Homework 2

CS 559 - Neural Networks - Fall 2019

Matteo Corain 650088272

September 25, 2019

1 Question 1

1.1 General architecture of the network

In order to design a network capable of implementing the proposed separation, it is possible to proceed as follows:

- First, we need to identify the inequalities that describe the two regions and implement them using a set of perceptrons;
- Then, since the inequalities for the two regions should be all satisfied at once, we need to connect the outputs of all perceptrons useful for describing a region using a perceptron implementing a logical AND;
- Finally, since the output of the network should be positive in case the pattern belongs to any of the two regions, we need to connect the outputs of the AND perceptrons using a perceptron implementing a logical OR.

Let us first begin to identify the linear relationships that are necessary to identify the colored regions. Using the nomenclature shown in figure 1, we have:

$$l_1 : -1 - x + y = 0, \quad l_2 : -1 + x + y = 0$$
$$l_3 : -1 + x = 0, \quad l_4 : 2 + x = 0, \quad l_5 : y = 0$$

Given this nomenclature, the two regions are described by the following set of inequalities:

$$R_1 = \begin{cases} l_1 \leq 0 \\ l_2 \geq 0 \\ l_3 \leq 0 \end{cases}, \quad R_2 = \begin{cases} l_1 \geq 0 \\ l_4 > 0 \\ l_5 < 0 \end{cases}$$

As it can be seen, some of these inequalities do not present the equality sign; since the perceptrons we are designing use the step activation function (which returns 1 when the pattern is on the separator), this means that those cannot be implemented by a single perceptron. Instead, we have

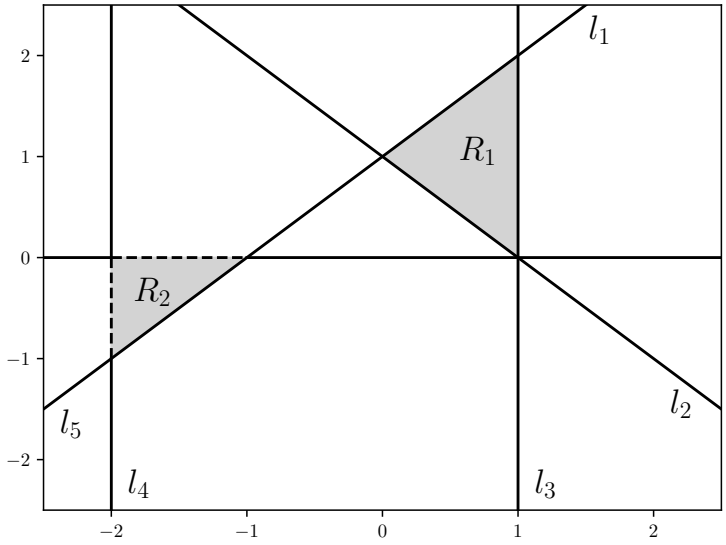

Figure 1: Nomenclature used for lines and regions

to make use of a succession of two perceptrons, the first implementing the complimentary inequality and the second implementing a logical NOT:

$$l_4 > 0 \Rightarrow \neg(l_4 \leq 0), \quad l_5 < 0 \Rightarrow \neg(l_5 \geq 0)$$

The block diagram of the network to implement is shown in figure 2; each block represents a perceptron to be designed. As we can see, the network is composed of four distinct layers:

- The first layer is the *line layer*, and includes the neurons used to implement the separation described by the linear inequalities;
- The second layer is the *NOT layer*, and includes the neurons used to implement the NOT operation as requested by inequalities not including the equality sign; for inputs that do not need the usage of a NOT perceptron, we can imagine that those are connected to a buffer perceptron, simply returning as output the input value;
- The third layer is the *AND layer*, and includes the neurons used to implement the AND operation among inequalities useful to describe the same region (all should be satisfied at the same time);
- The fourth layer is the *OR layer*, and includes the neuron used to implement the OR operation among the results of the previous layer (we want a positive result if the pattern is either in R_1 or in R_2).

