Neural Networks
Assignment 4 of the Machine Learning 1 course 2022/2023

Nicchairelli Gabriele

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
Università Degli Studi di Genova

1 Introduction

Artificial Neural networks (*ANN*) are inspired by the human brain structure. They are composed of several interconnected units, called *neurons*, capable of a single input-to-output transformation. There are usually several connections, and their properties are summarized in parameters called weights. In a Neural Network, neurons are organized in layers cascaded and connected to each other by patterns.

2 Theory of Neural Networks

A Neural Network is composed of a set of *neurons*. A neuron is a single unit that takes an input, makes a decision, and returns an output. It can receive inputs either from the outside world or from another neuron in the previous layer. Even the output of a neuron can be directed to the outside world or to a neuron in the following layer.
Each neuron is associated with a *weight* (**w**), that changes during the learning process accordingly to predefined rules, defined by the *weight update* procedure:

$$\mathbf{w}_{l+1} = \mathbf{w}_l + \Delta\mathbf{w}_l \tag{1}$$

After defining the weight we can put into formulas the network (*r*) as:

$$r = \mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{d} w_i x_i \tag{2}$$

Where **x** is the input vector of the network.
The output of a neuron is called *activation value* (*a*) and it's given by the following equation:

$$a = f(r - \theta) \tag{3}$$

Where θ is the threshold and *f()* is the *activation function*. The latter can correspond to many functions, some examples are the *Heaviside step, sign, Sigmoid, hyperbolic tangent*.
Now we can introduce the *Perceptron learning algorithm*, which is the simplest type of ANN. In the perception, we choose the sign function as the activation function and, returning to the first equation, we compute $\Delta\mathbf{w}_l$, which is called *correction factor*, as follows:

$$\Delta\mathbf{w}_l = \eta\delta_l\mathbf{x}_l \qquad \delta_l = \frac{1}{2}(t_l - a_l) \tag{4}$$

Where η is a (real positive) coefficient and t_l is the target of x_l. The perceptron rule guarantees convergence as stated by the following theorem:
If the training set is linearly separable, the perceptron learning procedure will find a separating hyperplane for it in a finite number of steps.

2.1 Adding layers to a Neural Network

If we add more layers to the network then we say that the ANN has *multiple layers*. The simplest multi-layer Neural Network is the one with one *input layer*, one inner layer (called *hidden layer*), and one *output layer*.
The input layer takes **x** and passes it to the hidden layer, then the hidden layer spreads its output to the output layer, which produces the output data.
As you can notice the hidden layer will be hidden from the outside of the neural network. The more hidden layers we have, the better the performances. However, if we add more hidden layers we will also require more computational power. Hence it is mandatory to find the correct balance between *performances* and *number of layers*.

2.2 Autoencoder

An *autoencoder* is a special type of multi-layer perceptron with two main characteristics:

- the number of units in the input and the output layers is the same, while the hidden layer must have fewer neurons;
- is trained to approximate the identity function, which means that it replicates the same pattern of the input at the output.

Notice that since we don't use any target, the autoencoder task is *unsupervised*. The interesting output of the autoencoder isn't the output value, which is just a low-quality approximation of the input, but the pattern output of the hidden layer.

3 Assignment

The assignment consists of three tasks:

1. Neural networks in Matlab

2. Feedforward multi-layer networks (multi-layer perceptrons)

3. Autoencoder

3.1 Neural networks in Matlab

In this assignment, we are going to use Matlab along with the *Deep Learning Toolbox*.
The first step was to become familiar with the toolbox following this tutorial.
The toolbox already provided some sample data sets along with ready networks. The sample data sets are already subdivided in:

- *train set*: 70% of the data set;
- *test set*: 15% of the data set;
- *validation set*: 15% of the data set.

The validation set is used to validate if the network is generalizing well and stop training before *overfitting*. If the network suffers from overfitting, it means that it recognizes well what it has already seen, but finds it difficult to recognize data that it has never seen.
We can now train the neural network and examine the result. For example, we can plot the regression, see the performances, and the error histogram.

3.2 Feedforward multi-layer networks

The second task asks to follow another tutorial. This time we want to solve a pattern recognition problem using a two-layers feed-forward network.
After selecting the data set to use we can select the number of hidden neurons (default is 10) and train our network, leaving the other parameters as default. The result of the network can be analyzed by observing the *confusion matrices* and the *ROC* (Receiving Operating Characteristic) graph that the toolbox generates.

3.3 Autoencoder

Finally, we have to implement an autoencoder in Matlab. For this task, we use the *MNIST* data set which is composed of 60.000 handwritten digits (from 0 to 9) stored in a $28 \times 28px$ grayscale image.
We select only two classes (digits) at a time and give them as input to the autoencoder. Since the data set is very large, from the two class sets we choose a portion (5%) of the observations randomly selected.
For the purpose of highlighting the autoencoder performances, some pairs of digits are chosen such that they are easier to discern, while others may be much more difficult:

$$classes = \begin{bmatrix} 1 & 8 \\ 3 & 9 \\ 1 & 7 \\ 2 & 5 \end{bmatrix} \tag{5}$$

The autoencoder is initialized with the functions *trainAutoencoder()* and *encode()*.
trainAutoencoder() requires as inputs the data set and the number of hidden nodes. Note that for using this toolbox we have to transpose our sets since it uses a different convention.
In order to have viewable data, we use 2 hidden nodes.
encode() function is then used to encode the data such that it can be plotted. It requires two arguments, the autoencoder and the data set, and it returns the encoded data.
The last thing to do is to plot the output of the two hidden layers, one per axis, using the *plotcl()* function, provided during the course.
If the points of the two classes are distant from each other, it means that the autoencoder has learned well. Otherwise, if the two sets are mixed, it means that the learning process was more difficult and the two classes aren't clearly distinguishable.

