

Notes on Machine learning

Matthew

March 1, 2019

1 Basics - Neural Networks

1.1 Math of neural networks

The cost of a neural network is effectively how wrong its output is from the desired output. The cost function can be calculated in a number of ways, but one of the most common is a variation of the mean square error:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

Change in C, where C is a function of multiple variables (in this example equation, just two variables v1 and v2, but there can be an unlimited amount) is modelled approximately by:

$$\Delta C \approx \frac{\delta C}{\delta v_1} \Delta v_1 + \frac{\delta C}{\delta v_2} \Delta v_2$$

The gradient vector of C (again, in this example a function of just two variables but it can be of many) is equivalent to:

$$\nabla C \equiv \left(\frac{\delta C}{\delta v_1}, \frac{\delta C}{\delta v_2} \right)^T$$

Therefore a change in C can be approximated by $\Delta C \approx \nabla C \cdot \Delta v$. As we are trying to minimise the cost value of C, we can fix Δv to a value that ensures ΔC will always be negative:

$$\Delta v = -\eta \nabla C$$

Where η is a small positive value known as the learning rate.

This means that:

$$\begin{aligned}\Delta C &= \nabla C \cdot \Delta v \\ &= \nabla C \cdot -\eta \nabla C \\ &= -\eta \|\nabla C\|^2\end{aligned}$$

i.e. the change in C can always be negative, towards the local minima. The speed of gradient descent is now:

$$v \rightarrow v' = v - \eta \nabla C$$

To ensure that the ΔC approximation is accurate, we need to choose a value for η that is small enough to ensure that ΔC is not positive (i.e. the cost will increase) but not too small that the training will take an inordinate amount of time.

The idea is to use gradient descent to minimise the cost value (there are other alternatives that are being researched that could more efficiently reduce the cost value, but this is the most basic). Gradient descent has the ability to go wrong, but it often works extremely well as a powerful way of minimizing the cost function.

In a neural network, gradient descent is used to find the optimal values of the weights w_k and the biases b_l which minimize the cost of the network (these are the values v of the previous equations). We can now redefine the gradient descent update rule in terms of these variables rather than in v :

$$\begin{aligned}w_k \rightarrow w'_k &= w_k - \eta \frac{\delta C}{\delta w_k} \\ b_l \rightarrow b'_l &= b_l - \eta \frac{\delta C}{\delta b_l}\end{aligned}$$

One of the challenges faced by gradient descent is time. One aspect of this is the fact that the cost function has the form:

$$C = \frac{1}{n} \sum_x C_x$$

i.e it is an average across all computed cost values of the input x (all training examples). This is a very costly process of calculating and averaging gradients for all training examples, especially for a large value of x .

To solve this problem, an approach called *stochastic gradient descent* can be used. In this process, a random *mini-batch* of training examples are selected and used to calculate the gradient. If the examples (say a count of

m from x in the total training set) are random, use of the batch can be a good approximation for the set calculated in a much shorter time. Provided this sample size is large enough:

$$\frac{\sum_{j=1}^m \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$$

where the second sum is the sum over all training data.

This approach can be applied to the neural network for changing the values of the weights and biases based on a small random batch. The network is trained on these mini-batches one at a time until the training set is exhausted (concluding an *epoch* of training), then we start again.

1.2 Implementing in python

Weights and biases are initialised using a Gaussian distribution random number generator in numpy, into arrays of the correct size. Biases are initialised an array of one-dimensional arrays (only bias needed per node, and none for the input layer), and weights are initialised into an array of 2-d arrays based on the number of nodes in adjacent layers. The notation for the weights matrices is that the `net.weights[n]` matrix (if we denote it w) contains values w_{jk} such that they are the weight for the connection between the k^{th} neuron in the $n + 1^{th}$ layer and the j^{th} neuron in the n^{th} layer.

The advantage of this notation is that the vector of activations of the n th layer of neurons is:

$$a_n = \sigma(wa_{n-1} + b)$$

This equation computes the activation of a layer based on a combination of the previous layer's activations, the weights connecting the two layers and the biases of the layer being calculated. The σ function is then applied to each of the calculated activation values (*vectorizing* the function σ).

2 Backpropagation

2.1 Computing the output from a neural network

Before learning about backpropagation, need to compute output quickly from the network. This is done using matrices.

Need an unambiguous way to refer to the nodes and connections in the network (i.e the weights and biases). We will do this with the notation w_{jk}^l to denote the weight of the connection between the j^{th} neuron of the l^{th} layer and the k^{th} neuron in the $(l - 1)^{th}$ layer. Remember, the reason for

the ordering of the j and k indices is to aid with the matrix computations, despite their order being counterintuitive with the input neuron second.

A similar notation is used for the bias notation, b_j^l for the bias of the j^{th} neuron in the l^{th} layer. And the same for activation, a_j^l for the activation of the j^{th} neuron in the l^{th} layer.

With this notation, the relation of the activation of the j^{th} neuron in the l^{th} layer, a_j^l , is computed from activations in the $(l-1)^{th}$ layer by the equation

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

where the sum is over all neurons k in the $(l-1)^{th}$ layer.

To rewrite this expression in matrix form we define a weight matrix w^l for each layer l . The entries in w^l are just the weights of the connections to the l^{th} layer of neurons. Similarly we define a bias vector b^l for each layer of neurons, and an activation vector a^l .

With these notations in mind we can rewrite the activation function as

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

which is a much simpler way of displaying the function and helps us relate the entire layer to the previous weights, biases and activations.

The computed value

$$w^l a^{l-1} + b^l$$

is used often enough to assign it the notation z^l such that

$$a^l = \sigma(z^l)$$

This is known as the weighted input to the layer l . Also note that the components of z^l are

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

such that z_j^l is the weighted input to the activation function in layer l of neuron j

2.2 Assumptions about the cost function

1. Cost function can be written as an average $C = \frac{1}{n} \sum_x C_x$ over cost function C_x for individual training examples x . This assumption holds for both the quadratic cost function we use and all other cost functions. This lets us compute the partial derivative $\frac{\delta C}{\delta w}$ and $\frac{\delta C}{\delta b}$ by averaging over all the training examples, where the partial derivatives are computed by the backpropagation algorithm.

2. The Cost can be written as a function of the outputs from the neural network. This assumption is held by the quadratic cost function, with the cost being determined by the sum of the square of the differences in the output activations and the desired output. Although it could be said that the cost is also a function of the desired output y , it makes more sense to say that C is merely a function of the outputs alone as y is fixed parameter that helps define the function.

2.3 The Hadamard product, $s \odot t$

An uncommon linear function is the *elementwise* product of two vectors of the same dimension. This is denoted by $s \odot t$ where the components of the operation are $(s \odot t)_j = s_j t_j$. For example:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

This is often referred to as the Hadamard Product.