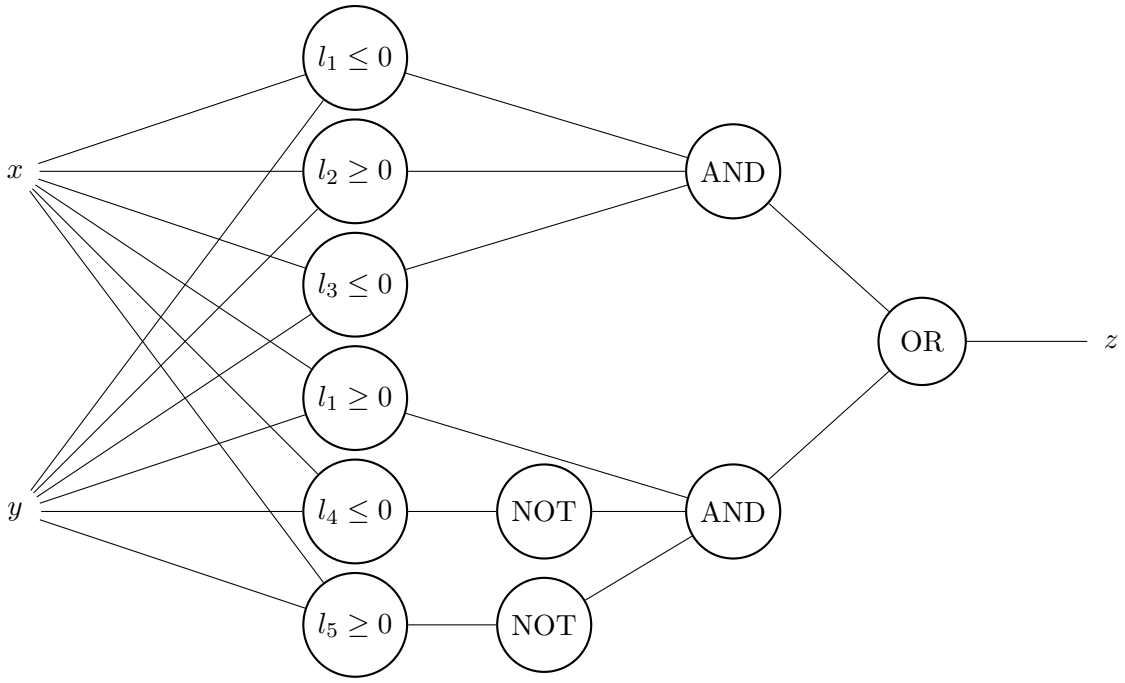

Figure 2: Block diagram of the requested network

1.2 Line neurons

Line neurons constitute the first layer of the network; they receive two inputs, namely the x and y inputs of the network, and return a binary output that is positive when the pattern satisfies the associated inequality. In order to design them correctly, we have to recall what kind of class separation a perceptron with step activation in general implements; for the case with two inputs x and y, this is expressed in the normal form:

$$w_0 + w_1 x + w_2 y \geq 0$$

Therefore, if we can write the inequalities that describe the regions for which we want a positive output in a form that matches the normal one, it is possible to derive the vector of weights w of the perceptron directly from the inequality itself. Using this procedure, the following results can be obtained:

- Neuron #1: for the first neuron, the inequality to implement is $-1 - x + y \leq 0$, which is, in normal form, $1 + x - y \geq 0$; thus, a possible choice of weights for this neuron is:

$$w_1 = [1 \quad 1 \quad -1]$$

- Neuron #2: for the second neuron, the inequality to implement is $-1 + x + y \geq 0$, already in normal form; thus, a possible choice of weights for this neuron is:

$$w_2 = [-1 \quad 1 \quad 1]$$

- Neuron #3: for the third neuron, the inequality to implement is $-1 + x \leq 0$, which is, in normal form, $1 - x \geq 0$; thus, a possible choice of weights for this neuron is:

$$w_3 = [1 \quad -1 \quad 0]$$

- Neuron #4: for the fourth neuron, the inequality to implement is $-1 - x - y \geq 0$, already in normal form; thus, a possible choice of weights for this neuron is:

