What is the difference between a Perceptron, Adaline, and neural network model?

Both Adaline and the Perceptron are (single-layer) neural network models. The Perceptron is one of the oldest and simplest learning algorithms out there, and I would consider Adaline as an improvement over the Perceptron.

What Adaline and the Perceptron have in common

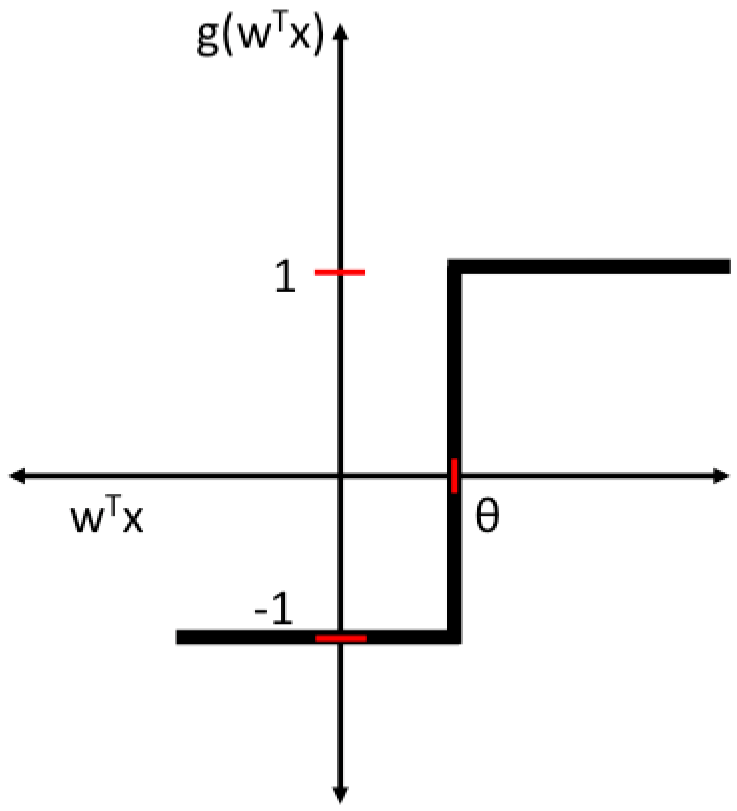
- they are classifiers for binary classification
- both have a linear decision boundary
- both can learn iteratively, sample by sample (the Perceptron naturally, and Adaline via stochastic gradient descent)
- both use a threshold function

Before we talk about the differences, let’s talk about the inputs first. The first step in the two algorithms is to compute the so-called net input *z* as the linear combination of our feature variables *x* and the model weights *w*.

$$\mathbf{z} = w_1x_1 + \cdots + w_mx_m = \sum_{j=1}^{m} x_jw_j = \mathbf{w}^T\mathbf{x}$$

Then, in the Perceptron and Adaline, we define a threshold function to make a prediction. I.e., if *z* is greater than a threshold theta, we predict class 1, and 0 otherwise:

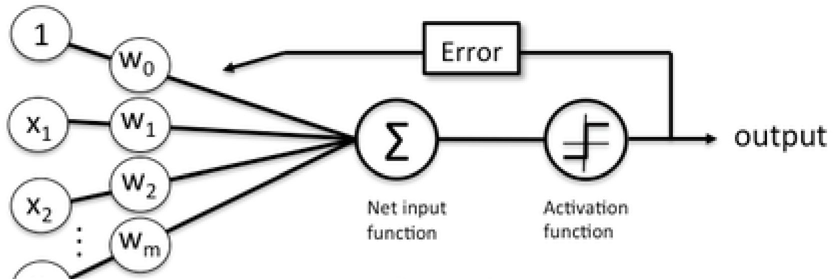
$$g(\mathbf{z}) = \begin{cases} 1 & \text{if } \mathbf{z} \geq \theta \\ -1 & \text{otherwise.} \end{cases}$$



