Week 3: Perceptron and Multi-layer Perceptron

Phong Le, Willem Zuidema

November 9, 2014

Last week we studied two famous biological neuron models, Fitzhugh-Nagumo model and Izhikevich model. This week, we will firstly explore another one, which is, though less biological, very computationally practical and widely used, namely *perceptron*. Then, we will see how to combine many neurons to build complex neural networks called multi-layer perceptrons.

Required Library We'll use the library RSNNS[1], which is installed by typing `install.packages("RSNNS")`.

Required R Code At `http://www.illc.uva.nl/LaCo/clas/fncm14/assignments/computerlab-week3/` you can find the R-files you need for this exercise.

1 Classification and Regression

In *supervised learning* a *training dataset* $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ is given, and the task is to find a function $y = f(x)$ such that the function is applicable to unknown patterns x. Based on the nature of y, we can classify those tasks into two classes:

Classification is to assign a predefined label to an unknown pattern. For instance, given a picture of an animal, we need to identify if that picture is of a cat, or a dog, or a mouse, etc. If there are two categories (or two classes), the problem is called binary classification; otherwise, it is multi-class classification. For the binary classification problem, there is a special case, where patterns of the two classes are perfectly separated by a hyper-plane (see Figure 1). We call the phenomenon *linear separability*, and the hyper-plane *decision boundary*.

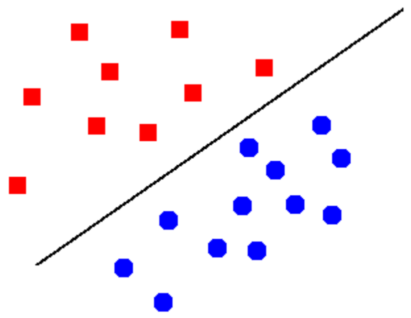

Figure 1: Example of linear separability.

[1]`http://cran.r-project.org/web/packages/RSNNS/index.html`

Regression differs from classification in that what we need to assign to an unknown pattern is a *real number*, not a label. For instance, given the height and age of a person, can we infer his/her weight?

2 Perceptron

A perceptron is a simplified neuron receiving inputs as a vector of real numbers, and outputing a real number (see Figure 2). Mathematically, a perceptron is represented by the following equation

$$y = f(w_1x_1 + w_2x_2 + ... + w_nx_n + b) = f(\mathbf{w}^T\mathbf{x} + b)$$

where $w_1, ..., w_n$ are weights, b is a bias, $x_1, ..., x_n$ are inputs, y is an output, and f is an activation function. In this section, we will use the threshold binary function (see Figure 3)

$$f(z) = \begin{cases} 1 & \textit{if } z > 0 \\ 0 & \textit{otherwise} \end{cases}$$

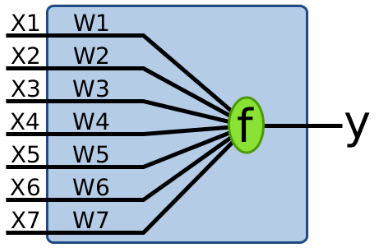

Figure 2: Perceptron (from `http://en.wikipedia.org/wiki/File:Perceptron.svg`).

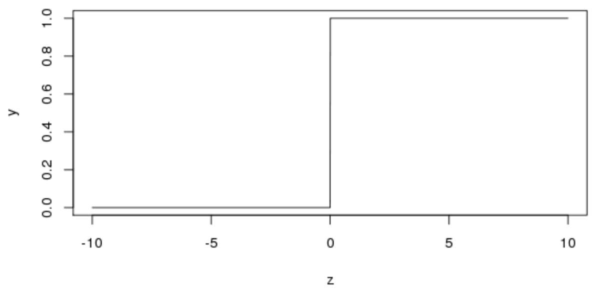

Figure 3: Threshold binary activation function.

2.1 Prediction

A new pattern $\mathbf{x}$ will be assigned the label $\hat{y} = f(\mathbf{w}^T\mathbf{x} + b)$. The mean square error (MSE) and classification accuracy on a sample $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)\}$ are respectively defined as

$$\text{error} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \quad \text{accuracy} = \frac{1}{m}\sum_{i=1}^{m} I_{y_i}(\hat{y}_i)$$

where $I_u(v)$ is an identity function, which returns 1 if $u = v$ and 0 otherwise.

