AIML Lecture 3 : Reading Material

1 The Neural Network Model

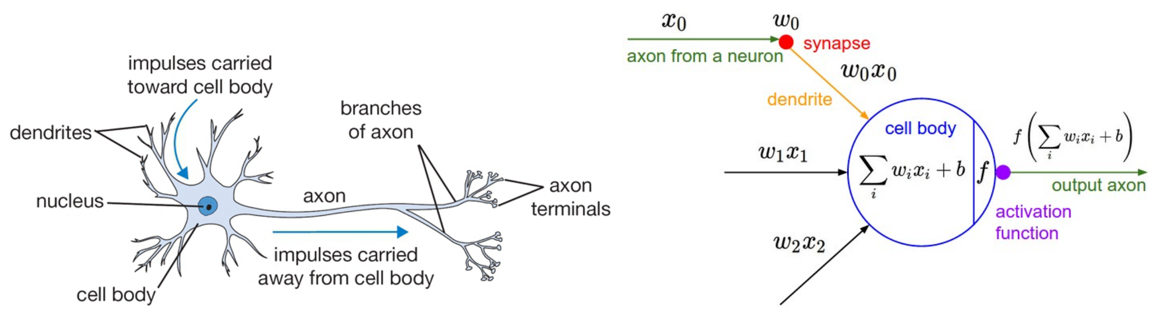


Figure 1: (Left) A biological neuron. (Right) An artificial neuron model.

A neuron is the basic computational unit of the brain. It has a sophisticated structure, with many parts (see figure above). However its working can be explained by simplifying it, and considering a mathematical model called the artificial neuron instead. The artificial neuron is defined by specifying an activation function and a set of parameters called weights (a vector $\mathbf{w}$) and bias (which is a real number b). It takes an input vector $\mathbf{x}$, and computes an output y (a real number) according to the formula $y = f\left(\sum_i w_i x_i + b\right)$.

It is similar to the linear classifier except having an activation function. The artificial neuron is studied using various activation functions like sign (1 for +ve, -1 for -ve inputs), sigmoid function ($1/(1+e^{-x})$) etc. The advantage of sigmoid function over the sign function is that it is differentiable, which is required for the gradient descent algorithm (which we will see later) to work properly. The activation function makes the artificial neuron a non linear function.

Though the artificial neuron was first proposed for studying the brain, it can also be used as a binary classifier in machine learning. Typically artificial neurons are used as a collection, but for this lecture we study only a single artificial neuron, which is similar to a linear classifier.

Recall that in the previous lecture, we said that linear classifier needs to be trained. That is the correct set of weights and biases needs to be found out so that the linear classifier “fits” the data. In the next two sections, we will look at two algorithms for doing this.

2 Perceptron Algorithm

The perceptron algorithm finds the parameters (weights $\mathbf{w}$ and bias b) of a linear classifier that can classify a dataset correctly. It is only guaranteed to work with the datasets that is linearly separable. That is there should exist some linear classifier that can separate the positive and negative examples. Let the supervised dataset be $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_n, y_n)\}$. The algorithm starts with a initial set of weights $\mathbf{w}(0)$, and keeps updating the weights after looking at each example. The resulting weights after each update will be denoted by $\mathbf{w}(1), \mathbf{w}(2), \cdots \mathbf{w}(n)$. The algorithm is as follows:

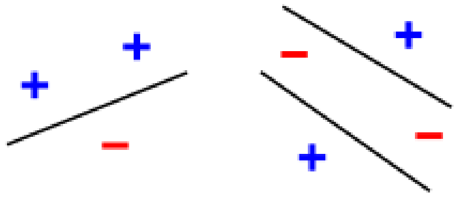


Figure 2: (Left) A linearly seperable dataset. (Right) A dataset that is not linearly seperable.

1. Start with the all zeros weight vector, $\mathbf{w}(0) = 0$ and set $t = 1$.
2. For $t = 1$ to n:
   (a) Compute $p = \mathbf{w}(t) \cdot \mathbf{x}_t$.
   (b) If $p \neq y_t$ (that is, the prediction is wrong), update as follows

$$\text{if y is 1 then } \mathbf{w}(t+1) = \mathbf{w}(t) + x(t) \text{ else } \mathbf{w}(t+1) = \mathbf{w}(t) - x(t)$$

The algorithm updates the weight vector only if the prediction is wrong.

