Understanding & Developing Artificial Neural Network with Objective-C and Python

Deerfield Academy Advance Computer Science Research

Yongyang (Neil) Nie

Feb, 2016

Contents

Abstract:

Machine learning is a subset of artificial intelligence that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can change when exposed to new data.

There are many methods and algorithms for learning algorithms, from Support Vector Machine[1] to Artificial Neural Networks[2]. The purpose of them are similar, they can all be used to classify complex data, such as images and DNA samples.

Introduction:

There are two types of machine learning, one is supervised machine learning[3]. In this research, we will mainly focus on artificial neural networks and supervised machine learning. Supervised learning is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training

[1]https://en.wikipedia.org/wiki/Support_vector_machine

[2]https://en.wikipedia.org/wiki/Artificial_neural_network

[3]Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar (2012) *Foundations of Machine Learning*, The MIT Press ISBN 9780262018258.

examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal).

You will discover that machine learning is beautiful and can be very simple. We will be able to implement a simple algorithm that can recognize hand written digits in less than a hundred lines of code. Only a small amount of mathematical proves and equations will be covered. The paper will discuss the principles behind designing, building, and debugging an artificial neural network. The projects will be written mainly in Objective-C or Python. You can find the resources on Github.

Forward Feed:

The neural network will begin be taking in some inputs and making some predictions based on the inputs and the weights between nodes. We can think of it like a one by two matrix. There will be a weight connecting every input layer node to every output layer node, therefore, there are six weights between the input layer and the second layer. The matrix calculation should yield us some result

$$\begin{bmatrix}0\\1\end{bmatrix}\ [{w_0}^0\ {w_0}^1\ {w_0}^2\ {w_1}^0\ {w_1}^1{w_1}^2] = \begin{bmatrix}z\\z\\z\end{bmatrix}$$

If we give the matrices some names, call inputs x, weights $w_{(l)}$ and output $z_{(l)}$. In our case, l indicate the layer, for example, the first layer weights are $w_{(1)}$. $z_{(l)}$ represent the output matrix with layer l, in this case, the output z is the hidden layer output.

$$\mathrm{x}w_{(n)} = \ z_{(n)} \tag{1}$$

Afterwards, we have to apply an activation function[4]. The activation function that I used in this research and this paper will be the sigmoid function. This is a complete cycle, there is one more layer to go in order to yield a result. Note, in a multilayer neural network, this process will be repeated until we yield some output. In our case, we only have to do this twice. The activation function is shown as below.

[4]https://en.wikipedia.org/wiki/Activation_function

$$\sigma\left(z\right) = \ \frac{1}{1+e^{-z}}$$
$$\mathrm{a}_{(2)} = f(z_{(2)}) \tag{2}$$

By applying the same process as before, this time, the inputs will be $a_{(2)}$ *then,* $z_{(3)}$ will be our final result. This process is known as forward feed. It's relatively straightforward.

$$z_{(3)} = \ \ \mathrm{a}_{(2)}w_{(2)}$$

After we yield some result, we can more on and look at how far off we are from the expected result. We need some methods to quantify this and so adjustments to the network to minimize error.

$$\sigma'\left(z\right) = \ \sigma\left(x\right)\left(1-\sigma(z)\right)$$

Quantifying and Minimizing Cost:

Now the neural network can make calculation/predictions, however, the result is far from desired. In almost all learning algorithms, the input data cannot be altered, therefore the x term is constant in equation one. In order to change the output z the only option is to change the weights w.

First of all, we have to come up with ways to quantify the cost.

$$(3)\ C = \sum_j \frac{1}{2}(\hat{y}-y)^2 \tag{3}$$

C is the cost, which equals to the sum of all the differences between calculated result and actual result squared and times one half. We can take advantage of the equation derived above and substitute for some of the variable.

$$(4)\ C = \ \sum_J \frac{1}{2}\left(y - f\left(f\left(xw_{(1)}\right)w_{(2)}\right)\right)^2$$

Here we have it, a way to quantify the cost of the neural network. This function will be referred to as the cost function. Now, we have to

solve the problem, how do we minimize C, will brute force work? It turns out, no, because in a three-node neural network we have to compute more than a million possible weights, which will be gruesome.

We can think of the equation above as a function of cost in terms of all possible weights. There will be one set of weights that will bring the cost to the lowest. Then, this becomes a minimization problem.

Gradient Descent:

The best way to minimization the cost is to use gradient descent, a very fast and classic way to solve problems like this. In fact, gradient descent is widely used in math, image process and machine learning.

Gradient descent is a first-order iterative optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient (or of the approximate gradient) of the function at the current point.

Gradient descent is also known as steepest descent, or the method of steepest descent. The process can be seen as a ball rolling down a hill[5] and trying to find the lowest point. Note that actual physics doesn't apply here and we will define our own movement of the ball.

There are limitations to this method. First of all, what if we are stuck in a local minimum, our goal is to find the global minimum for the cost function. In another word, this method will not work for a non-convex function. In fact, this problem is solved in equation (3), by squaring the difference in $(\hat{y} - y)^2$, we transformed this function to a convex function[6]. Even though there are tens of thousands of variables (dimension), the function will still be convex, thus, we can apply gradient descent to find the global minimum.

