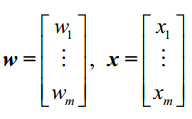
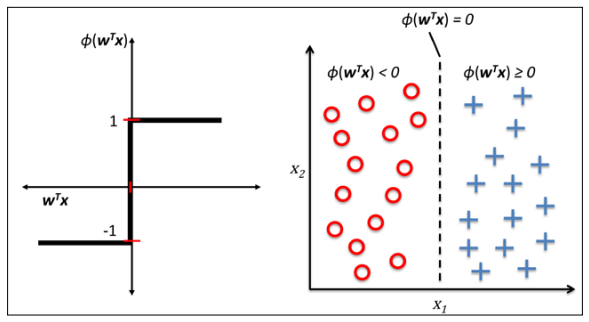
More formally, we can pose this problem as a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. We can then define an *activation function*   that takes a linear combination of certain input values x and a corresponding weight vector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

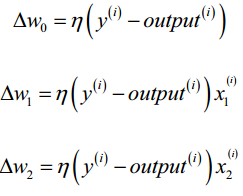
Update of each weight  in the weight vector w can be more formally written as :



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

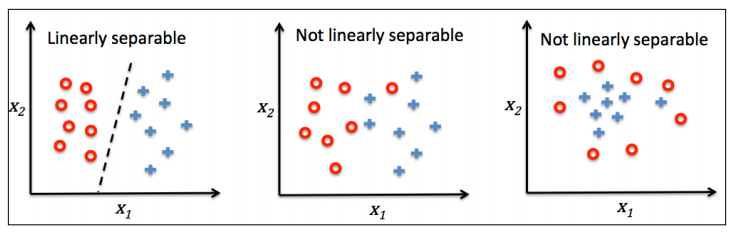


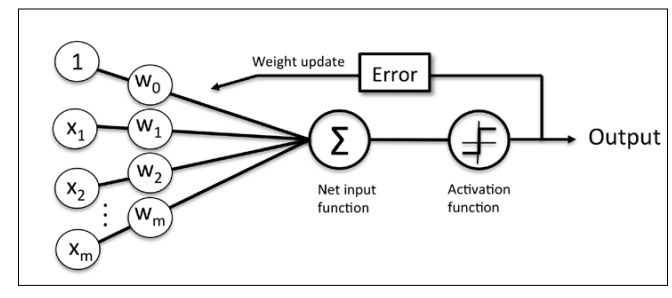
Where η is the learning rate (a constant between 0.0 and 1.0), is the true class label of the i th training sample, and  is the predicted class label. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute the  before all of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

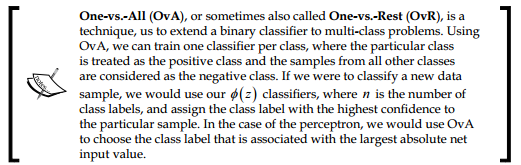
Python source code can be seen in Perceptron.py.

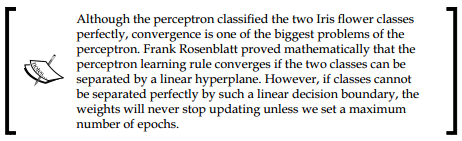




The preceding figure illustrates how the perceptron receives the inputs of a sample x and combines them with the weights w to compute the net input. The net input is then passed on to the activation function (here: the unit step function), which generates a binary output -1 or +1—the predicted class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

Source code and training method can been in Perceptron.py and DecisionBoundary.py





**Adaptive linear neurons and the Convergence of learning**

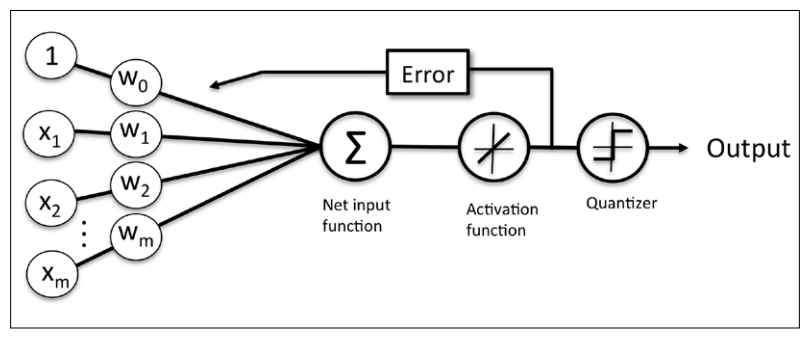
**ADAptive LInear NEuron (AdaLine):**

Adaline was published, only a few years after Frank Rosenblatt’s perceptron algorithm.

Adaline algorithm is particularly interesting because it illustrates the key concept of defining and minimizing cost functions, which will lay the groundwork for understanding more advanced machine learning algorithms for classification, such as logistic regression and support vector machines, as well as regression models .

The key difference between the Adaline rule and perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function  is simply the identity function of the net input so that 

While the linear activation function is used for learning the weights, a *quantizer*,which is similar to the unit step function that we have seen before, can then be used to predict the class labels, as illustrated in the following figure:



The difference between this method and previous perceptron algorithm is that we know to use the continuous valued output from the linear activation function to compute the model error and update the weights, rather than the binary class labels.

