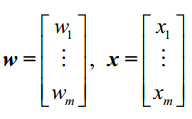
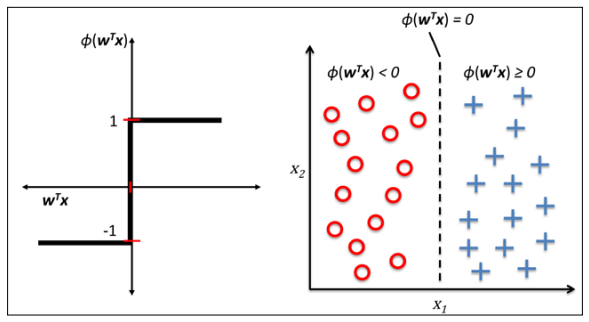
More formally, we can pose this problem as a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. We can then define an *activation function*   that takes a linear combination of certain input values x and a corresponding weight vector w , where z is the so-called net input ():



Now, if the activation of a particular sample , that is , the output of , is greater than a defined threshold , we predict class 1 and class -1, otherwise , in the perceptron algorithm , the activation function  is a simple *unit step function*, which is sometimes also called the *Heaviside step function*:



The following figure illustrates how the net input  is squashed into a binary output (-1 or 1) by the activation function of the perceptron (left subfigure) and how it can be used to discriminate between two linearly separable classes (right subfigure):



Rosenblatt’s initial perceptron rule is fairly simple and can be summarized by the following steps:

1. Initialize the weights to 0 or small random numbers;
2. For each training sample  perform the following steps:
3. Compute the output value ;
4. Update the weights

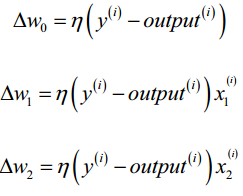
Update of each weight  in the weight vector w can be more formally written as :



The value of , which is used to update the weight , is calculated by the perceptron learning rule :

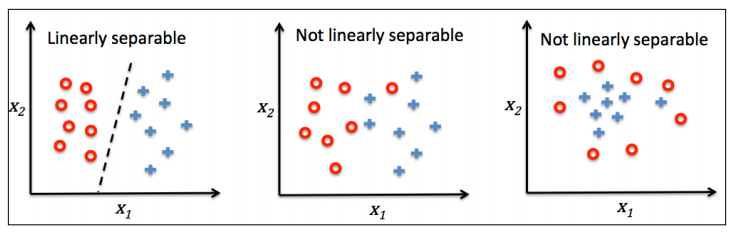


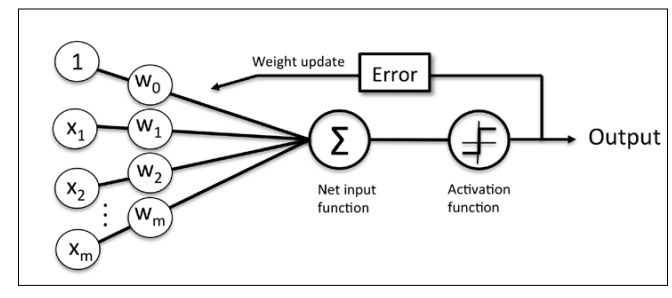
Where η is the learning rate (a constant between 0.0 and 1.0), is the true class label of the i th training sample, and  is the predicted class label. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute the  before all of the weights  were updated. Concretely, for a 2D dataset, we would write the update as follows:



It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (*epochs*) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise:

Python source code can be seen in Perceptron.py.





The preceding figure illustrates how the perceptron receives the inputs of a sample x and combines them with the weights w to compute the net input. The net input is then passed on to the activation function (here: the unit step function), which generates a binary output -1 or +1—the predicted class label of the sample. During the learning phase, this output is used to calculate the error of the prediction and update the weights.