$$w_4 = [-1 \quad -1 \quad 1]$$

- Neuron #5: for the fifth neuron, the inequality to implement is $2 + x \leq 0$, which is, in normal form, $-2 - x \geq 0$; thus, a possible choice of weights for this neuron is:

$$w_5 = [-2 \quad -1 \quad 0]$$

- Neuron #6: for the sixth neuron, the inequality to implement is $y \geq 0$, already in normal form; thus, a possible choice of weights for this neuron is:

$$w_6 = [0 \quad 0 \quad 1]$$

1.3 NOT neurons

NOT neurons constitute the second layer of the network; they receive a single input, namely the output of a line neuron, and return the logical inverse of that input. If we suppose that the network topology has to be complete, i.e. each neuron is connected to the output of each neuron in the previous layer, then it is possible to suppose that the weights relative to the unused inputs are set to 0. They can be implemented using the standard set of weights:

$$w = [0.5 \quad -1]$$

Not all line perceptrons need to be negated for implementing the requested network. If we want to build a complete layer of neurons, we can imagine to introduce a set of "buffer" neurons, which simply return as output the received input; these can be implemented using the following set of weights that is opposite to the one of the NOT neurons:

$$w = [-0.5 \quad 1]$$

1.4 AND neurons

The two AND neurons constitute the third layer of the network; they both receive three inputs, namely the outputs of the line neurons (possibly negated) that describe a single region, and return a binary output that is positive when the pattern is located inside the region (all inequalities are satisfied at once). They can be implemented using the standard design procedure:

- The bias w_0 is set to $-n_i + 0.5$, where n_i is the number of inputs of the perceptron;

- Weights $w_1, \ldots, w_{n_i}$ are set to 1.

In this case, therefore, both AND neurons may use the following set of weights:

$$w = [-2.5 \quad 1 \quad 1 \quad 1]$$

Also in this case, if we assume that the network topology has to be complete, weights corresponding to unused inputs may be set to 0.

1.5 OR neuron

The single OR neuron constitute the fourth and last layer of the network; it receives two inputs, namely the outputs of the two AND neurons, and returns a binary output that is positive when the pattern is located inside one of the two regions. It can be implemented using the standard design procedure:

- The bias w_0 is set to -0.5;
- Weights $w_1, \ldots, w_{n_i}$ are set to 1.

In this case, therefore, the OR neuron may use the following set of weights:

$$w = [-0.5 \quad 1 \quad 1]$$

1.6 Network testing

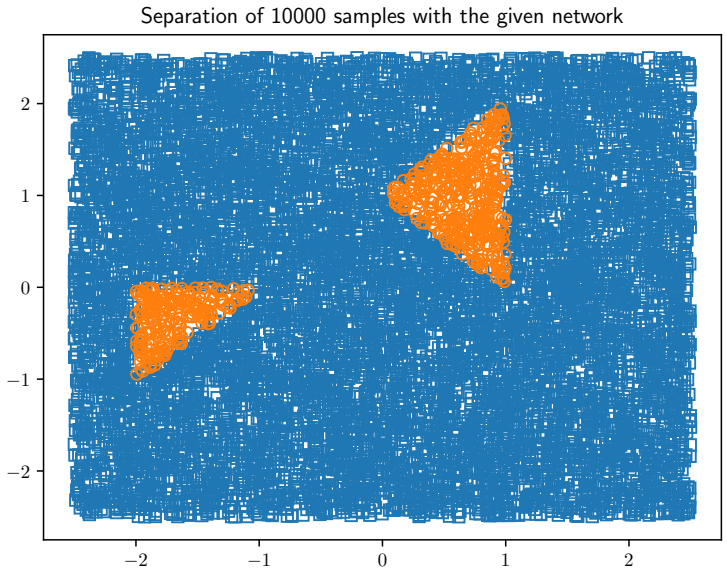

Figure 3: Separation of 10000 samples with the given network

2 Question 2

2.1 Parsing of the input files

The first step to take for the sake of the exercise is to parse the input files, encoded using the binary IDX format, into Numpy arrays that can be used to train the network using the multicategory PTA. As described in the provided website, those files are structured according to the following scheme:

