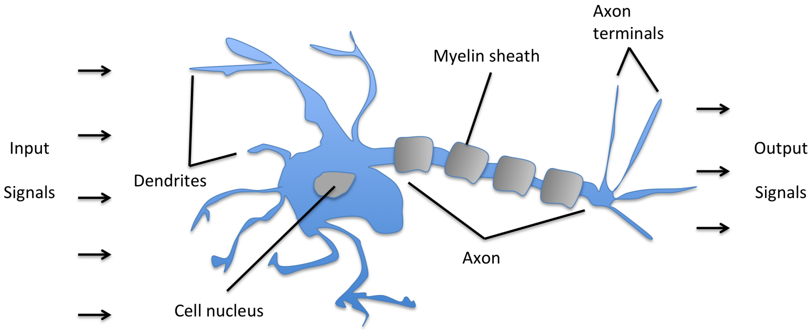
Chapter 2

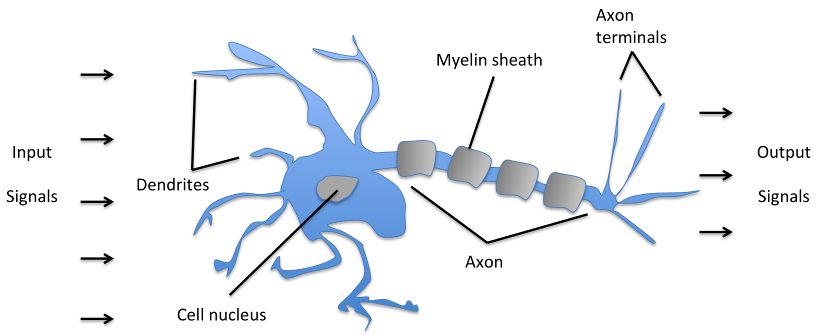
Training Machine Learning Algorithms for Classification

August 21, 2019

Biology


Input
Signals
Dendrites
Cell nucleus
Myelin sheath
Axon
terminals
Axon
Output
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

Rosenblatt Perceptron

- Binary classification task
- Positive class (1) vs. negative class (-1)
- Define activation function $\phi(z)$
- 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

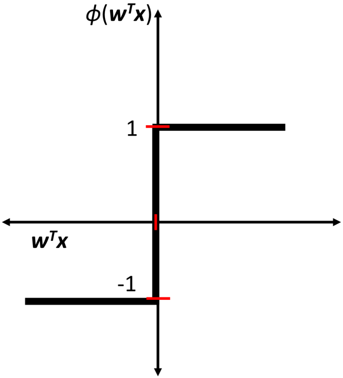
$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases} .$$

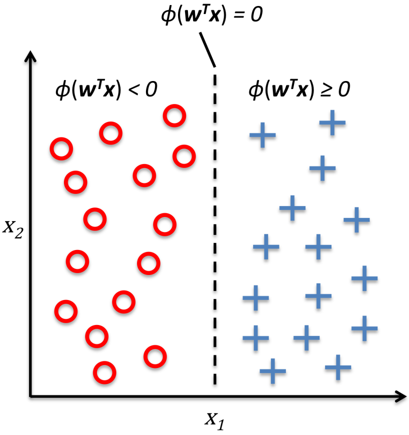
Vector dot product

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

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

Input squashed into a binary output





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 $\hat{y}^{(i)}$
    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, and $\hat{y}^{(i)}$ is the predicted class label.

Update rule examples

Correct prediction, weights unchanged:

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

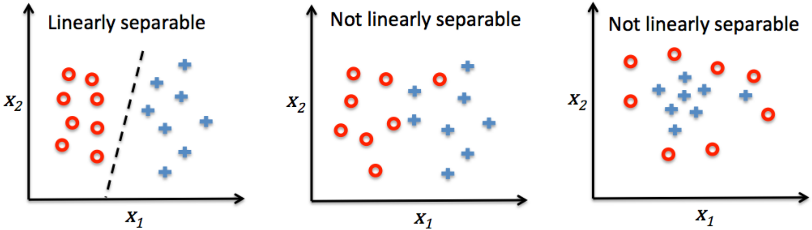
$$\Delta w_j = \eta\Big(1 - 1\Big)x_j^{(i)} = 0$$

