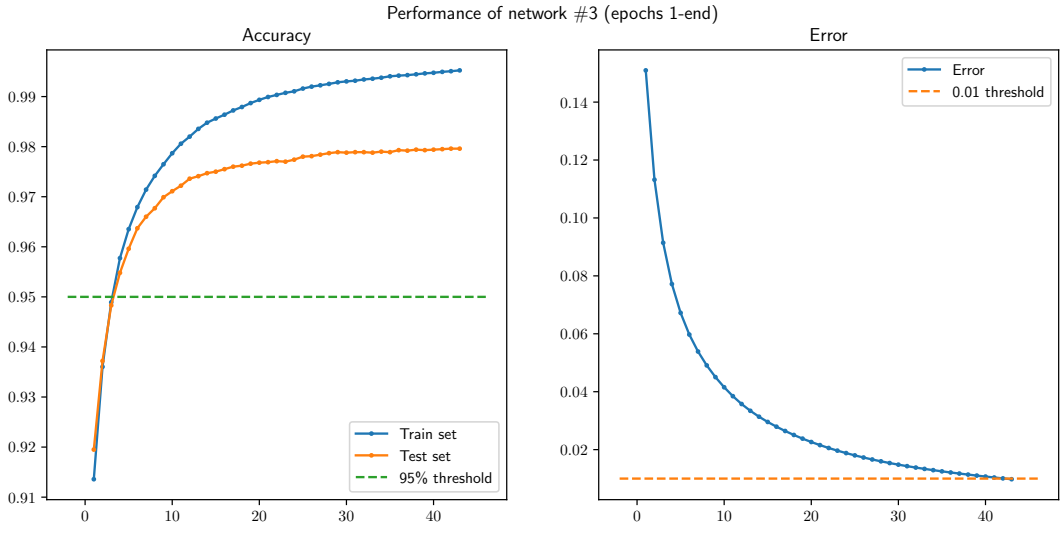
A summary of the results of the training process are shown in table 2. The overall best network, in terms of ratio between accuracy and training time is the fifth one, although the sixth converges in a more limited number of epochs. The figures in the following pages show the value of the error function and the accuracy for increasing number of epochs for the considered networks.

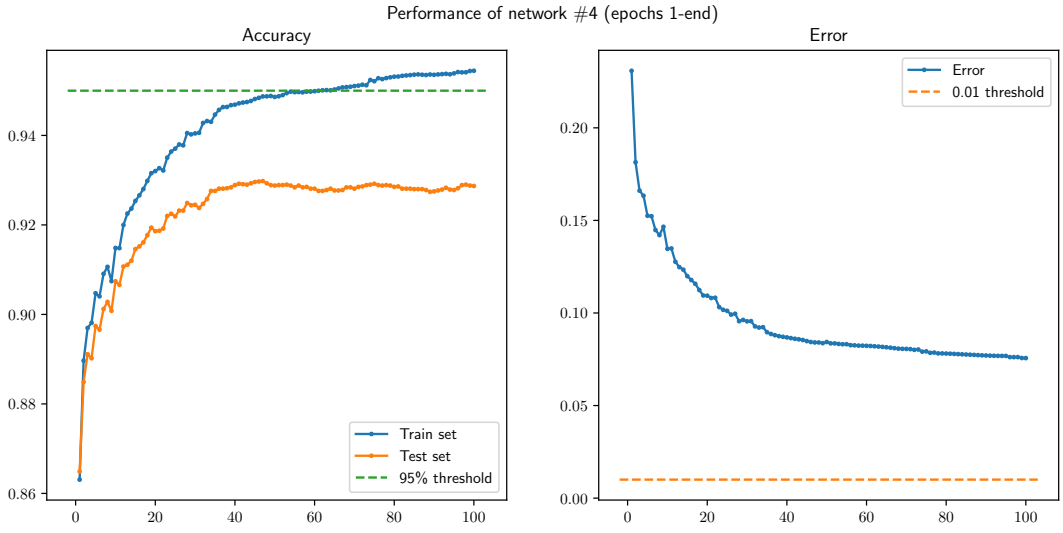
Network #	Epochs	Training time (s)	Training set accuracy	Test set accuracy
1	100+	913	93.79%	91.79%
2	63	3486	99.46%	97.62%
3	43	7704	99.54%	97.90%
4	100+	1130	95.44%	92.87%
5	29	1366	99.48%	97.54%
6	24	2303	99.50%	98.00%

