COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF APPLIED INFORMATICS

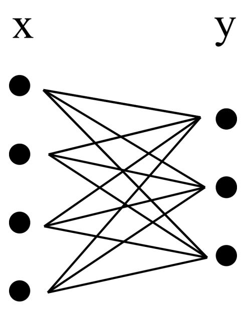
Introduction to Artificial Intelligence
Exercise 9 - Perceptron

November 16, 2022

9. PERCEPTRON

We will program a simple neural network - single layer continuous perceptron - which will learn to classify input data into multiple categories (classes).

Model:



Between each input x_j and output y_i is a connection with weight w_{ij} - at the beginning, these weights are random, during learning they improve iteratively. Besides n input neurons, there is one more x_{n+1}, so-called *bias*, that is always equal to 1.

Output computation:

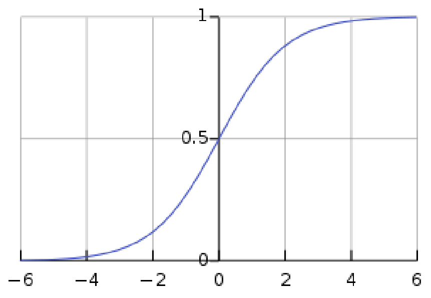
The perceptron's output is computed as a weighted sum of all inputs, which is then transformed by a so-called activation function:

$$\boldsymbol{y_i} = f\Big(\sum_{j=1}^{n+1} w_{ij}.x_j\Big), \qquad x_{n+1} = 1$$

Let's take a vector $\boldsymbol{x}$ as a column vector of all inputs and bias $\boldsymbol{x} = (x_1, ..., x_n, x_{n+1})^T$, vector $\boldsymbol{y}$ as a vector of all outputs $\boldsymbol{y} = (y_1, ..., y_m)^T$ and matrix $\boldsymbol{W}$ with size $(m \times (n+1))$ containing all weights w_{ij}, then all the outputs can be computed at once according to a simple formula:

$$\boldsymbol{y} = f(\boldsymbol{W}.\boldsymbol{x}),$$

Here the function f is applied to each element of the resulting vector. As the activation function we will use logistic sigmoid, which has a simple derivative:



$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) = \frac{\mathbf{1}}{\mathbf{1} + \boldsymbol{e}^{-\boldsymbol{x}}} \qquad \boldsymbol{f}'(\boldsymbol{x}) = \boldsymbol{y}.(\mathbf{1} - \boldsymbol{y})$$

Computation of error:

For training we have prepared a training dataset, containing n-dimensional vectors $\boldsymbol{x}$ and their respective correct outputs - m-dimensional vectors $\boldsymbol{d}$. Calculation of error is analogous to previous exercise, it is a regression error, however we have to sum through all elements of our data points. So for one data point:

$$e = \sum_{i=1}^{m} (d_i - y_i)^2 = ||\boldsymbol{d} - \boldsymbol{y}||^2$$

Total error E is just a sum (or mean) of errors e across all samples from the training data.

Weights adjustment:

For minimalization of error we use the *gradient descent* algorithm, where we adjust weights w_{ij} opposite to the direction of error e derivative, resulting in the formula:

$$w_{ij} := w_{ij} + \alpha \cdot (d_i - y_i) \cdot f'(w_i x) \cdot x_j$$

where α is constant, so-called learning rate. Inputting sigmoid derivative (instead of f'), we get:

$$w_{ij} := w_{ij} + \alpha \cdot (d_i - y_i) \cdot y_i \cdot (1 - y_i) \cdot x_j$$

And again, we do not have to compute each weight w_{ij} individually, we can use matrix operations:

$$\boldsymbol{W} := \boldsymbol{W} + \alpha \cdot (\boldsymbol{\delta} \times \boldsymbol{x}), \qquad \delta_i = (d_i - y_i) \cdot y_i \cdot (1 - y_i)$$

Please note that the operation $\boldsymbol{\delta} \times \boldsymbol{x}$ is the outer product, so the result is a matrix.

Training:

The training is performed in iterations - epochs: in each epoch we iterate through all samples from the training data and for each of them we compute the network's output, error and then we adjust the weights accordingly. The samples from training data are shuffled in each epoch to prevent some negative side-effects. The learning stops after a certain number of epochs or if the total error E is small enough.

Algorithm 1 Neural network training pseudocode:

```
1: procedure TRAINING
2:     Weight matrix $W$ initialization
3:     while stopping criterion is not met do
4:         $E \leftarrow 0$
5:         for each input $\boldsymbol{x}$ and its target $\boldsymbol{d}$ in shuffle(inputs) do
6:             $\boldsymbol{y} \leftarrow$ compute output for $\boldsymbol{x}$
7:             $e \leftarrow$ compute error of $\boldsymbol{y}$
8:             $E \leftarrow E + e$
9:             adjust weight matrix $\boldsymbol{W}$ using $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{d}$
```

Data:

We will classify digits "drawn" in 4×7 pixel grid (black is represented as 1 and white as 0), for example:

These 10 digits are copied N-times (parameter of the code, default is 10x) and then a little noise is added - each pixel is flipped with some small probability. This way we can get a richer dataset to train the network for digit classification task.

The input vector $\boldsymbol{x}$ has $4\times7 = 28$ elements (and bias). We will classify the digits into one of ten possible classes 0 to 9, while we will utilize the so-called one-hot encoding of outputs: output vector $\boldsymbol{y}$ (or $\boldsymbol{d}$) has 10 elements - all are zeros, just the one whose index equals the correct class of $\boldsymbol{x}$ is one. For example the target for digit 0 is $\boldsymbol{d}$=(1,0,0,0,0,0,0,0,0,0), for seven $\boldsymbol{d}$=(0,0,0,0,0,0,0,1,0,0), and so on.
However while training the network, it will not output exactly zeros or ones but something in between. Then "winner" neuron is the one with the highest value.
For example output $\boldsymbol{y}$= (0.4,0.2,0.1,0.1,0.3,0.2,0.9,0.1,0.2) will be interpreted as seven. There are two input files: in file *numbers.in* is for classification of digits as described above. File *odd_even.in* is also for classification of digits but only into two classes - even/odd (also using one-hot encoding). The even/odd classification is a simpler task, the network should be able to learn in a couple of tens of epochs. Classification into 10 classes is harder and can take hundreds of epochs to learn.

Assignment (1p):

Finish the program *perceptron.py* with the following functions and training procedure:

- *initialize_weights()* - initialize weight matrix *self.W* (you need to input matrix shape)
- *sigmoid(x)* - computes the value of sigmoid, while x is a number (scalar). If you use *np.exp*, then x can be a vector, and output would also be a vector.
- *compute_output(x)* - computes network's output for a given input x. **Output is the resulting vector y.**
- *compute_error(d, y)* - using the target output d and network's output y (both are vectors) computes regression error e. Output is a single number.
- *train(num_epoch)* - using the aforementioned functions trains the neural network. Learning should last *num_epoch* epochs.

Numpy:

As you may have noticed, this assignment includes a lot of vectors and matrices, so we will use *numpy* package allowing us to perform vector and matrix operations easily and effectively. Thanks to *numpy* we don't have to compute $\boldsymbol{y}$, e, $\boldsymbol{\delta}$, or $\boldsymbol{W}$ element by element in for cycles, everything can be computed using a one-line code with matrix operations. A quick intro to *numpy* is also included in the code, please have a look mainly at different possibilities of matrix multiplications.
Please note: using libraries as TensorFlow, Theano, Keras, Lasagne, etc. is forbidden.