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

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

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

Linear separability



Convergence

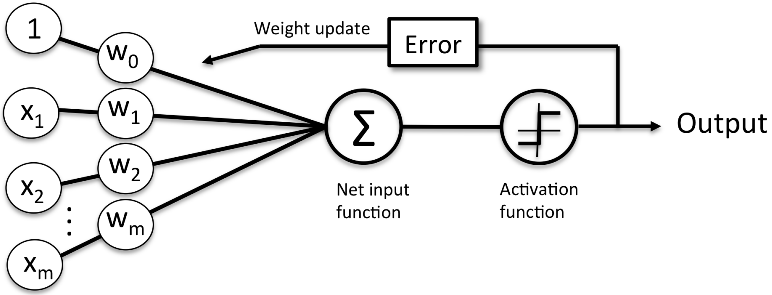
Convergence guaranteed if

- 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 implementation

iPython notebook on github

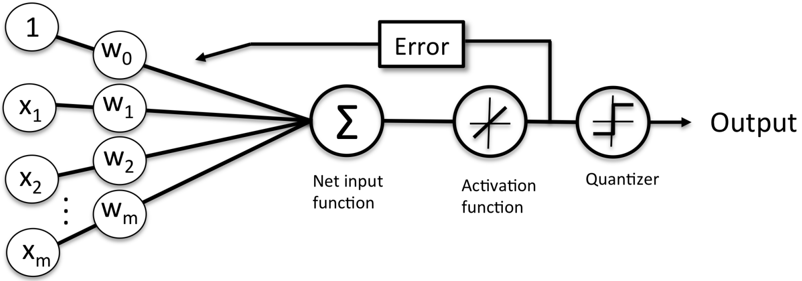
ADAPtive LInear NEuron (Adaline)

- Weights updated based on a linear activation function
- Remember that perceptron used a unit step function
- $\phi(z)$ 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

Adaline: notice the difference with perceptron



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)

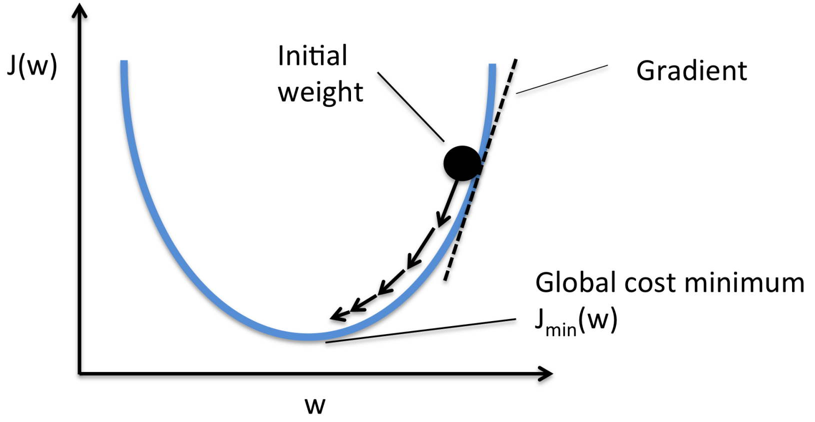
$$J(\mathbf{w}) = \frac{1}{2} \sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right)^2$$

Advantages of Adaline cost function

- The linear activation function is differentiable
- Why derivatives?
  - We need to know how much each variable affects the output!
- Unlike the unit step function
- It is convex
- Can use *gradient descent* to learn the weights

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.