2.2 Training

Traditionally, a perceptron is trained in an online-learning manner with the delta rule: we randomly pick an example $(\mathbf{x}, y)$ and update the weights and bias as follows

$$\mathbf{w}_{new} \leftarrow \mathbf{w}_{old} + \eta(y - \hat{y}_{old})\mathbf{x} \tag{1}$$
$$b_{new} \leftarrow b_{old} + \eta(y - \hat{y}_{old}) \tag{2}$$

where η is a learning rate ($0 < \eta < 1$), $\hat{y}_{old}$ is the prediction based on the old weights and bias. Intuitively, we only update the weights and bias if our prediction $\hat{y}_{old}$ is different from the true label y, and the amount of update is (negatively, in the case $y = 0$) proportional to $\mathbf{x}$.

To see why it could work, let apply the weights and bias after being updated to that example

$$\begin{aligned}\mathbf{w}_{new}^T\mathbf{x} + b_{new} &= (\mathbf{w}_{old} + \eta(y - \hat{y}_{old})\mathbf{x})^T\mathbf{x} + b_{old} + \eta(y - \hat{y}_{old}) \\ &= \mathbf{w}_{old}^T\mathbf{x} + b_{old} + \eta(y - \hat{y}_{old})(\|\mathbf{x}\| + 1)\end{aligned}$$

Here, let's assume that $\hat{y}_{old} = 0$, then $\mathbf{w}_{old}^T\mathbf{x} + b_{old} < 0$. If $y = 0$, nothing happens. Otherwise, $\eta(y - \hat{y}_{old})(\|\mathbf{x}\| + 1) > 0$, and therefore $\mathbf{w}_{new}^T\mathbf{x} + b_{new} > \mathbf{w}_{old}^T\mathbf{x} + b_{old}$. In other words, the update is to pull the decision boundary to a direction that making prediction on the example is 'less' incorrect (see Figure 4). **?** proves that if the training dataset is linearly separable, it is guaranteed to find a hyperplane correctly separates the training dataset (i.e., *error* $= 0$) in a finite number of update steps. Otherwise, the training won't converge.

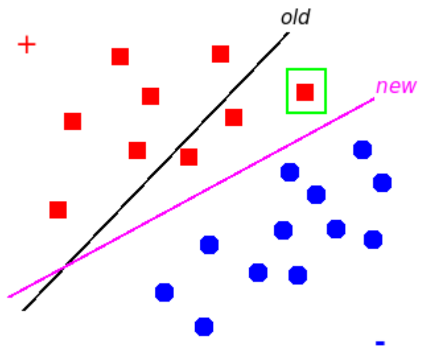

Figure 4: Example of weight update. The chosen example is enclosed in the green box. The update pulls the plane to a direction that making prediction on the example is 'less' incorrect.

Exercise 2.1: We have set up an experiment for the logic operation OR in the file 'perceptron_logic_opt.R'. You should see

```
x <- t(matrix(c(
                0,0,
                0,1,
                1,0,
                1,1),
        2,4))

# OR #
y <- matrix(c(
                0,
                1,
                1,
                1),
        4, 1)
```

which are mathematically written

$$\begin{aligned} \mathbf{x}_1 &= (0,0) \quad y_1 = 0 \\ \mathbf{x}_2 &= (0,1) \quad y_2 = 1 \\ \mathbf{x}_3 &= (1,0) \quad y_3 = 1 \\ \mathbf{x}_4 &= (1,1) \quad y_4 = 1 \end{aligned}$$

and parameters

```
maxit = 25 # number of iterations
learn.rate = 0.1 # learning rate
stepbystep = TRUE # the program will output intermediate results on R console
animation = TRUE
```

Now, typing `source("perceptron_logic_opt.R")` to execute the file. Firstly, you will see the below in your R console

```
------ init -----
weights  0 0
bias  0
learning rate:  0.1

pick x[ 1 ] =  0 0
```

then compute yourself

- true target y: ....
- prediction $\hat{y}$: ....
- weights after update: ....
- bias after update: ....

Press Enter to see if your computation is correct or not. At the same time, a plot will appear to inform you which example (black circle) is being taken, and how the current decision boundary looks like. Repeat that until the program finishes.

Exercise 2.2: Repeat the exercise 2.1 for the XOR operation. Before that, you need to open the file 'perceptron_logic_opt.R' to change `y` such that the dataset expresses the XOR operation. What could you conclude about the convergence of the training? Explain why.