- The first four bytes represent the *magic number*, which has the following meaning:
  - The first two bytes are identically set to 0;
  - The third byte identifies the type of the data stored in the file, which is in this case *unsigned integer* (associated to the hexadecimal value 0x08);
  - The fourth byte identifies the *number of dimensions* of the stored data, which is set to 1 for the labels files (single vector containing all the labels) and to 3 for the images files (vector containing a list of 2-dimensional matrices).

  In this case, the magic number is therefore set to decimal value 2051 for images files and to decimal value 2049 for labels files.
- The following four bytes tell how many items are present in the collection; this is set to 60000 for training files and to 10000 for test files;
- In the images files, the following pair of 4-bytes integers explicit the width and height of the samples (in this case, they are 28 pixels by 28 pixels);
- The following bytes represent the data to be parsed.

All numeric values are stored using the *big endian* format, which means that the most significant bytes are stored at lower offset values. In order to parse the images and labels files, two separate functions have been coded:

- 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 provided by the Python language 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.

2.2 Running the multicategory PTA algorithm

The multicategory Perceptron Training Algorithm has been coded as a separate function, called `multcat_pta()`. It takes as arguments the list of training samples (`train_samples`), the list of training labels (`train_labels`), the initial weights matrix (`start_weights`), the number of samples to use (`n`), the value of η (`eta`), the value of ϵ (`eps`) and the maximum number of epochs for which the algorithm can run (`max_epochs`); it returns the computed weights, the number of epochs needed and the number of classification errors per epoch. This function performs the following actions:

- It creates a copy of the initial weights into the `weights` matrix and initializes the `epochs` variable to 0;

- It allocates the `errors` list, which is initialized with the number of classification errors obtained using the initial weights; those are computed by using the `count_errors()` function, which performs the sum of the elements of a list (created using the list comprehension syntax) that are set to 1 when the computed label is different from the expected label (extracted from the vector $v = Wx_i$ by means of the `array_to_label()` function, which simply returns the index of the component with the maximum value by means of the `argmax()` function of the Numpy library);

- It loops until either the proportion of misclassified samples falls below the given value of ϵ or the maximum number of epochs has been reached; at each iteration, the following actions are taken:

  - The number of epochs is incremented;
  - The weights matrix is updated once for each training sample, by adding to it the quantity $\eta(d(x_i) - u(Wx_i))x_i^T$; vector $d(x_i)$ is created using `label_to_array()`, which creates a binary array having as its only component set to 1 the one with the index given by the label, while function $u(\cdot)$ has been implemented using the `heaviside()` function of the Numpy library;
  - The number of classification errors obtained using the updated weights, computed using the already described `count_errors()` function, is appended to the `errors` list.

- When the loop terminates, the values of `weights`, `epochs` and `errors` are returned.

The number of classification errors per epoch may then be plotted using the `plot_errors()` function, which creates a Matplotlib figure, plots the data, sets the title of the graph, enables the grid and shows the graph without blocking the execution.

2.3 Results for variable n

For the first part of the exercise, the multicategory PTA has been used to train a network having fixed the values of $\eta = 1$, of $\epsilon = 0$ and of an initial weight matrix W_0 to a set of random values in $[-1, 1]$. The only variable in the three cases has been the number of training samples n that the algorithm had to use, which was set to 50, 1000 and 60000 (the entire set) in the three runs. Results are shown in table 1.

n	Epochs for convergence	Test set errors
50	5	4667 (46.670%)
1000	42	1749 (17.490%)
60000	-	1559 (15.590%)

Table 1: Results for the runs with variable n

The algorithm was able to reach convergence both for $n = 50$ and for $n = 1000$ in a relatively low number of epochs (5 and 42, respectively); this means that the trained network is able to classify correctly all the considered items of the training set (since ϵ is set to 0). However, the performances of the network are very poor, especially for $n = 50$: when the obtained set of weights is used to classify the patterns in the test set, in fact, we get a significant proportion of misclassified items.