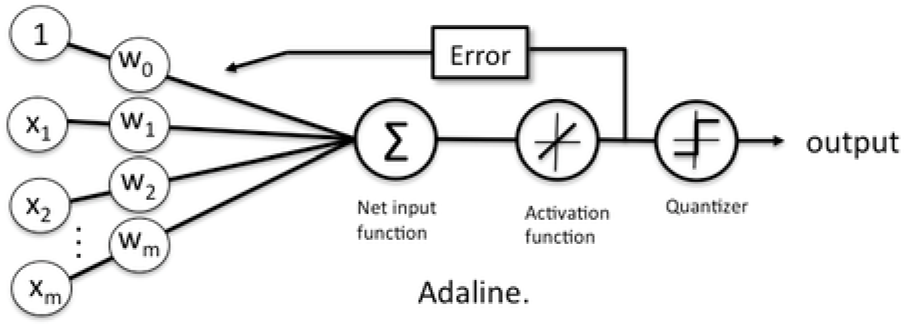
The differences between the Perceptron and Adaline

- the Perceptron **uses the class labels to learn model coefficients**
- Adaline **uses continuous predicted values (from the net input**) to learn the model coefficients, which is more “powerful” since it tells us by “how much” we were right or wrong

So, in the perceptron, as illustrated below, we simply use the predicted class labels to update the weights, and in Adaline, we use a continuous response:



Perceptron rule.

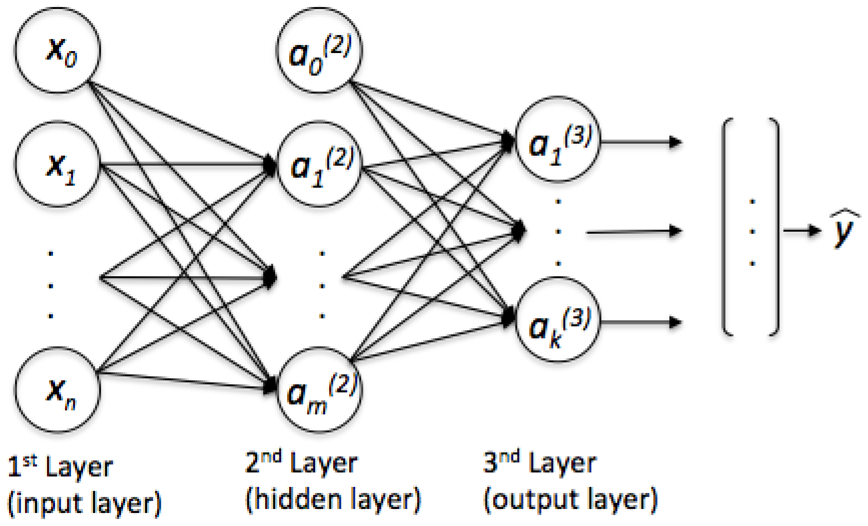


Adaline.

Important note: the "activation function" in Adaline just for illustrative purposes; here, this activation function is simply the identity function i.e. net input is used for weight update.

Multi-layer neural networks

Although you haven't asked about multi-layer neural networks specifically, let me add a few sentences about one of the oldest and most popular multi-layer neural network architectures: the Multi-Layer Perceptron (MLP). The term "Perceptron" is a little bit unfortunate in this context, since it really doesn't have much to do with Rosenblatt's Perceptron algorithm.

