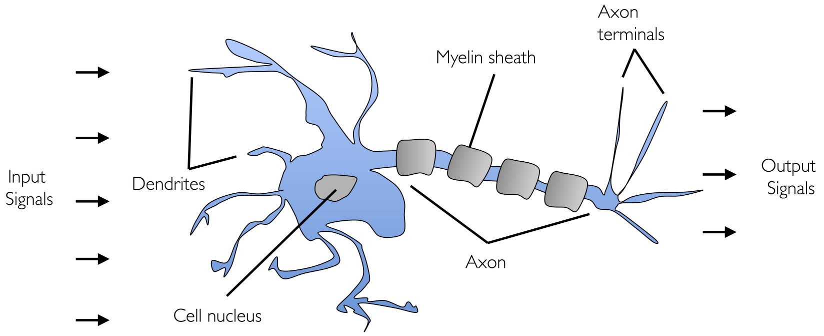
Chapter 2

Training Machine Learning Algorithms for Classification

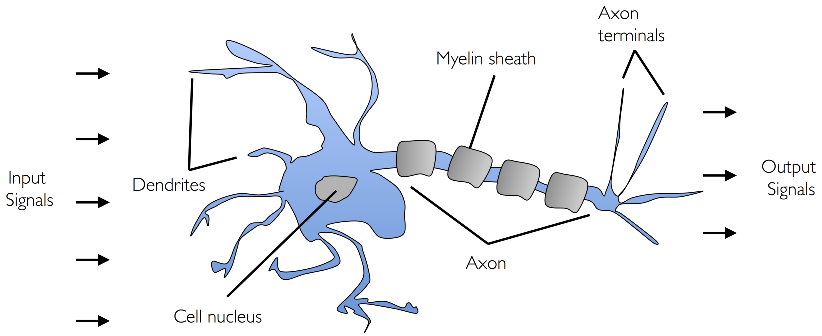
August 13, 2018

Logic Gate



- Warren McCullockand Walter Pitts published the first concept of a simplified brain cell, the so-called McCullock - Pitts (MCP) neuron, in 1943.
- Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals.

Logic Gate



- Simple logic gate with binary outputs
- Signals arrive at dendrites
- Integrated into cell body
- If signal exceeds threshold, generate output, and pass to axon

1. In 1957, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model.
2. He proposed an algorithm that would automatically learn the optimal weight coefficients that are the input features in order to make the decision of whether a neuron fires or not.
3. In the context of supervised learning and classification, such an algorithm could then be used to predict if a sample belongs to one class or the other.

Perceptron

- Binary classification task
- Positive class (1) vs. negative class (-1)
- Define activation function $\phi(z)$ or a decision function that takes a linear combination of certain input values x and a corresponding weight vector w
- Takes as input a dot product of input and weights
- Net input: $z = w_1 x_1 + \cdots + w_m x_m$

$$\mathbf{w} = \begin{bmatrix} w^{(1)} \\ w^{(2)} \\ \vdots \\ w^{(m)} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(m)} \end{bmatrix}$$

Heaviside step function

- $\phi(z)$ known as activation
- if activation above some threshold, predict class 1
- predict class -1 otherwise

Heaviside Step Function

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

Step function simplified

Bring the threshold θ to the left side of the equation and define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$, so that we write $\mathbf{z}$ in a more compact form

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \mathbf{w}^{\mathbf{T}} \mathbf{x}$$

and

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases} .$$

The negative threshold, or weight, $w_0 = -\theta$, is usually called the bias unit.

Basic Linear Algebra

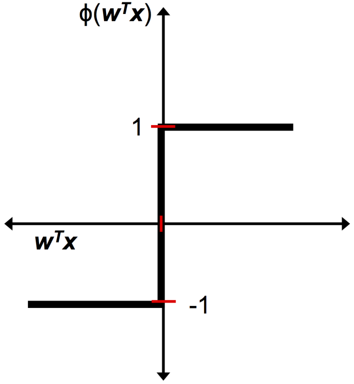
Vector dot product

$$z = \mathbf{w}^{\mathbf{T}}\mathbf{x} = \sum_{j=0}^{m} \mathbf{w_j x_j}$$

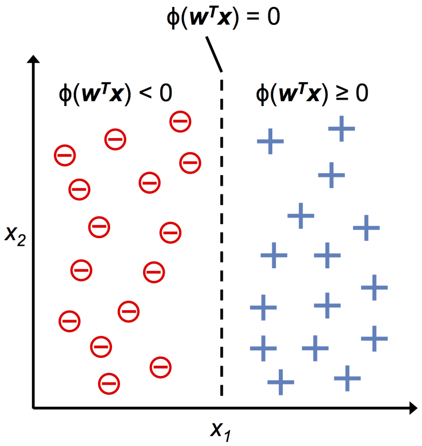
$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32.$$

For this case, it predicts the instance belongs to the positive class.