This may be justified by the small dimensions of the data sets used for training: the separation obtained via this kind of network, in fact, may be sufficient for the classification of a reduced number of training samples, but it may be quite different from the "desired" one. In other words, the computed sets of weights are able to correctly discriminate the few data points in the training set (that means that the algorithm worked correctly), but they are not good enough to correctly classify test samples (the separation is not optimal).

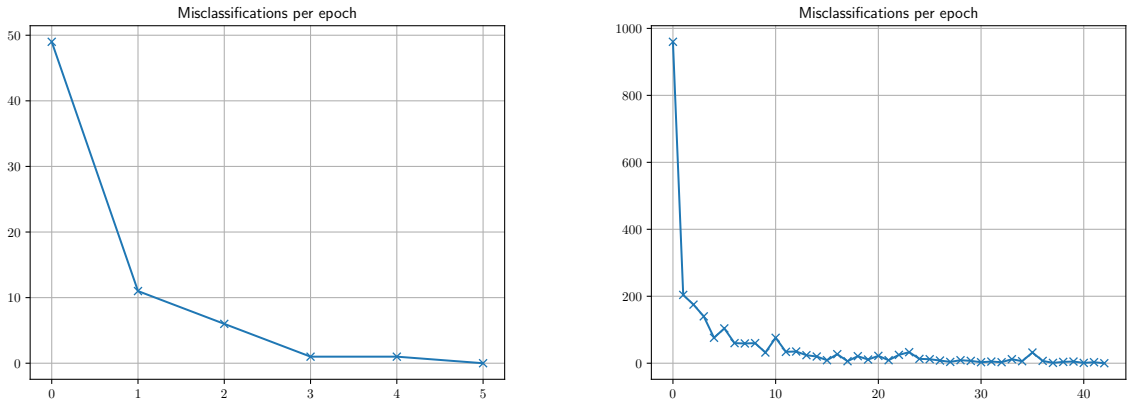

Figure 4: Classification error rate per epoch ($n = 50, 1000$)

In the third run, the network is trained using the entire data set; in this case, the algorithm was not able to converge in a manageable number of epochs (epoch limit was set to 250). This is due to the size of the data set and to the impossibility to linearly separate the classes of interest: in fact,

after the initial epochs, the error rate did not continue to show a decreasing tendency, but instead remained quite stable around 12%-18%. At the time the epoch limit was reached, the network still misclassified 9071 elements of the training set (15.118%).

In this case, we can see that the proportion of misclassified patterns for the training set matches more closely the results found for the test set (15.590%, coming from 1559 misclassifications). This was somehow expected, since we are using all the data we have in the training set: given the greater dimension of the input set, in fact, this measure becomes statistically significant also for the training set (it was not in the previous cases, using a reduced number of samples). In any case, due to the impossibility to linearly separate the samples, training with the entire data set shows a limited improvement over the case with $n = 1000$, which took considerably less time to run.

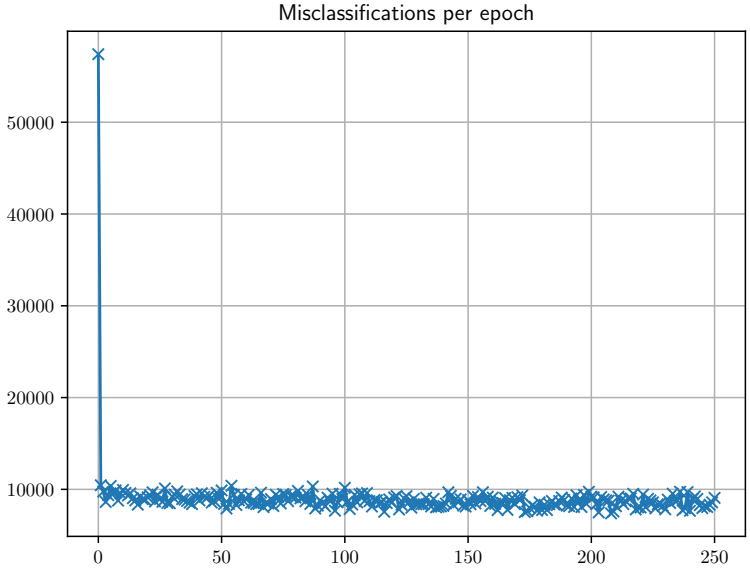

Figure 5: Classification error rate per epoch ($n = 60000$)