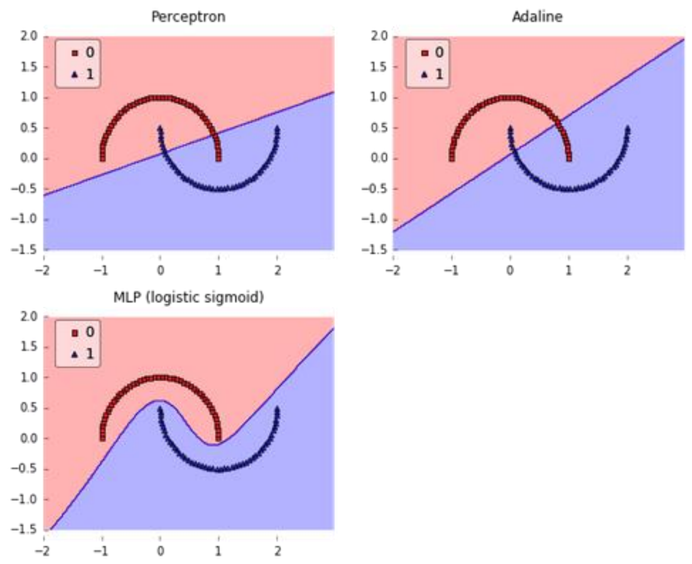


MLPs can basically be understood as a network of multiple artificial neurons over multiple layers. Here, the activation function is not linear (like in Adaline), but we use a non-linear activation function like the logistic sigmoid (the one that we use in logistic regression) or the hyperbolic tangent, or a piecewise-linear activation function such as the rectifier linear unit (ReLU). In addition, we often use a softmax function (a generalization of the logistic sigmoid for multi-class problems) in the output layer, and a threshold function to turn the predicted probabilities (by the softmax) into class labels.

So, what the advantage of the MLP over the classic Perceptron and Adaline?

By connecting the artificial neurons in this network through non-linear activation functions, we can create complex, non-linear decision boundaries that allow us to tackle problems where the different classes are not linearly separable.

Let me show you an example :)



Multiple Adaptive Linear Neuron (Madaline)

Madaline which stands for **Multiple Adaptive Linear Neuron**, is a network which consists of many Adalines in parallel.

It will have a single output unit. Some important points about Madaline are as follows −

- It is just like a multilayer perceptron, where **Adaline will act as a hidden unit between the input and the Madaline layer.**
- The weights and the bias between the input and Adaline layers, as in we see in the Adaline architecture, are **adjustable.**
- The Adaline and Madaline layers have **fixed weights and bias of 1.**

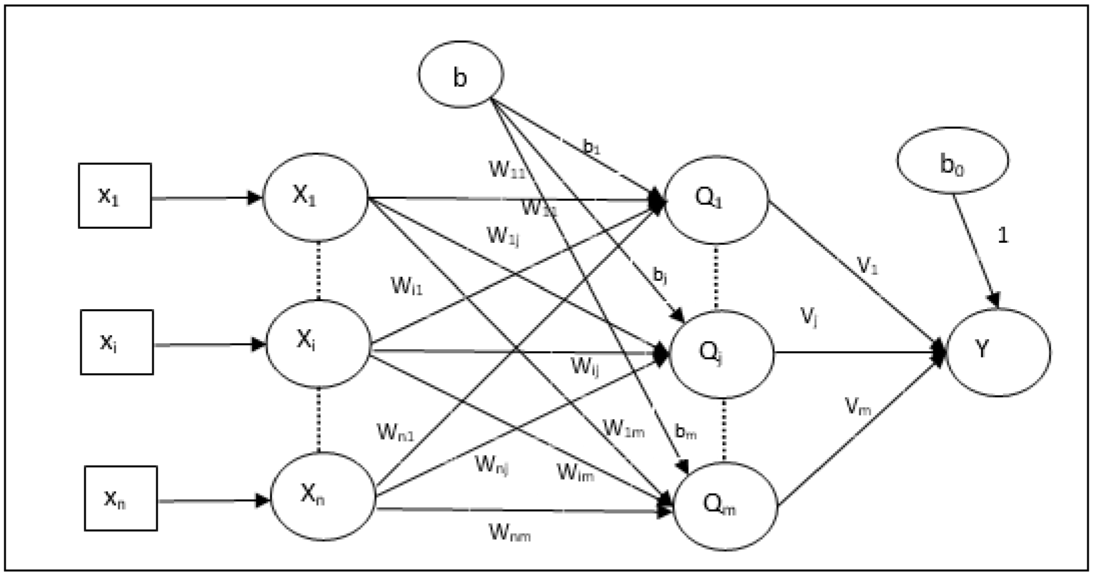
Architecture

The architecture of Madaline consists of **"n"** neurons of the input layer,

"m" neurons of the Adaline layer, and

1 neuron of the Madaline layer.

The Adaline layer can be considered as the hidden layer as it is between the input layer and the output layer, i.e. the Madaline layer.



