A BRIEF INTRODUCTION TO NEURAL NETWORKS AND BACK PROPAGATION
-
TALK FOR MAIML@DU

Evans Hedges

Abstract. This talk will cover a brief overview of gradient descent, neural networks, and the back propagation method of computing a gradient for a given neural network. We will not go into detail on the general capabilities or limitations of neural networks, but we will outline the structure of such models and their inspiration.

Talk given on 07Nov2022 at the University of Denver for the Mathematics of Artificial Intelligence and Machine Learning seminar.

1. Goals of Machine Learning and Cost Functions

In Machine Learning we typically have some sort of loss or cost function that we are seeking to minimize. This could be something like mean squared error, binary cross entropy, or any other loss function of your choice. In any case, the primary goal is to find the parameters of our model that minimize the total cost across our training dataset.

I.e., for some cost function $C : \mathbb{R}^d \to \mathbb{R}$, we are attempting to find

$$\min_{\vec{v} \in \mathbb{R}^d} C(\vec{v})$$

Where $\vec{v}$ is a vector that describes the parameters of our model.

As we have seen in previous talks we can sometimes find this minimum analytically. For example, a standard linear regression model may be solved by:

$$\vec{v} = (A^T A)^{-1} A^T \vec{y}$$

However, in general we may not have a closed form solution to find this minimum. A tool to address this that we will discuss today is gradient descent.

Date: 07 November 2022.

2. Gradient Descent

For gradient descent I will first outline an example and then I will describe a general algorithm to implement this approach.

Consider some continuous and differentiable $f : \mathbb{R} \to \mathbb{R}$ that we know to have a global minimum. We are allowed to compute values of f and values of its derivative. How would we approach finding the minimum?

One approach is to brute force guess. We're looking at $\mathbb{R}$ here so this is probably not a great idea, even if we know our solution should be "close" to 0. The next option I'm going to suggest since we have access to f' is the following.

First make a guess x_0, then look at $f'(x_0)$. If we have $f'(x_0) > 0$ then we know if we let $x_1 = x_0 - \epsilon$, we should have $f(x_1) < f(x_0)$. Similarly if $f'(x_0) > 0$ we let $x_1 = x_0 + \epsilon$. We can proceed from here progressively getting closer to some minimum (hopefully).

Let's quickly look at a toy example, $f(x) = x^2$. (Board work here)

DRAW THIS

Clearly we have only talked about $\mathbb{R}$, but in general we are in $\mathbb{R}^d$ for some d looking for a minimum here. Luckily this isn't a problem when we have gradients of our cost function. Here we need a starting point $x_0 \in \mathbb{R}^d$, and a "learning rate" $\epsilon > 0$. We then inductively construct better and better guesses by the following algorithm:

$$x_{n+1} = x_n - \epsilon \nabla C(x_n)$$

By definition we are descending along the steepest gradient of our cost function to our next x_{n+1}.

Clearly this algorithm has issues/questions related to our learning rate ϵ, how many steps we should take, whether or not we find the global minimum vs a local minimum, etc. But these questions are outside the scope of this talk. The important thing here is that gradient descent is a tool that uses the gradient of our cost function to find a set of parameters that minimize our cost.

3. Neural Networks

Neural Networks are inspired by how neurons work (hence the name) in animals. Typically neurons fire an all or nothing signal to the next neuron and the signal is fired once a certain threshold is crossed in their activation.

For Artificial Neural Networks (ANNs), in general they can be represented as a combination of: a weighted directed graph, a collection of biases, and an activation function. The weighted directed graph represents the neuronal structure of the system, the activation function is a generalization of the "activation" of a biological neuron which allows us to do more interesting things than binary "go/no-go" decisions on a per-neuron level, and the weights and biases associated with each edge/neuron are the things that we alter to change our model.

I will now outline the general structure of a neural network. *Draw diagram on board*

Example Diagram.jpg

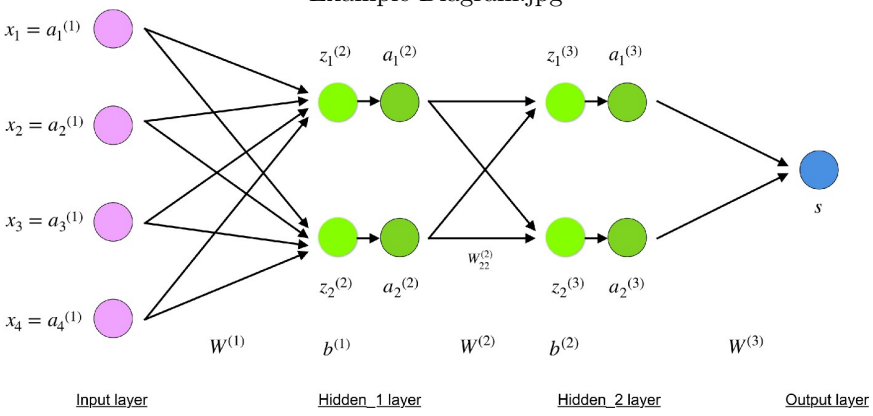

Figure 1. Our ANN Example

In general we can represent an ANN with the following items:

$$\text{Weight matrix entering layer } \ell: \quad W^\ell$$

Where $W^\ell_{i,j}$ is the weight connecting the ith neuron in the $\ell-1$th layer to the jth neuron in the ℓth layer.

$$\text{Bias vector: } b^\ell$$

Where b^ℓ_j is the bias of the jth neuron in the ℓth layer.

$$\text{Activation vector: } a^\ell$$

Where a^ℓ_j is the activation of the jth neuron in the ℓth layer.

$$\text{Weighted Input: } z^\ell = W^\ell a^{\ell-1} + b^\ell$$

Where z^ℓ_j is the input into the jth neuron in the ℓth layer.