2.4 Results for variable W_0

For the last part of the exercise, the training algorithm has been used to train a network after having fixed the values of $n = 60000$ (all training samples used), $\epsilon = 0.125$ (maximum error rate on the training set of 12.5%) and $\eta = 5$. Three runs have been executed, all using different values for the matrix of the initial weights W_0. Results are shown in table 2.

As we can see, results for the three runs are quite close for all parameters except for the number of epochs required to make the algorithm converge. In fact, the Perceptron Training Algorithm is very dependent on the choice of initial weights; even in the cases in which it guarantees convergence (in case of linear separability of the classes), it may take more or less epochs to reach it. In the presented case, the second run was the one that required the least number of epochs to reach convergence at $\epsilon = 12.5\%$ (53).

Run	Epochs for convergence	Training set errors	Test set errors
1	114	7486 (12.477%)	1315 (13.150%)
2	53	7387 (12.312%)	1293 (12.930%)
3	182	7271 (12.118%)	1283 (12.830%)

Table 2: Results for the runs with variable W_0

As for the classification accuracy, we can see that, as in the previous case, results for training and test sets are quite close, which is expected since we are training the network with all the available training data. Of the three runs, the one that yielded the best results overall was the third, with a test set inaccuracy of 12.83%.

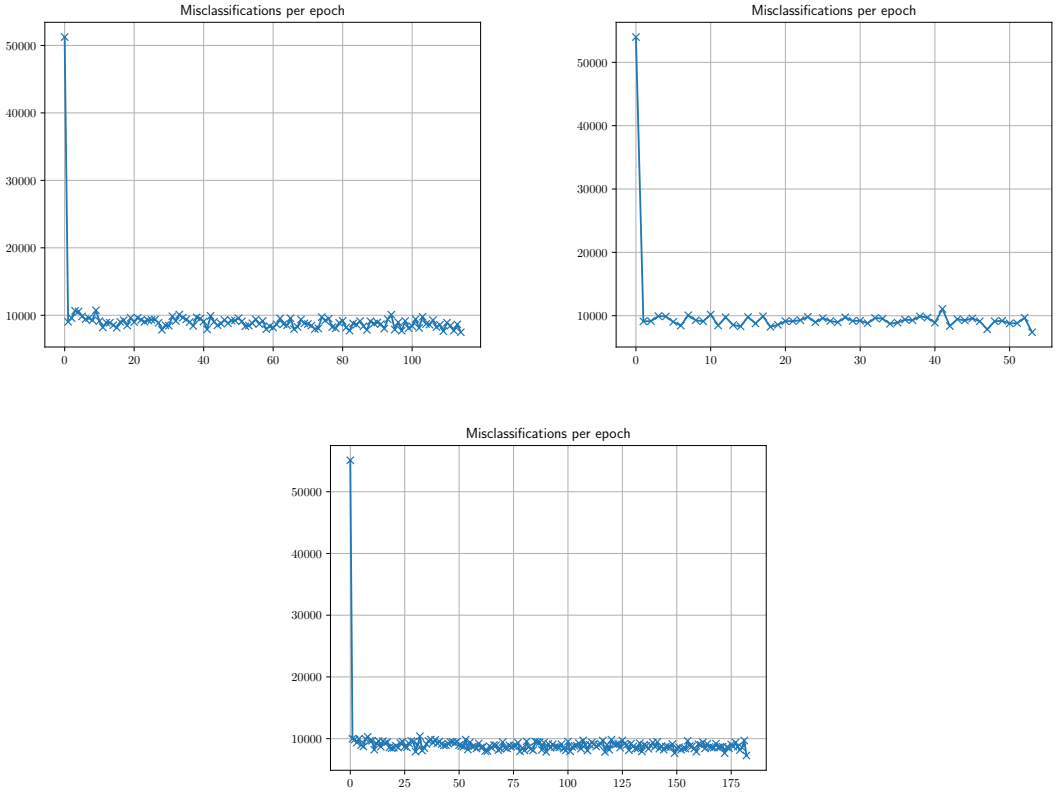

Figure 6: Classification error rate per epoch (variable W_0)