Training Algorithm

By now we know that only the weights and bias between the input and the Adaline layer are to be adjusted, and the weights and bias between the Adaline and the Madaline layer are fixed.

Step 1 − Initialize the following to start the training −

- Weights
- Bias
- Learning rate αα

For easy calculation and simplicity, weights and bias must be set equal to 0 and the learning rate must be set equal to 1.

Step 2 − Continue step 3-8 when the stopping condition is not true.

Step 3 − Continue step 4-7 for every bipolar training pair **s:t**.

Step 4 − Activate each input unit as follows −

$$x_i = s_i (i = 1 \text{ to } n) x_i = s_i (i = 1 \text{ to } n)$$

Step 5 − Obtain the net input at each hidden layer, i.e. the Adaline layer with the following relation −

$$Q_{inj}=b_j+\sum_{i}^{n}x_i w_{ij} \quad j=1\,to\,m$$ Qinj=bj+∑inxiwijj=1tom

Here **‘b’** is bias and **‘n’** is the total number of input neurons.

Step 6 − Apply the following activation function to obtain the final output at the Adaline and the Madaline layer −

$$f(x)=\begin{cases}1 & if\,x\geqslant 0\\ -1 & if\,x<0\end{cases}$$ f(x)={1ifx⩾0−1ifx<0

Output at the hidden (Adaline) unit

$$Q_j=f(Q_{inj})$$ Qj=f(Qinj)

Final output of the network

$$y=f(y_{in})$$ y=f(yin)

i.e. $y_{inj}=b_0+\sum_{j=1}^{m}Q_j v_j$ yinj=b0+∑j=1mQjvj

Step 7 − Calculate the error and adjust the weights as follows −

Case 1 − if **y ≠ t** and **t = 1** then,

$$w_{ij}(new)=w_{ij}(old)+\alpha(1-Q_{inj})x_i$$ wij(new)=wij(old)+α(1−Qinj)xi

$$b_j(new)=b_j(old)+\alpha(1-Q_{inj})$$ bj(new)=bj(old)+α(1−Qinj)

In this case, the weights would be updated on $\mathbf{Q_j}$ where the net input is close to 0 because **t = 1**.

Case 2 − if **y ≠ t** and **t = -1** then,

$$w_{ik}(new)=w_{ik}(old)+\alpha(-1-Q_{ink})x_i$$ wik(new)=wik(old)+α(−1−Qink)xi

$$b_k(new)=b_k(old)+\alpha(-1-Q_{ink})$$ bk(new)=bk(old)+α(−1−Qink)

In this case, the weights would be updated on $\mathbf{Q_k}$ where the net input is positive because **t = -1**.

Here **‘y’** is the actual output and **‘t’** is the desired/target output.

Case 3 − if **y = t** then

There would be no change in weights.

Step 8 − Test for the stopping condition, which will happen when there is no change in weight or the highest weight change occurred during training is smaller than the specified tolerance.

Back Propagation Neural Networks

Back Propagation Neural (BPN) is **a multilayer neural network** consisting of the input layer, at least one hidden layer and output layer. As its name suggests, back propagating will take place in this network. The error which is calculated at the output layer, by comparing the target output and the actual output, will be propagated back towards the input layer.