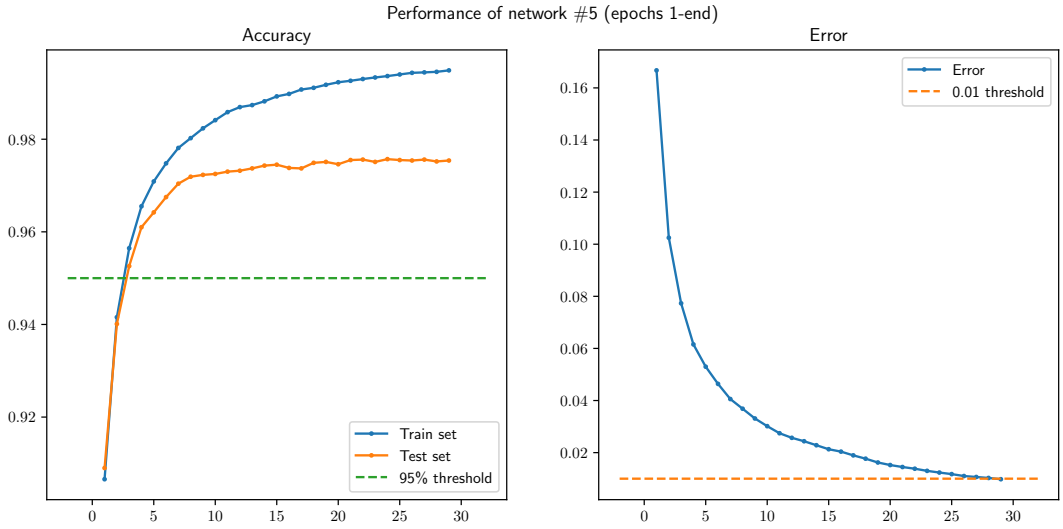
Table 2: Training results for the tested networks

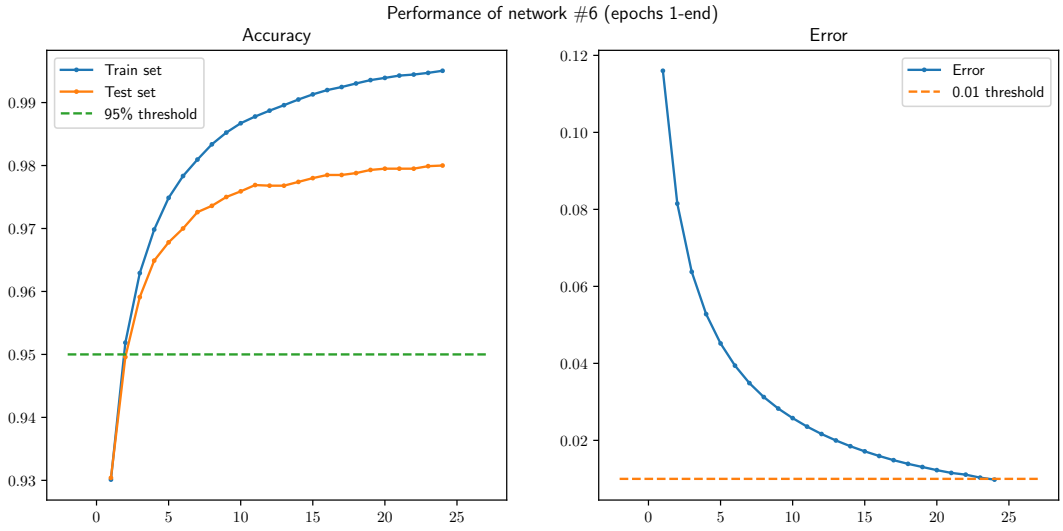





Performance of network #3 (epochs 1-end)
Accuracy
Error
Train set
Test set
95% threshold
Error
0.01 threshold