There are other types and variation of gradient descent as well. One of the most commonly used one is Stochastic gradient descent (SGD), also known as incremental gradient descent, is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions. In other words, SGD tries to find minima or maxima by iteration. [7]

Back propagation:

With gradient descent, we can create a set of routines that can help us to change the weights of the network to minimize the cost.

Phase 1: Propagation: Each propagation involves the following steps:

1. Forward propagation of a training pattern's input through the neural network in order to generate the network's output value(s).
2. Backward propagation of the propagation's output activations through the neural network using the training pattern target in order to generate the deltas (the difference between the targeted and actual output values) of all output and hidden neurons.

Phase 2: Weight update: For each weight, the following steps must be followed:

1. The weight's output delta and input activation are multiplied to find the gradient of the weight.
2. A ratio (percentage) of the weight's gradient is subtracted from the weight.

Backpropagation is based around four fundamental equations. Together, those equations give us a way of computing both the error $\delta^L$and the gradient cost of the function. In fact, the backpropagation equations are so rich that understanding them well requires considerable times and patience. [8]

An equation for the error in the output layer δ^L

$$\delta_J^L = \frac{\partial C}{\partial a_J^L}\sigma'(z_J^L) \tag{BP1}$$

[5]https://iamtrask.github.io/2015/07/27/python-network-part2/ by Andrew

[6]Proof and definition of convex functions: http://mathworld.wolfram.com/ConvexFunction.html

[7]http://www.mit.edu/~dimitrib/Incremental_Survey_LIDS.pdf Dimitri P. Bertsekas Report LIDS - 2848

[8]The equations were created by Michael A. Neilson "Neural Networks and Deep Learning", Determination Press, 2015

The first term on the right, $\frac{\partial J}{\partial a_J^L}$ measures how fast the cost is changing as a function of j$^{\text{th}}$ output activation. Everything in (BP1) is easily calculated. We computed z_J^L while computing the behavior of the network. Depending on the cost function, in our case, the quadratic cost function is relatively easy to compute.

Equation (BP1) is a perfectly good expression, however, it's not matrix based, form that backpropagation desires. The fully matrix form becomes.

$$\delta^L = (a^l - y)(\sigma'(z^l) \tag{4}$$

An equation for the error δ^l in terms of the error in the next layer, δ^{l+1} in particular:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \bigodot \sigma'(z^l) \tag{BP2}$$

where $(w^{l+1})^T$ is the transpose of the weight matrix (w^{l+1}) for the *(l+1)th* layer. Suppose we know the error δ^{l+1} at the *(l+1)th* layer. When the transpose weight matrix is applied $(w^{l+1})^T$, we can think of this as moving the error backward through the network, giving us some sort of measure of the error at the output of the l$^{\text{th}}$ layer. Finally, we take the Hadamard product $\bigodot \sigma'(z^l)$.

By combining (BP1) with (BP2), the error δ^l for any layer in the network can by computed. We start by using (BP1) to compute δ^l, then apply Equation (BP2) to compute δ^{L-1}, then Equation (BP2) again to compute δ^{L-1}, and so on, all the way back through the network.

An equation for the rate of change of the cost with respect to any bias in the network

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{BP3}$[9]$$

This is the error δ_j^l is exactly equal to the rate of change $\frac{\partial C}{\partial b_j^l}$. This equation can be rewritten as

$$\frac{\partial C}{\partial b} = \delta$$

An equation for the rate of change of the cost with respect to any weights in the network

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{BP4}$[10]$$

This equation can help us to compute the partial derivative $\frac{\partial C}{\partial w_{jk}^l}$ in terms of the quantity δ^l and a^{l-1}. It can be rewritten in a less index-heavy notation:

$$\frac{\partial C}{\partial w} = a_{in}\delta_{out} \tag{5}$$

Backpropagation with simple neural network example

In a simple three-layer neural network, below are the four equations that we derive with the principle of gradient descend that will help us minimize the error.[11]

$$\delta_{(3)} = -(y - \hat{y})(\sigma'(z_{(3)}))$$

Calculate the δ for the third layer of the network.

$$\frac{\partial J}{\partial w_{(2)}} = (a^{(2)})^T \delta_{(3)}$$

Use the calculated result to help us modify the second layer of weights in the network.

$$\delta_{(2)} = \delta_{(3)} (w_{(2)})^T \sigma'(z_{(2)})$$

Calculate the δ for the second layer of the network.

$$\frac{\partial J}{\partial w_{(1)}} = x^T \delta_{(2)}$$

[9]The equation was reference from Michael A. Neilson "Neural Networks and Deep Learning", Determination Press, 2015

[10]The equation was referenced from Michael A. Neilson "Neural Networks and Deep Learning", Determination Press, 2015

[11]The equation was referenced from Michael A. Neilson "Neural Networks and Deep Learning", Determination Press, 2015

Finally, we can modify the first layer of weights in the network. This process is often repeated until the accuracy of the network reaches a threshold.

Examples and Applications:

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Overfitting and Regularization:

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.