Perceptron



$\phi(\boldsymbol{w}^T\boldsymbol{x})$
1
$\boldsymbol{w}^T\boldsymbol{x}$
-1



$\phi(\boldsymbol{w}^T\boldsymbol{x}) = 0$
$\phi(\boldsymbol{w}^T\boldsymbol{x}) < 0$
$\phi(\boldsymbol{w}^T\boldsymbol{x}) \geq 0$
x_2
x_1

Rosenblatt perceptron algorithm

1. Initialize the weights to 0 or small random numbers.
2. For each training sample $\mathbf{x}^{(i)}$, perform the following steps:
    1. Compute the output value $\hat{y}$.
    2. Update the weights.

Weight update

Weight update rule:

$$w_j := w_j + \Delta w_j$$

Perceptron learning rule:

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

Where η is the learning rate (a constant between 0.0 and 1.0), $y^{(i)}$ is the true class label of the ith training sample, $\hat{y}^{(i)}$ is the predicted class label, and $x_j^{(i)}$ is the training sample.

Update for a 2-dimensional dataset

$$\Delta w_0 = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

$$\Delta w_1 = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_1^{(i)}$$

$$\Delta w_2 = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_2^{(i)}$$

Update rule examples

Correct prediction, weights unchanged:

$$\Delta w_j = \eta\left(-1 - -1\right)x_j^{(i)} = 0$$

$$\Delta w_j = \eta\left(1 - 1\right)x_j^{(i)} = 0$$

Wrong prediction, weights pushed towards the positive or negative class:

$$\Delta w_j = \eta\left(1 - -1\right)x_j^{(i)} = \eta(2)x_j^{(i)}$$

$$\Delta w_j = \eta\left(-1 - 1\right)x_j^{(i)} = \eta(-2)x_j^{(i)}$$

Example

$$\hat{y}^{(i)} = -1, y^{(i)} = +1, \eta = 1$$

$$\Delta w_j = \left(1 - -1\right)0.5 = (2)0.5 = 1$$

The weight update is proportional to the value of $x_j^{(i)}$. If $x_j^{(i)} = 2$, that is incorrectly classified as -1, we would push the decision boundary by an even larger extent to classify this sample correctly the next time:

$$\Delta w_j = \left(1 - -1\right)2 = (2)2 = 4$$

Convergence

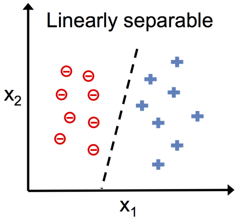
Convergence guaranteed if

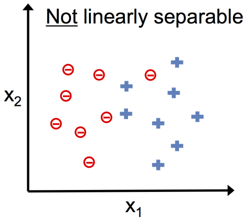
- The two classess linearly separable
- Learning rate is sufficiently small

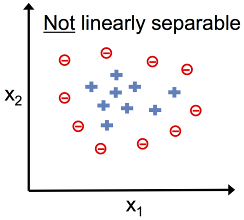
If classes cannot be seprated:

- Set a maximum number of passes over the training dataset (epochs)
- Set a threshold for the number of tolerated misclassifications
- Otherwise, it will never stop updating weights (converge)

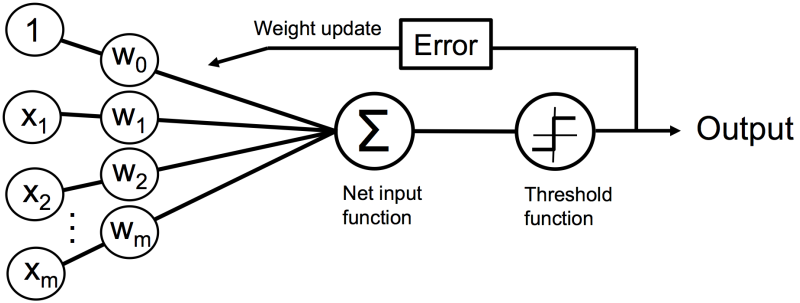
Linear separability







Perceptron Diagram



- The perceptron receives the input of a sample x and combines them with the weights *w* to compute the net input.
- The net input is then passed on the threshold function, which generates a binary output -1 or +1 (the predicted class label of the sample).
- During the learning phase, the output is used to calculate the error of the prediction and update the weights.

Perceptron implementation

iPython notebook

Adaline - ADAptive LInear NEuron

- Oct/1960
- Adaline illustrates the key concepts of defining and minimizing continuous cost functions.
- This lays the groundwork for understanding more advanced machine learning algorithms for classification, such as:
  - logistic regression
  - support vector machines
  - 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.