4 Results

The results of the first two tasks are not reported since they are tutorials. The results can be seen in the links to the tutorial. In the following figures, we can see the separation of the pair of classes of the third task.
From the figures below we can observe that the chosen pair are quite difficult to distinguish. In particular, we can observe that the points of each pair are very mixed, so they are almost indistinguishable from the point of view of the encoder.

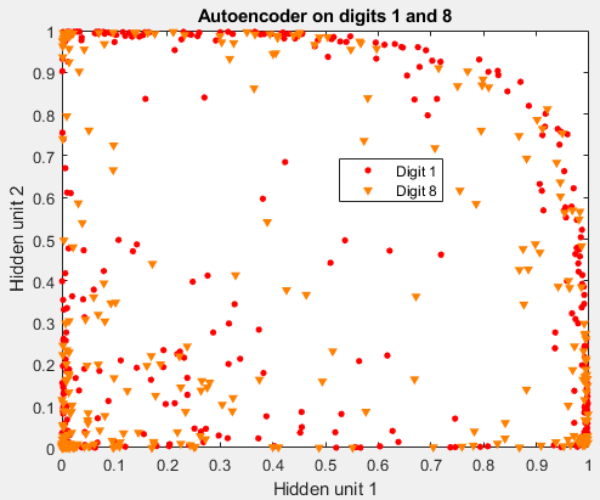

Figure 1. Autoencoder (1 vs. 8).

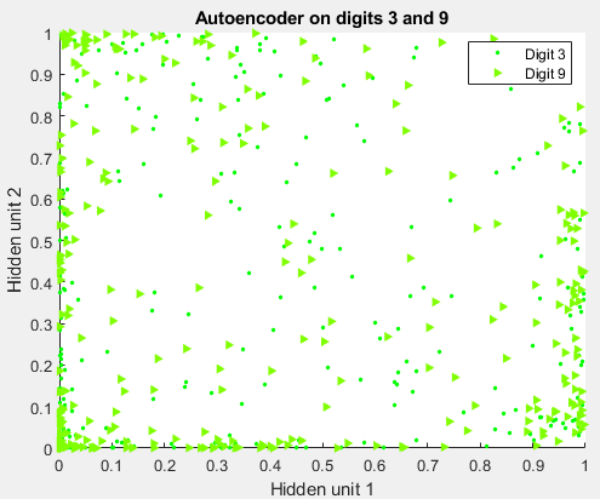

Figure 2. Autoencoder (3 vs. 9).

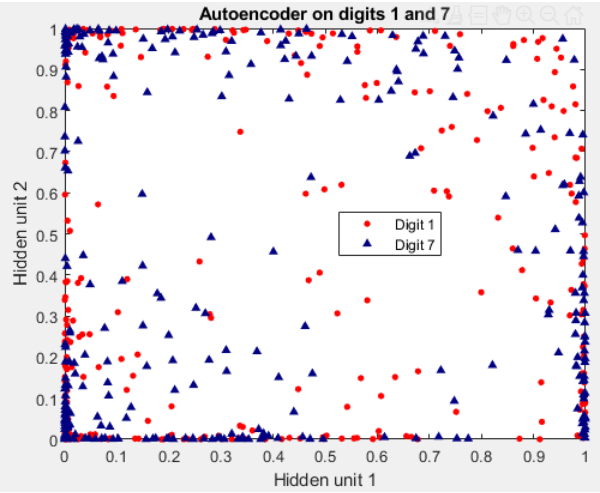

Figure 3. Autoencoder (1 vs. 7).

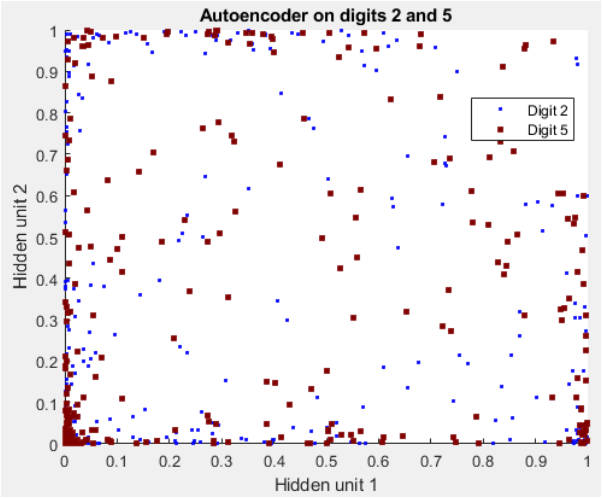

Figure 4. Autoencoder (2 vs. 5).