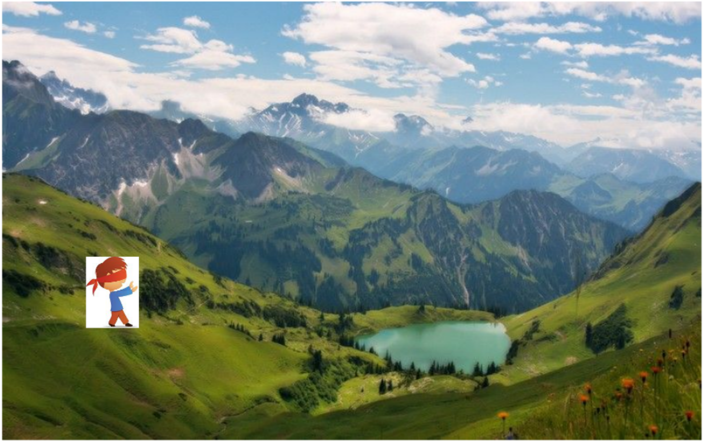

J(w)
Initial
weight
Gradient
Global cost minimum
$J_{min}(w)$
w

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.

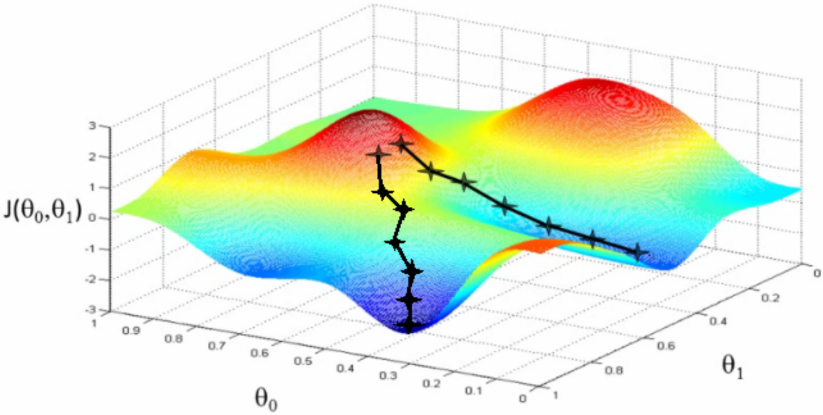
https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/

Gradient Descent: an intuition



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

Gradient Descent: an intuition



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 away from the gradient $\nabla J(\mathbf{w})$ of the cost function

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

- The weight change is defined as follows:

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

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,

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

Weight update of weight w_j

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

We update all weights simultaneously, so Adaline learning rule becomes

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

Partial derivatives

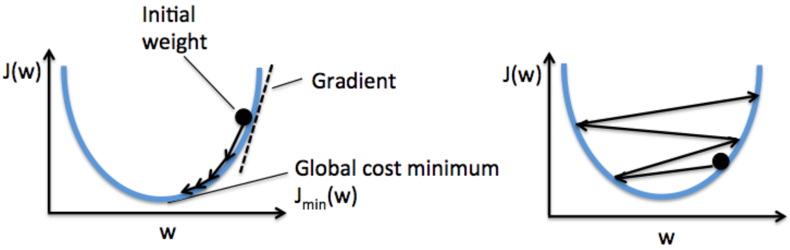
$$
\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right)^2 \\
&= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right)^2 \\
&= \frac{1}{2} \sum_i 2\big(y^{(i)} - \phi(z^{(i)})\big) \frac{\partial}{\partial w_j} \Big(y^{(i)} - \phi(z^{(i)}) \Big) \\
&= \sum_i \big(y^{(i)} - \phi(z^{(i)})\big) \frac{\partial}{\partial w_j} \Big(y^{(i)} - \sum_i \big(w_j^{(i)} x_j^{(i)} \big) \Big) \\
&= \sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right) \left(-x_j^{(i)} \right) \\
&= -\sum_i \left(y^{(i)} - \phi\big(z^{(i)}\big) \right) x_j^{(i)}
\end{aligned}
$$

- 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

Perceptron 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

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!

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

- SGD updates the weights incrementally for each training sample

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

SGD details

- Approximation of gradient descent
- Reaches convergence faster because of frequent weight updates
- Important to present data in random order
- Learning rate often gradually decreased (adaptive learning rate)
- Can be used for online learning
- Middle ground between SGD and batch GD is known as *mini-batch learning*
    - E.g. 50 examples at a time
    - Can use vector/matrix operations rather than loops as in SGD
    - Vectorized operations highly efficient