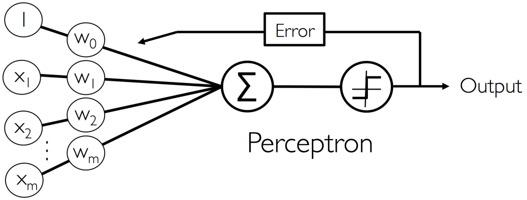
ADAPtive LInear NEuron (Adaline)

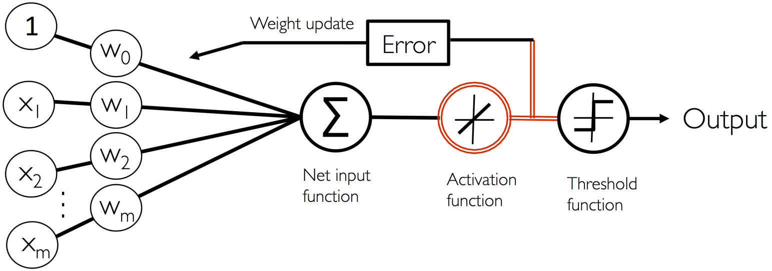
- Weights updated based on a linear activation function
- Remember that perceptron used a unit step function
- In Adaline, $\phi(z)$ is simply the identity function of the net input

$$\phi(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

- A quantizer is then used to predict class label (a threshold function to make the final prediction, which is similar to the unit step function that we have seen earlier)

Adaline: notice the difference with perceptron





Adaptive Linear Neuron (Adaline)

Adaline: notice the difference with perceptron

- Basically, the difference is Adaline compares the true class labels with the linear activation function continuous valued output to compute the model error and update the weights.
- In contrast, the perceptron compares the true class labels to the predicted class labels.

Cost functions

- ML algorithms often define an *objective* function
- This function is optimized during learning
- It is often a *cost* function we want to minimize
- Adaline uses a cost function $J(\cdot)$
- Learns weights as the sum of squared errors (SSE)

$$J(\mathbf{w}) = \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

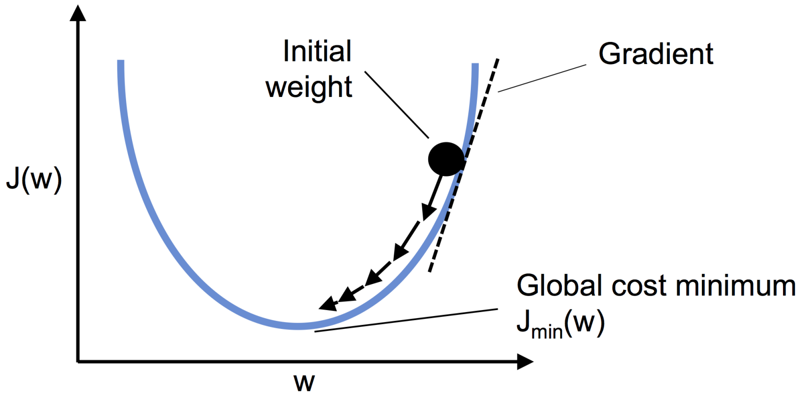
Advantages of Adaline cost function

- The term $\frac{1}{2}$ is just added for our convenience, which will make it easier to derive the gradient
- The main advantage of this continuous linear activation function, in contrast to the unit step function, is that the cost function becomes differentiable.
- In order to find the weights that minimize our cost function to classify the samples we can use gradient descent.

What is the gradient? Ask Wikipedia:

- The gradient is a multi-variable generalization of the derivative. While a derivative can be defined on functions of a single variable, for functions of several variables, the gradient takes its place.
- Like the derivative, the gradient represents the slope of the tangent of the graph of the function. More precisely, the gradient points in the direction of the greatest rate of increase of the function, and its magnitude is the slope of the graph in that direction.

Gradient Descent



Gradient descent: an intuition

- Suppose you are at the top of a mountain, and you have to reach a lake which is at the lowest point of the mountain (a.k.a valley). A twist is that you are blindfolded and you have zero visibility to see where you are headed. So, what approach will you take to reach the lake?
- The best way is to check the ground near you and observe where the land tends to descend. This will give an idea in what direction you should take your first step. If you follow the descending path, it is very likely you would reach the lake.

https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/

Gradient Descent

- Weights updated by taking small steps
- Step size determined by learning rate
- Take a step in the opposite direction of the gradient $\nabla J(\mathbf{w})$ of the cost function