**Minimizing cost functions with gradient descent**

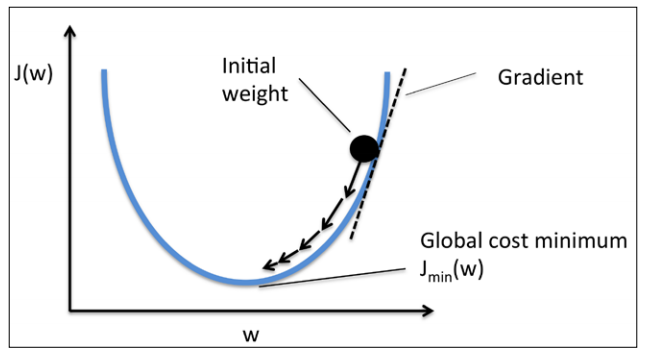
Cost function: J = sum of squared errors (SSE) between the calculated outcome and the true class label



Advantages of the cost function:

1. Differentiable
2. Is convex. Thus we can use a simple , yet powerful, optimization algorithm called gradient descent to find the weights that minimize our cost function to classify the samples in the Iris dataset.

Gradient descent can be illustrated as follow:

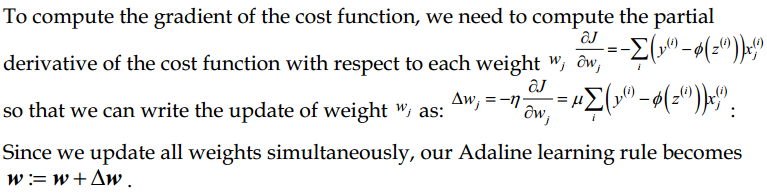


Using gradient descent, we can now update the weights by taking a step away from  
the gradient  of our cost function :



Here , the weight change  is defined as the negative gradient multiplied by the learning rate :





Learning rule difference between perceptron and Adaline:

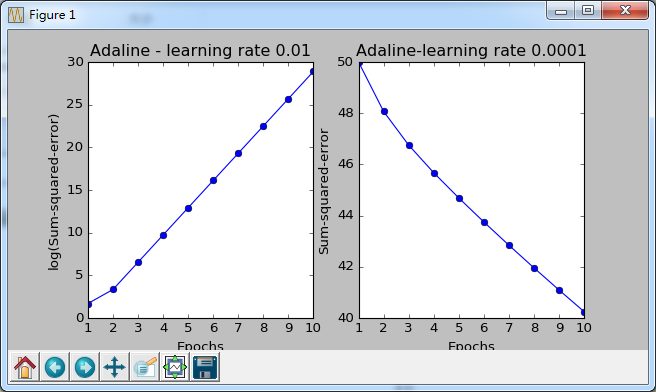
1. The  with  is a real number and not an integer class label.
2. Weight update is calculated based on all samples in the training set (instead of updating the weights incrementally after each sample).

Python implemented Adaptive Linear Neuron:

Source code :AdalineGD.py

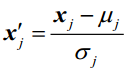
The learning rate η , as well as the number of epochs n\_iter, are the so-called *hyperparameters* of the perceptron and Adaline learning algorithms.

plot the cost against the number of epochs for two different learning rates:  
**>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))**  
**>>> ada1 = AdalineGD(n\_iter=10, eta=0.01).fit(X, y)**  
**>>> ax[0].plot(range(1, len(ada1.cost\_) + 1),**  
**... np.log10(ada1.cost\_), marker='o')**  
**>>> ax[0].set\_xlabel('Epochs')**  
**>>> ax[0].set\_ylabel('log(Sum-squared-error)')**  
**>>> ax[0].set\_title('Adaline - Learning rate 0.01')**  
**>>> ada2 = AdalineGD(n\_iter=10, eta=0.0001).fit(X, y)**  
**>>> ax[1].plot(range(1, len(ada2.cost\_) + 1),**  
**... ada2.cost\_, marker='o')**  
**>>> ax[1].set\_xlabel('Epochs')**  
**>>> ax[1].set\_ylabel('Sum-squared-error')**  
**>>> ax[1].set\_title('Adaline - Learning rate 0.0001')**  
**>>> plt.show()**



The left chart shows what could happen if we choose a learning  
rate that is too large—instead of minimizing the cost function, the error becomes larger in every epoch because we *overshoot* the global minimum.

Gradient descent is one of the many algorithms that benefit from feature scaling. Here we will use a feature scaling method called standardization, which gives our data the property of a standard normal distribution. The mean of each feature is centered at value 0 and the feature column has a standard deviation of 1. For example, to standardize the j th feature, we simply need to subtract the sample mean µ j from every training sample and divide it by its standard deviation σ j:



Standardization can easily be achieved using the NumPy methods mean and std:

>>> X\_std = np.copy(X)

>>> X\_std[:,0] = (X[:,0]-X[:,0].mean())/X[:,0].std();

>>> X\_std[:,1] = (X[:,1]-X[:,1].mean())/X[:,0].std();

After standardization, we will train the Adaline again and see that it now converges using a learning rate η = 0.01:

However, note that the SSE remains non-zero even though all samples were classifed correctly.