And we get our next activation by

$$\text{Activation: } a^\ell = \sigma(z^\ell) = \sigma(W^\ell a^{\ell-1} + b^\ell)$$

Where σ is applied coordinate-wise and σ is called our "activation function".

As you can see by how the input/activations are defined, we must start with $\ell = 0$ and compute the activation of each layer one at a time, using what we call "feed forward" to feed the activations forward through the neural net.

4. Computing Gradients

When we want to train our model we need some cost function. For the purposes of our talk today we'll just use MSE as our example and for simplicity we will compute our gradient one datapoint at a time. So, given a data point $(x, y) \in \mathbb{R}^d \times \mathbb{R}^n$, our cost is

$$C(x, y) = ||y - a^L(x)||^2$$

Where L is our output activation layer.

When training our model we have control over the weights connecting our neurons and the biases of each neuron, we're assuming that the structure of the network is fixed (i.e. we don't add neurons or layers etc.). So to perform gradient descent we need:

$$\frac{\partial C}{\partial w^\ell_{jk}} \text{ and } \frac{\partial C}{\partial b^\ell_j}$$

For every weight and bias. Since, remember, we will update these parameters by:

$$w^\ell_{jk} += -\epsilon \frac{\partial C}{\partial w^\ell_{jk}}$$

Where ϵ is our "learning rate."

One thing we could do is go and compute all our gradient functions using chain rule, but computing this for our first layer would require computing:

$$\frac{\partial C}{\partial w^1} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \cdots \cdot \frac{\partial z^1}{\partial w^1}$$

Which is a lot of chain rule if we approach this naively. If you notice though, if you start from the end of the network, your gradients are easier to compute and then you can just plug them in to the previous layer. This is essentially what back propagation is doing, using a clever order to compute gradients that minimizes the work we need to do.

5. Back Propagation

Now it's time for some Chain rule and setting up of equations.

$$\text{Chain Rule: } \frac{\partial C}{\partial w^\ell} = \frac{\partial C}{\partial z^\ell}\frac{\partial z^\ell}{\partial w^\ell}$$

$$\text{Chain Rule: } \frac{\partial C}{\partial b^\ell} = \frac{\partial C}{\partial z^\ell}\frac{\partial z^\ell}{\partial b^\ell}$$

Let's first tackle the easy parts. Remember:

$$z^\ell = W^\ell a^{\ell-1} + b^\ell$$

So taking derivatives we see

$$\frac{\partial z^\ell}{\partial W^\ell} = a^{\ell-1}$$

$$\frac{\partial z^\ell}{\partial b^\ell} = 1$$

Lastly we have a common term, we will denote that

$$\delta^\ell = \frac{\partial C}{\partial z^\ell}$$

We now again use a nice chain rule here to see:

$$\delta^\ell = \frac{\partial C}{\partial z^\ell} = \frac{\partial C}{\partial z^{\ell+1}} \cdot \frac{\partial z^{\ell+1}}{\partial a^\ell} \cdot \frac{a^\ell}{\partial z^\ell}$$

$$\delta^\ell = \delta^{\ell+1} \cdot \frac{\partial z^{\ell+1}}{\partial a^\ell} \cdot \frac{a^\ell}{\partial z^\ell}$$

Now we find the other parts of this equation:

$$\frac{\partial z^{\ell+1}}{\partial a^\ell} = w^{\ell+1}$$

$$\frac{\partial a^\ell}{\partial z^\ell} = \sigma'(z^\ell)$$

We can finally put this all together to see

$$\delta^\ell = (W^{[\ell+1]^T} \cdot \delta^{\ell+1}) \odot \sigma'(z^\ell)$$

Where $\odot$ is the Hadamard product of element-wise multiplication.

Let's bring this all together to see our algorithm. First remember what we have found:

$$\frac{\partial C}{\partial w^\ell} = \delta^\ell a^{\ell-1}$$
$$\frac{\partial C}{\partial b^\ell} = \delta^\ell$$
$$\delta^\ell = (W^{[\ell+1]^T} \cdot \delta^{\ell+1}) \otimes \sigma'(z^\ell)$$

So we start with our final layer L and compute δ^L normally. We then update our model as follows:

$$W^L += -\epsilon \delta^L a^{L-1}$$
$$b^L += -\epsilon \delta^L$$

We then compute

$$\delta^{L-1} = (W^{[L]^T} \cdot \delta^{L1}) \otimes \sigma'(z^{L-1})$$

We can now inductively proceed updating each layer as we go backwards.

6. Some nice sources

Great medium article that covers this topic in sufficient detail: Link
Source for the diagram in these notes: Link
A nice set of notes from J.G. Makin at Cornell: Link
Another set of notes from Kevin Clark at Stanford: Link