Performance of network #4 (epochs 1-end)
Accuracy
Error
Train set
Test set
95% threshold
Error
0.01 threshold


Performance of network #5 (epochs 1-end)
Accuracy
Error
Train set
Test set
95% threshold
Error
0.01 threshold


Performance of network #6 (epochs 1-end)
Accuracy
Error
Train set
Test set
95% threshold
Error
0.01 threshold

1.5 Complete Python code

1.5.1 Data parsing functions

```
import numpy as np

def label_to_array(label):
    # Transform a label into a binary array
    return np.array([1 if i == label else 0 for i in range(10)]).reshape(10, 1)

def idx_images_parse(filename):
    with open(filename, "rb") as f:
        # Check magic number
        if np.fromfile(f, dtype=">u4", count=1)[0] != 2051:
            return False, False, False, False

        # Read number of samples, rows and columns
        n_samples = np.fromfile(f, dtype=">u4", count=1)[0]
        n_rows = np.fromfile(f, dtype=">u4", count=1)[0]
        n_cols = np.fromfile(f, dtype=">u4", count=1)[0]

        # Parse samples
        samples = [np.fromfile(f, dtype=">u1", count=n_rows*n_cols).reshape(n_rows * n_cols, 1) for i in range(
            n_samples)]
        samples = [(s - np.mean(s)) / 255 for s in samples]

    return n_samples, n_rows, n_cols, samples

def idx_labels_parse(filename):
    with open(filename, "rb") as f:
        # Check magic number
        if np.fromfile(f, dtype=">u4", count=1)[0] != 2049:
            return False, False

        # Read number of labels
        n_labels = np.fromfile(f, dtype=">u4", count=1)[0]

        # Parse labels
        labels = [label_to_array(np.fromfile(f, dtype=">u1", count=1)[0]) for i in range(n_labels)]

    return n_labels, labels
```

1.5.2 Network class definition

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

class Network:
    def __init__(self, weights, actfuncs, derfuncs):
        self.weights = weights
        self.actfuncs = actfuncs
        self.derfuncs = derfuncs

    def feedforward(self, x):
        v = []
        y = [x.reshape(len(x), 1)]

        # Compute local fields and outputs
        for i in range(len(self.weights)):
            v.append(self.weights[i] @ np.concatenate(([1], y[i]), axis=None).reshape(len(y[i]) + 1, 1))
            y.append(self.actfuncs[i](v[i]).reshape(len(v[i]), 1))
```

```
        return v, y

    def backpropagate(self, x, d):
        # Compute local fields and outputs
        v, y = self.feedforward(x)
        delta = [0] * len(self.weights)
        dEW = [0] * len(self.weights)

        # Compute delta signals
        delta[len(self.weights) - 1] = (d - y[len(self.weights)]) * self.derfuncs[len(self.weights) - 1](v[len(
            self.weights) - 1])
        for i in reversed(range(len(self.weights) - 1)):
            delta[i] = self.weights[i + 1].transpose()[1:] @ delta[i + 1] * self.derfuncs[i](v[i])

        # Compute the gradient
        for i in range(len(self.weights)):
            dEW[i] = -delta[i] @ np.concatenate(([1], y[i]), axis=None).reshape(1, len(y[i]) + 1)

        return dEW

    def error(self, data, labels):
        # Compute the value of the error function
        return sum([(np.linalg.norm(labels[i] - self.feedforward(data[i])[1][-1])) ** 2 for i in range(len(data)
            )]) / len(data)

    def train(self, train_data, train_labels, test_data, test_labels, eta, eps, epoch_limit, save_suffix):
        # Run the gradient descent method
        epochs = 0

        errors = [self.error(train_data, train_labels)]
        training_accuracy = [self.test(train_data, train_labels)]
        test_accuracy = [self.test(test_data, test_labels)]
        print("Epoch {}: eta = {}, err = {}, train = {}, test = {}".format(epochs, eta, errors[epochs],
            training_accuracy[epochs], test_accuracy[epochs]))

        while epochs < epoch_limit and errors[epochs] >= eps:
            # Increment epochs
            epochs = epochs + 1