Exercise 2.3: In this exercise, you will build a perceptron for a EEG data classification task. A dataset is provided in the file 'leftright.dat' which contains 2048 rows (i.e., cases) with the following format

```
feature_1    feature_2    ...     feature_96  label
```

This dataset contains data from one subject instructed to imagine either repetitive self-paced left hand movements (left, class '2') or repetitive self-paced right hand movements (right, class '3'). The dataset represents measurements on 8 channels, split out by 12 frequency components each, yileding 96 columns. (For a description of the data, see `http://bbci.de/competition/iii/desc_V.html`; our data is the training data for subject 1 using classes 2 and 3 and the 'precomputed features' option).

Have a look at this data to see how easily you can recognize the brain signatures of imagining a movement of the left or on the right. You can make scatter plots with 2 features (here '5' and '6') at a time using a command like:

```
plot(x[,5],x[,6],col=c("green","red")[y+1])
```

You can now play with the file 'perceptron_EEG_classifier.R'.

- Open the file and set values for `maxit` (say 100) and `learn.rate` (say 0.1).
- Execute the file (`source("perceptron_EEG_classifier.R")`).
- Report what you get.
- Train the perceptron and report what you get again. Do you get the same results? Why or why not?
- Look at the graph showing the error at each iteration, explain why there are many peaks.

3 Multi-layer Perceptron

The perceptron model presented above is very limited: it is theoretically applicable only to linearly separable data. In order to handle non-linearly separable data, perceptron is extended to a more complex structure, namely multi-layer perceptron (MLP). A MLP is a neural network in which neuron layers are stacked such that the output of a neuron in a layer is only allowed to be an input to neurons in the upper layer (see Figure 5).

It turns out that, if the activation functions of those neurons are non-linear, such as the sigmoid function (or logistic function) (see Figure 6)

$$sigm(z) = \frac{1}{1+e^{-z}}$$

MLP is capable to capture high non-linearity of data: **?** proves that we can approximate any continuous function at an arbitrary small error by using complex-enough MLPs.

Let's denote the weight of the link from the i-th neuron in the l-th layer to the j-th neuron in the $(l+1)$-th layer $w_{li,(l+1)j}$, then we can formalize the i-th neuron in the l-th layer by the equation

$$y_{li} = f_{li}(z_{li}) \; ; \; z_{li} = \sum_{j=1}^{n_{l-1}} w_{(l-1)j,li} y_{(l-1)j} + b_{li} \tag{3}$$

where y_{li}, f_{li}, b_{li} are respectively its output, activation function, and bias; n_l is the number of neurons in the l-th layer. (Note that for the convenience, we denote $y_{0i} \equiv x_i$.) Informally speaking, a neuron is activated by the sum of weighted outputs of the neurons in the lower layer.

3.1 Prediction

Prediction is very fast thanks to the *feedforward* algorithm (Figure 7-left). The algorithm says that, given $\mathbf{x}$, we firstly compute the outputs of the neurons in the first layer; then we compute the outputs of the neurons in the second layer; and so on until we reach the top layer.