2.5 Complete Python code

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

# Use TeX for text rendering
plt.rc('text', usetex=True)

# Set the random seed for reproducibility
np.random.seed(25)

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 array_to_label(array):
    # Transform a binary array into a label
    return np.argmax(array)

def count_errors(S, n, W, d):
    # Count misclassifications with the given weights
    return sum([1 if array_to_label(W @ S[i]) != d[i] else 0 for i in range(n)])

def multcat_pta(train_samples, train_labels, start_weights, n, eta, eps, max_epochs):
    # Initialize PTA variables
    weights = np.array(start_weights)
    epochs = 0

    # Initialize errors array
    errors = [count_errors(train_samples, n, weights, train_labels)]
    print("Epoch {} errors: {} ({:.03%}).".format(epochs, errors[len(errors)-1], errors[len(errors)-1]/n))

    # Run the multicategory PTA algorithm
    while errors[epochs] / n > eps and epochs < max_epochs:
        epochs = epochs + 1
        for i in range(n):
            weights = weights + eta * (label_to_array(train_labels[i]) -
                np.heaviside(weights @ train_samples[i], 1)) @ train_samples[i].transpose()
        errors.append(count_errors(train_samples, n, weights, train_labels))
        print("Epoch {} errors: {} ({:.03%}).".format(epochs, errors[len(errors)-1], errors[len(errors)-1]/n))

    return weights, epochs, errors

def plot_errors(errors):
    # Create new figure
    plt.figure()

    # Plot data
    plt.plot(errors, marker="x", fillstyle="none")

    # Set plot options
    plt.title("Misclassifications per epoch")
    plt.grid()

    # Show plot
    plt.show(block=False)

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)]

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

    return n_labels, labels

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

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

# First run: fixed W0, n = 50-1000-60000, eps = 0, eta = 1
W0 = np.random.uniform(-1, 1, size=(10, n_rows*n_cols))
eps = 0
eta = 1
max_epochs = 250

for n in [50, 1000, n_train_samples]:
    print("Running with n = {}, eps = {}, eta = {}.".format(n, eps, eta))

    # Run the multicategory PTA algorithm
    weights, epochs, errors = multcat_pta(train_samples, train_labels, W0, n, eta, eps, max_epochs)
    print("Multicategory PTA finished.")

    # Plot errors per epoch
    plot_errors(errors)

    # Classify test set using computed weights
    test_errors = count_errors(test_samples, n_test_samples, weights, test_labels)
    print("Test set errors: {} ({:.03%}).".format(test_errors, test_errors / n_test_samples))

# Second run: variable W0, n = 60000, eps = 0.125, eta = 1-10-0.1
n = n_train_samples
eps = 0.125
eta = 5
max_epochs = 250

for i in range(3):
    W0 = np.random.uniform(-1, 1, size=(10, n_rows*n_cols))
```

```
    print("Running with n = {}, eps = {}, eta = {}.".format(n, eps, eta))

    # Run the multicategory PTA algorithm
    weights, epochs, errors = multcat_pta(train_samples, train_labels, W0, n, eta, eps, max_epochs)
    print("Multicategory PTA finished.")

    # Plot errors per epoch
    plot_errors(errors)

    # Classify test set using computed weights
    test_errors = count_errors(test_samples, n_test_samples, weights, test_labels)
    print("Test set errors: {} ({:.03%}).".format(test_errors, test_errors / n_test_samples))

# Maintain graphs
plt.show()
```