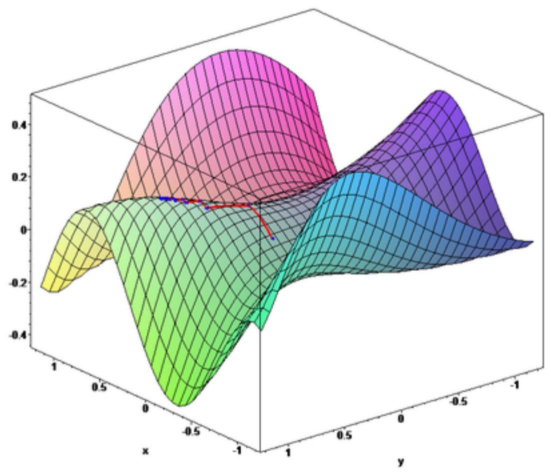
3 Gradient Descent

The gradient descent algorithm is a generalization of the perceptron algorithm, that works with a variety of models (not just linear classifiers). For understanding it, we need to introduce a loss function for the model. The loss function measures how bad the predictions of current model is (or how bad the current parameters of weights and bias are) compared to the ground truth labels in the training dataset. It is defined by first specifying a way measuring how bad the prediction is from the correct value for a single example. The loss for the training dataset is the summation of loss for each example.

For a training dataset $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_n, y_n)\}$, let $p_1, \cdots, p_n$ be the predictions of a linear classifier with weights $\mathbf{w}$ (that is $p_i = \mathbf{w} \cdot \mathbf{x}_i$). A simple loss for an example is the squared error $(y_1 - p_1)^2$. Hence the loss for the training dataset is given by the summation. The mean squared error is just this divided by number of examples n.

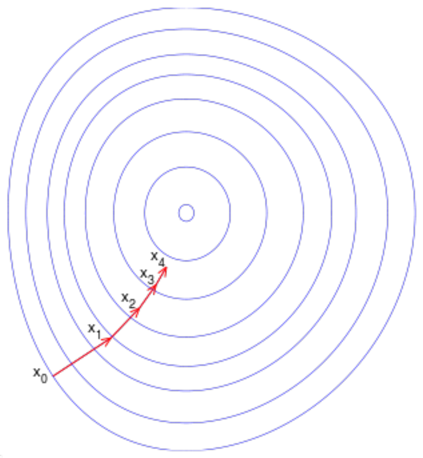
$$\text{MSE} = \sum_i (y_i - p_i)^2 / n.$$

Typically loss functions are positive valued and the loss is zero when the predictions match the ground truth values. Now the problem of finding the parameters $\mathbf{w}$ that "fits" the training data, is just the problem of minimizing the loss function. Such loss function minimization problems are also called optimization problems. We can plot the loss function when the weight are 2 dimensional, by taking w_1, w_2 as the x,y axes and the value of the loss to be the z axes. So this plot is in 3D, forms a surface called the loss surface.



A simple description of the gradient descent algorithm is as follows: Think of each of the weights $\mathbf{w}_i$'s as the value of a knob that you can tune. Move each knob slightly in each direction and see if the loss function (MSE) is decreasing. Then move the knobs to that direction such that the loss function decreases the most. Keep repeating this process, until we reach a stage, when no matter which direction any of the knob's are turned the loss function doesn't decrease. Such a state is called a Minima of the loss function. Note that if $\mathbf{w}$ gives the correct predictions in all the data, the MSE will be zero, and no matter how you change the parameters, it will not decrease.

The above description corresponds to moving in the loss surface by making small steps. The algorithm stops when we have reached a valley. For a real loss function computed on a dataset, the amount by which each knob needs to be turned is obtained by the partial derivative of MSE with respect the weight corresponding to the knob. This algorithm will find the correct $\mathbf{w}$ if there is only one valley (local minima). When there are multiple minima, we would ideally like to find the one which is the least among them (called global minima), however there is no guarantee that gradient descent will find it. But in practice, with many of the more powerful model a variant of gradient descent works well.



4 References

1. Neural Networks Course Notes / Blog
   http://cs231n.github.io/neural-networks-1/
   https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/
2. Gradient Descent
   http://cs231n.github.io/optimization-1/
3. Perceptron (with Math Proofs)
   https://www.cs.cmu.edu/~avrim/ML10/lect0125.pdf
4. Perceptron (slides)
   http://aass.oru.se/~lilien/ml/seminars/2007_02_01b-Janecek-Perceptron.pdf