Architecture

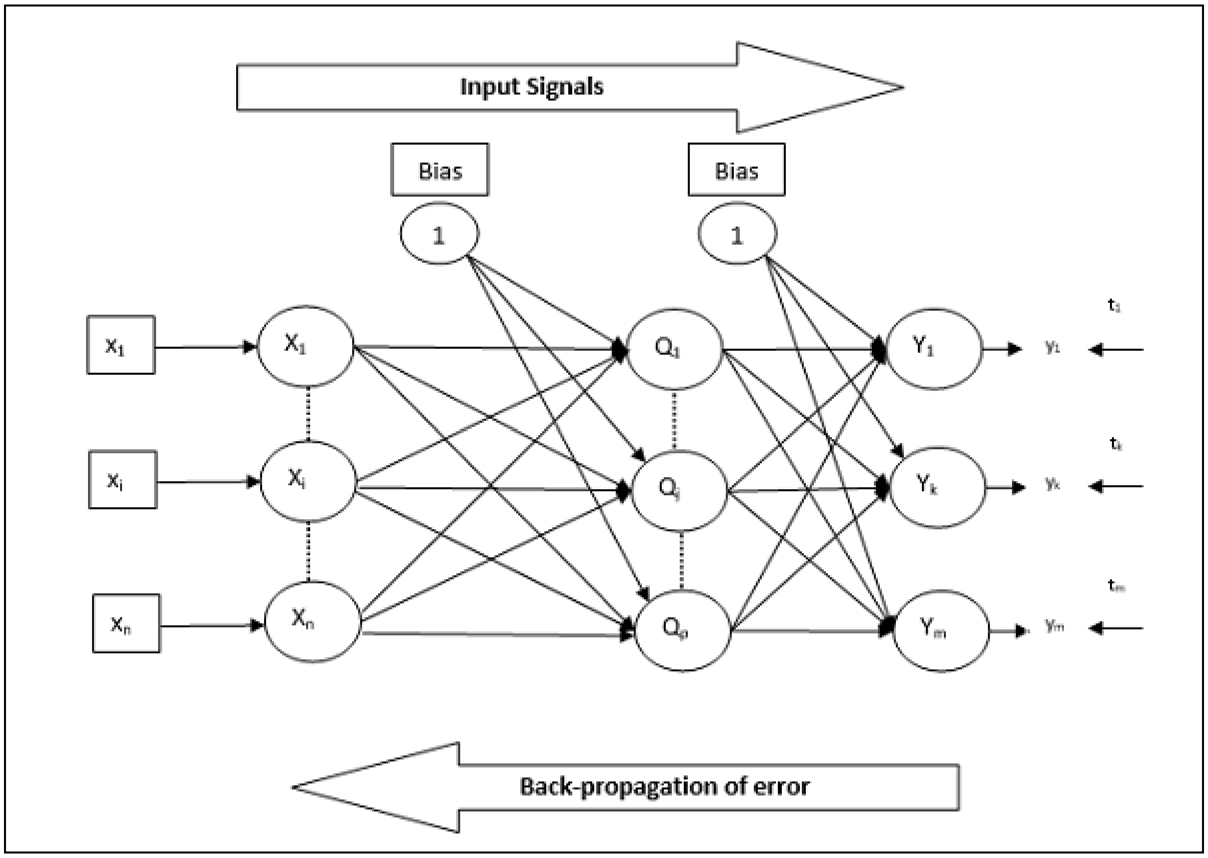
As shown in the diagram, the architecture of BPN has three interconnected layers having weights on them.

The hidden layer as well as the output layer also has bias, whose weight is always 1, on them.

, the working of BPN is in two phases.

One phase sends the **signal from the input layer to the output layer**, and

the other **phase back propagates the error from the output layer to the input layer**.



Training Algorithm

For training, BPN will use binary sigmoid activation function. The training of BPN will have the following three phases.

- **Phase 1** − Feed Forward Phase
- **Phase 2** − Back Propagation of error
- **Phase 3** − Updating of weights

All these steps will be concluded in the algorithm as follows

Step 1 − Initialize the following to start the training −

- Weights
- Learning rate αα

For easy calculation and simplicity, take some small random values.

Step 2 − Continue step 3-11 when the stopping condition is not true.

Step 3 − Continue step 4-10 for every training pair.

Phase 1

Step 4 − Each input unit receives input signal **x_i** and sends it to the hidden unit for all **$i = 1$ to n**

Step 5 − Calculate the net input at the hidden unit using the following relation −

$$Q_{inj} = b_{0j} + \sum_{i=1}^{n} x_i v_{ij} \quad j = 1 \text{ to } p$$
Qinj=b0j+∑i=1nxivijj=1top

Here **b_{0j}** is the bias on hidden unit, **v_{ij}** is the weight on **j** unit of the hidden layer coming from **i** unit of the input layer.

Now calculate the net output by applying the following activation function

$$Q_j = f(Q_{inj})$$
Qj=f(Qinj)

Send these output signals of the hidden layer units to the output layer units.