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}.$$

- Where the weight change $\Delta\mathbf{w}$ *the negative gradient* multiplied by the learning rate η:

$$\Delta\mathbf{w} = -\eta\nabla J(\mathbf{w})$$

Gradient computation

To compute the gradient of the cost function, we need to compute the partial derivative of the cost function with respect to each weight w_j,

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right) x_j^{(i)},$$

Weight update of weight w_j

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right) x_j^{(i)}$$

We update all weights simultaneously, so Adaline learning rule becomes

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}.$$

Partial derivatives

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} J(w)$$

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

$$= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

$$= \frac{1}{2} \sum_i 2\left(y^{(i)} - \phi(z^{(i)})\right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \phi(z^{(i)}) \right)$$

Partial derivatives

$$
\begin{aligned}
&= \sum_i \left(y^{(i)} - \phi(z^{(i)})\right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)}\right)\right) \\
&= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\left(-x_j^{(i)}\right) \\
&= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}
\end{aligned}
$$

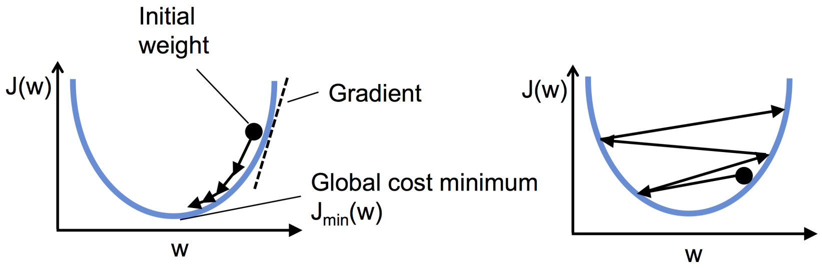
Adaline learning rule vs. Perceptron rule

- Looks (almost) identical. What is the difference?
- $\phi(z^{(i)})$ with $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)}$ is a real number
- And not an integer class label as in Perceptron
- The weight update is done based on *all* samples in training set
- Perceptron updates weights incrementally after each sample
- This approach is known as "batch" gradient descent

Adaline implementation

▸ iPython notebook on github

Lessons learned



- Learning rate too high: error becomes larger (overshoots global min)
- Learning rate too low: takes many epochs to converge
- Feature normalization (helps gradient descent learning to converge more quickly.)

Adaline + Feature Normalization implementation

▸ iPython notebook

Stochastic gradient descent (SGD)

- Large dataset with millions of data points ("big data")
- Batch gradient descent costly
- Need to compute the error for the entire dataset ...
- ... to take one step towards the global minimum!

$$\Delta \mathbf{w} = \eta \sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right) \mathbf{x}^{(i)}.$$

- Instead of updating the weights based on the sum of the accumulated errors over all samples (gradient descent), SGD updates the weights incrementally for each training sample

$$\Delta \mathbf{w} = \eta \left(y^{(i)} - \phi\big(z^{(i)}\big) \right) \mathbf{x}^{(i)}.$$

SGD details

- Approximation of gradient descent
- Reaches convergence faster because of frequent weight updates
- Each gradient is calculated based on a single training example
- Important to present data in random order to obtain satisfying results
- Also, we want to shuffle the training set for every epoch

SGD details

- Learning rate often gradually decreased (adaptive learning rate)
- In stochastic gradient descent implementations, the fixed learning is often replaced by an adaptive learning rate that decreases over time, for example, where c_1 and c_2 are constant:
$$\frac{c_1}{epochs + c_2}.$$
- Can be used for online learning

Online Learning

- Model is trained on the fly as new training data arrives
- Useful if we are accumulating large amounts of data, e.g., customer data in web applications
- The system can immediately adapt to changes and the training data can be discarded after updating the model
- SGD can be used for online learning

Mini-batch learning

- A mix between SGD and batch GD is known as *mini-batch learning*
  - Applies batch gradient descent to smaller subsets of the training data, for example, 50 examples at a time
  - Can use vector/matrix operations rather than loops as in SGD
  - Vectorized operations highly efficient
- Advantages over batch gradient descent:
  - Convergence is reached faster via mini-batches because of the more frequent weight updates
  - It allows us to replace the loop over the training samples in stochastic gradient descent with vectorized operations, which can further improve the computational efficiency of our learning algorithm

In Summary...

- Batch Gradient Descent (BGD)
  - Minimize a cost function by taking a step in the opposite direction of a cost gradient that is calculated from the whole training set.
  - Computationally quite costly.
- Stochastic Gradient Descent (SGD)
  - Sometimes also called iterative or online gradient descent.
  - Instead of updating the weights based on the sum of the accumulated erros over all samples $x^{(i)}$, we have:

$$\Delta \mathbf{w} = \eta \sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right) \mathbf{x}^{(i)}.$$

SGD implementation

▸ iPython notebook