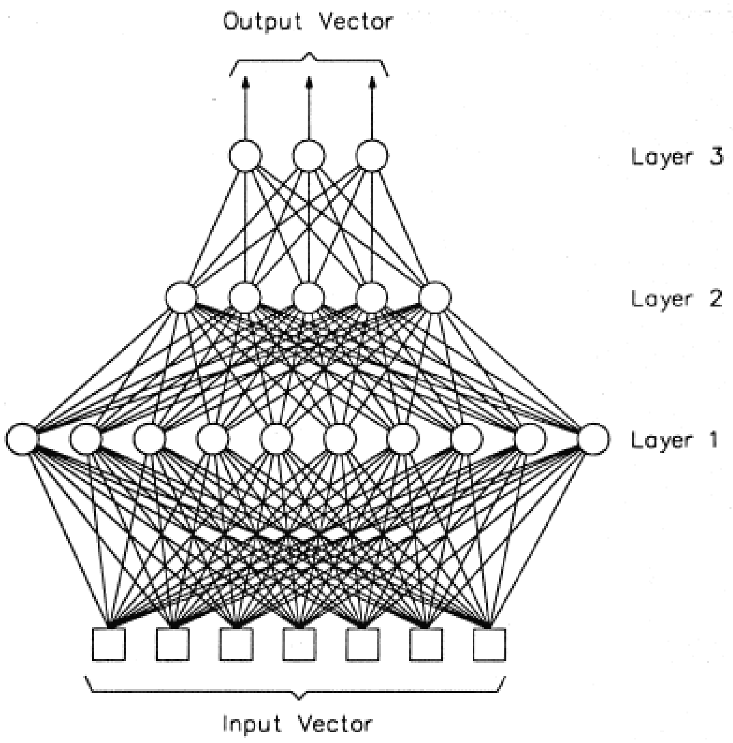

Figure 5: A 3-layer perceptron with 2 hidden layers (from `http://www.statistics4u.com/fundstat_eng/cc_ann_bp_function.html`.)

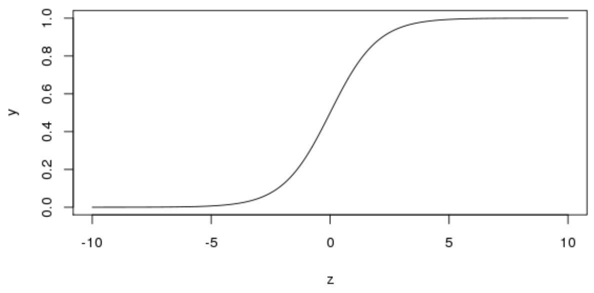

Figure 6: Sigmoid activation function.

3.2 Training

Training a MLP is to minimize an objective function w.r.t. its parameters (i.e., weights and biases) which is related to the task that the MLP is used for. For instance, for the binary classification task, the following objective function is widely used

$$E(\theta) = \frac{1}{n} \sum_{(\mathbf{x},y) \in D} (y - \hat{y})^2$$

where D is a training dataset, $\hat{y}$ is the output of the MLP given an input $\mathbf{x}$, and θ is its set of weights and biases. In order to minimize the objective function $E(\theta)$, we will use the *gradient descent* method (see Figure 8) which says that an amount of update for a parameter is negatively proportional to the gradient at its current value.

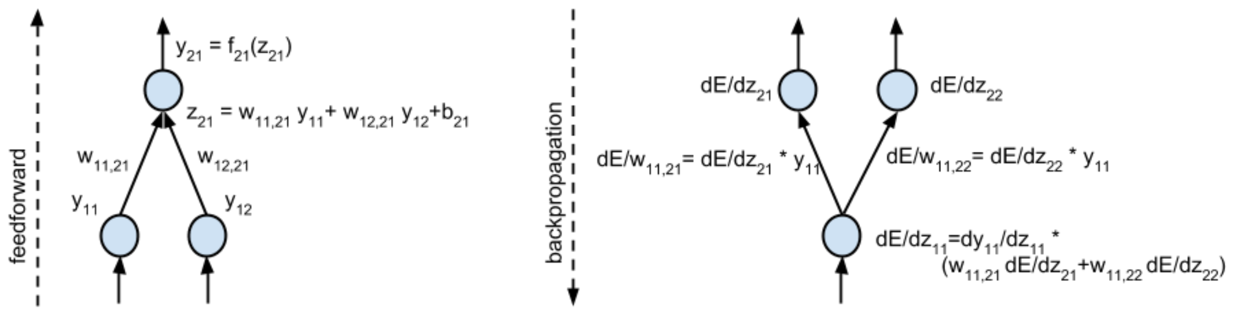

Figure 7: Feedforward (left) and backpropagation (right).