            # Update weights
            for i in range(len(train_data)):
                dEW = self.backpropagate(train_data[i], train_labels[i])
                for j in range(len(dEW)):
                    self.weights[j] = self.weights[j] - eta * dEW[j]

            # Register the error
            errors.append(self.error(train_data, train_labels))
            training_accuracy.append(self.test(train_data, train_labels))
            test_accuracy.append(self.test(test_data, test_labels))
            print("Epoch {}: eta = {}, err = {}, train = {}, test = {}".format(epochs, eta, errors[epochs],
                training_accuracy[epochs], test_accuracy[epochs]))

            # Decrease eta if necessary
            if errors[epochs] > errors[epochs - 1]:
                eta = 0.9 * eta

        if save_suffix is not None:
            # Save weights and errors to file
            #np.save("weights_{}.npy".format(save_suffix), self.weights)
            np.save("errors_{}.npy".format(save_suffix), errors)
            np.save("trainacc_{}.npy".format(save_suffix), training_accuracy)
            np.save("testacc_{}.npy".format(save_suffix), test_accuracy)
```

```
        return epochs, errors, training_accuracy, test_accuracy

    def test(self, test_data, test_labels):
        # Count classification errors on the test set
        return sum([
            1 if np.argmax(test_labels[i]) == np.argmax(self.feedforward(test_data[i])[1][-1])
            else 0
            for i in range(len(test_data))
        ]) / len(test_data)
```

1.5.3 Training of the networks

```
from sys import argv
from time import time
import numpy as np

from network import Network
from idxfuncs import idx_images_parse, idx_labels_parse

# Set the random seed for reproducibility
np.random.seed(2019)

# Activation functions
phi1 = lambda v : np.tanh(v)
der_phi1 = lambda v : 1 - np.tanh(v) ** 2
phi2 = lambda v : 1 / (1 + np.exp(-v))
der_phi2 = lambda v : np.exp(-v) / (1 + np.exp(-v)) ** 2

# Parse the training set
n_train_samples, n_rows, n_cols, train_samples = idx_images_parse("../hw02/train-images.idx3-ubyte")
_, train_labels = idx_labels_parse("../hw02/train-labels.idx1-ubyte")
print("Training set loaded.")

# Parse the test set
n_test_samples, _, _, test_samples = idx_images_parse("../hw02/t10k-images.idx3-ubyte")
_, test_labels = idx_labels_parse("../hw02/t10k-labels.idx1-ubyte")
print("Test set loaded.")

# Parse number of neurons from command line
neurons = [n_rows * n_cols] + [int(argv[i]) for i in range(1, len(argv))] + [10]

# Define standard deviations for each layer
sigma = [np.sqrt(1 / (neurons[i] + 1)) for i in range(len(neurons) - 1)]

# Initialize weights for each layer
W = [np.random.normal(0, sigma[i], size=(neurons[i + 1], neurons[i] + 1)) for i in range(len(sigma))]

# Define activation functions for each layer
phi = [phi1] * (len(W) - 1) + [phi2]
der_phi = [der_phi1] * (len(W) - 1) + [der_phi2]

# Create the network
net = Network(W, phi, der_phi)

# Train the network
eta = 0.01
eps = 0.01
epoch_limit = 100

start_time = time()
epochs, errors, training_accuracy, test_accuracy = net.train(train_samples, train_labels, test_samples,
    test_labels, eta, eps, epoch_limit, " ".join(f"{n}" for n in neurons))
print(f"Elapsed time: {time() - start_time}")
```