Step 6 − Calculate the net input at the output layer unit using the following relation −

$$y_{ink} = b_{0k} + \sum_{j=1}^{p} Q_j w_{jk} \quad k = 1 \text{ to } m$$
yink=b0k+∑j=1pQjwjkk=1tom

Here **b_{0k}** is the bias on output unit, **w_{jk}** is the weight on **k** unit of the output layer coming from **j** unit of the hidden layer.

Calculate the net output by applying the following activation function

$$y_k = f(y_{ink})$$
yk=f(yink)

Phase 2

Step 7 − Compute the error correcting term, in correspondence with the target pattern received at each output unit, as follows −

$$\delta_k=(t_k-y_k)f'(y_{ink})$$

δk=(tk−yk)f′(yink)

On this basis, update the weight and bias as follows −

$$\Delta v_{jk}=\alpha\delta_k Q_{ij}$$

Δvjk=αδkQij

$$\Delta b_{0k}=\alpha\delta_k$$

Δb0k=αδk

Then, send δ_kδk back to the hidden layer.

Step 8 − Now each hidden unit will be the sum of its delta inputs from the output units.

$$\delta_{inj}=\sum_{k=1}^{m}\delta_k w_{jk}$$

δinj=∑k=1mδkwjk

Error term can be calculated as follows −

Here’s the Python code if you want to reproduce these plots:

```
from mlxtend.evaluate import plot_decision_regions

from mlxtend.classifier import Perceptron

from mlxtend.classifier import Adaline

from mlxtend.classifier import MultiLayerPerceptron

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_moons

import matplotlib.gridspec as gridspec

import itertools


gs = gridspec.GridSpec(2, 2)xw

X, y = make_moons(n_samples=100, random_state=123)

fig = plt.figure(figsize=(10,8))
```

```
ppn = Perceptron(epochs=50, eta=0.05, random_seed=0)

ppn.fit(X, y)

ada = Adaline(epochs=50, eta=0.05, random_seed=0)

ada.fit(X, y)


mlp = MultiLayerPerceptron(n_output=len(np.unique(y)),

                           n_features=X.shape[1],

                           n_hidden=150,

                           l2=0.0,

                           l1=0.0,

                           epochs=500,

                           eta=0.01,

                           alpha=0.0,

                           decrease_const=0.0,

                           minibatches=1,

                           shuffle_init=False,

                           shuffle_epoch=False,

                           random_seed=0)


mlp = mlp.fit(X, y)



for clf, lab, grd in zip([ppn, ppn, mlp],

                         ['Perceptron', 'Adaline', 'MLP (logistic sigmoid)'],

                         itertools.product([0, 1], repeat=2)):


    clf.fit(X, y)

    ax = plt.subplot(gs[grd[0], grd[1]])

    fig = plot_decision_regions(X=X, y=y, clf=clf, legend=2)
    plt.title(lab)


plt.show()
```

Error term can be calculated as follows −

$$\delta_j=\delta_{inj}f'(Q_{inj})\delta j=\delta injf'(Qinj)$$

On this basis, update the weight and bias as follows −

$$\Delta w_{ij}=\alpha\delta_j x_i \Delta wij=\alpha\delta jxi$$

$$\Delta b_{0j}=\alpha\delta_j \Delta b0j=\alpha\delta j$$

Phase 3

Step 9 − Each output unit ***(y_kk = 1 to m)*** updates the weight and bias as follows −

$$v_{jk}(new)=v_{jk}(old)+\Delta v_{jk} vjk(new)=vjk(old)+\Delta vjk$$

$$b_{0k}(new)=b_{0k}(old)+\Delta b_{0k} b0k(new)=b0k(old)+\Delta b0k$$

Step 10 − Each output unit ***(z_jj = 1 to p)*** updates the weight and bias as follows −

$$w_{ij}(new)=w_{ij}(old)+\Delta w_{ij} wij(new)=wij(old)+\Delta wij$$

$$b_{0j}(new)=b_{0j}(old)+\Delta b_{0j} b0j(new)=b0j(old)+\Delta b0j$$

Step 11 − Check for the stopping condition, which may be either the number of epochs reached or the target output matches the actual output.

Ref: https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_supervised_learning.htm___