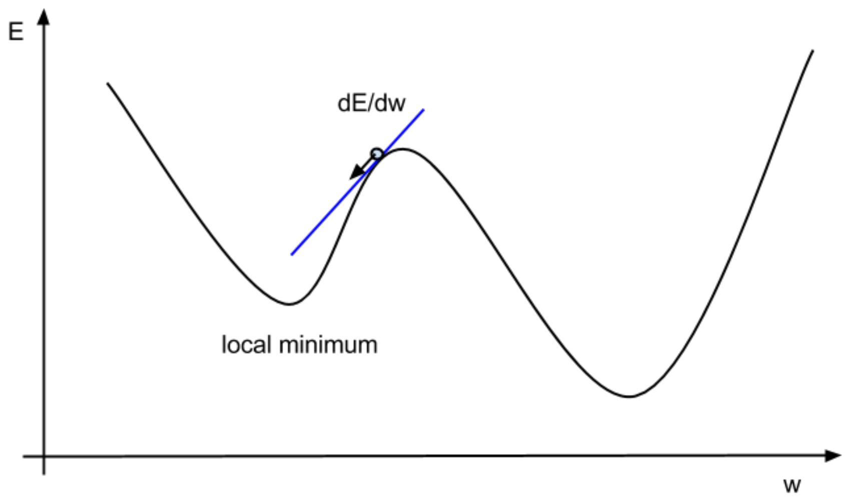

Figure 8: Illustration for the gradient descent method. The blue line is the tangent at the current value of the parameter w. If we update w by subtracting an amount proportional to the gradient at that point, the value of E will be pushed along the arrow and hence decrease. However, this method only guarantees to converge to a local minimum.

The center point of the gradient descent method is to compute the gradient $\frac{\partial E}{\partial w}$ for all $w \in \theta$ which is easily done with the chain rule:

$$\frac{\partial E}{\partial z_{Li}} = \frac{\partial E}{\partial y_{Li}} \frac{\partial y_{Li}}{\partial z_{Li}} \tag{4}$$

$$\frac{\partial E}{\partial z_{li}} = \sum_j \frac{\partial E}{\partial z_{(l+1)j}} \frac{\partial z_{(l+1)j}}{\partial z_{li}} = \sum_j \frac{\partial E}{\partial z_{(l+1)j}} w_{li,(l+1)j} \frac{\partial y_{li}}{\partial z_{li}} \tag{5}$$

$$\frac{\partial E}{\partial w_{li,(l+1)j}} = \frac{\partial E}{\partial z_{(l+1)j}} \frac{\partial z_{(l+1)j}}{\partial w_{li,(l+1)j}} = \frac{\partial E}{\partial z_{(l+1)j}} y_{li} \tag{6}$$

That's the main idea of the *back-propagation* algorithm (see Figure 7-right).

Exercise 3.1(!): You can realize that none of the formula above mentions about how to compute $\partial E/\partial b_{li}$ for all l, i. That's your task.

Exercise 3.2(!): Compute y_{li} by using the feedforward algorithm, and $w_{li,(l+1)j}, b_{li}$ by using the backpropagation algorithm for all l, i, j, for the 2-layer MLP shown in Figure 9 which all biases are 0.5 and all the neurons have the sigmoid activation function, with $\mathbf{x} = (-0.5, 0.2), y = 1$.

The RSNNS library provides us with a function named `mlp` for training a MLP. (You may want to read the description for that function in the library document `http://cran.r-project.org/web/packages/RSNNS/RSNNS.pdf` because you will use it in the next exercise). Note that, its arguments `x,y` are similar to the ones of the `perceptron` function above. If `learnFunc="Std_Backpropagation"` (which is default), then `learnFuncParams` plays the role as `learn.rate`. The default values for other parameters are almost perfect for us, you might not change them.

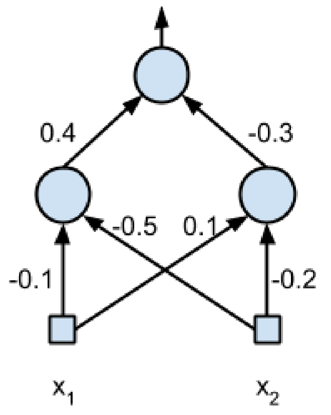

Figure 9: MLP for Exercise 3.2. All the biases are 0.5. All the neurons have the sigmoid activation function.

Exercise 3.3: Use the file 'mlp_logic_opt.R' for Exercise 2.1 and 2 (OR and XOR) (you don't need to compute by yourself anything).

- Open the file and set values for `x` and `y` (training dataset).
- Set values for `maxit` (says 100) and `learn.rate` (says 0.1), and `size` (says `c(5)`, i.e. one hidden layer with 5 neurons).
- Execute the file (`source("mlp_logic_opt.R")`).

Find a value for `maxit` such that the accuracy is 1. What is *weighted SSE*?

Exercise 3.4: Use the file 'mlp_EEG_classifier.R' for Exercise 2.3

- Open the file and set values for `maxit` (says 100) and `learn.rate` (says 0.1), and `size` (says `c(5)`, i.e. one hidden layer with 5 neurons).
- Execute the file (`source("mlp_EEG_classifier.R")`).

4 Submission

Submit a file named 'ass3_your_name.pdf' for those exercises requiring explanations and math solutions. Note that exercises marked with (!) do not need to be submitted.