Neural Networks

Otis Saint

November 2022

Contents

1 Introduction

2 The Perceptron

We begin by defining a single perceptron, which is the smallest unit of a neural network.

Definition 1 (Perceptron)**.** *A perceptron can be viewed as a mapping $P : \mathbb{R}^n \to \mathbb{R}$. Given a set of weights, $\vec{w}$, an input vector, $\vec{x}$, and a bias, b, and outputs the value*

$$\hat{y}(\vec{x}) = \sigma(\vec{w} \cdot \vec{x} + b)$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the ***activation function****, usually Relu, Sigmoid, Softmax,...etc.*

We often write $a = \vec{w} \cdot \vec{x} + b$, allowing the above equation to be written in a more simple manner

$$\hat{y}(\vec{x}) = \sigma(a)$$

2.1 Training the perceptron

The training of a perceptron, put simply, works as follows:

Given a set of inputs and target outputs $A = \{(\vec{x_i}, y_i) : i = 0, ..., N)\}$

- Calculate $\hat{y}_i$ for an input $\vec{x}_i$
- Calculate the error/loss function $\mathcal{L}(\hat{y}_i, y_i)$.
- Update weights and biases following **gradient descent** in order to minimise the error

For the purposes of this document, we will use the loss function

$$\mathcal{L}(\vec{y}, \hat{\vec{y}}) = \frac{1}{2r} \sum_{j=1}^{r} (\hat{y}_j - y_j)^2,$$

which, in the case of a single perceptron, is simply

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2.$$

Gradient descent works as follows: each of our parameters $\theta \in \{w_i, b\}$, will be updated according to

$$\theta^{n+1} = \theta^n - \alpha \partial_{\theta^t} \mathcal{L},$$

where $\alpha \in (0, 1]$ is called the **learning rate** of the perceptron.

Lets now calculate the partial derivatives. Using the chain rule, it easy to obtain

$$\partial_{w_k} \mathcal{L} = \partial_{\hat{y}} \mathcal{L} \; \partial_{w_k} \hat{y} = (\hat{y} - y) \; \partial_{w_k} \sigma \Big(\sum_i w_i x_i + b \Big) = (\hat{y} - y) \; \sigma'(a) \; \partial_{w_k} \Big(\sum_i w_i x_i + b \Big).$$

Now, $\forall i \neq k, \partial_{w_k}(w_i x_i + b) = 0$, and hence

$$\partial_{w_k} \mathcal{L} = (\hat{y} - y) \; \sigma'(a) \partial_{w_k}(w_k x_k + b) = (\hat{y} - y) \; \sigma'(a) \; x_k$$

With this, we obtain the equation for updating the weights of the perceptron

$$w_k \leftarrow w_k - \alpha \; (\hat{y} - y) \sigma'(a) x_k$$

. Carrying out a similar equation for the biases will lead to

$$b \leftarrow b - \alpha(\hat{y} - y)$$

3 Multi-Layer Perceptron Networks

With a basic understanding of how a single perceptron is defined, and trained, we can now begin to tackle **multi-layer perceptrons**, or MLPs.

We will assume a fully connected network with M inputs, N hidden layers, each with K neurons (i.e $N \times K$ neurons in the hidden layers) and P output neurons. In this network, we have $N + 1$ true neuron layers (since the input neurons do not have weights, bias, or activation function).

Before jumping in to the calculations, we first need to upgrade our notation from the previous section. For a single perceptron, the weights are given by a single vector of values, and the bias is simply a scalar. For a single layer of perceptrons in our new network however, we require that each perceptron in the layer has its own unique set of weights and a bias.

We will denote the weights connecting layer $k - 1$, to layer k by, $w_{ij}^{[k]}$. The i^{th} output of layer k will be denoted by $x_i^{[k]}$, and the corresponding bias will be written $b_i^{[k]}$. We will also assume that the activation function for the hidden layers is given by $\phi : \mathbb{R} \to \mathbb{R}$, and the activation function for the output layers is $\phi_0 : \mathbb{R} \to \mathbb{R}^+$.

With notation now defined, we can begin writing some equations!

3.1 Feed-forward

Similar to the equation for a single perceptron, we can write the output for an entire layer as

$$x_j^{[k]} = \phi\Big(\sum_i w_{ij}^{[k]} x_i^{[k-1]} + b_j^{[k]}\Big) = \phi\big(a_j^{[k]}\big). \tag{1}$$

Note that at the output layer we have $k = k^* = N + 1$, and the output of the network, $\hat{y}_j = x_j^{[k^*]}$, is given by

$$\hat{y}_j = \phi_0\Big(\sum_{i=1}^{P} w_{ij}^{[N+1]} x_i^{[N]} + b_j^{[N+1]}\Big). \tag{2}$$

One can then calculate the error using

$$\mathcal{L}(\hat{\vec{y}}, \vec{y}) = \sum_{j=1}^{P} (\hat{y}_j - y_j)^2$$

3.2 Backpropogation

The learning process for a multi-layer perceptron network is pretty much identical to that of the single perceptron. The main difference is that due to having multiple layers, in order for the entire network to learn, the error must be sequentially propogated backwards between layers. This idea will become clearer as we get in to the equations.

Each parameter of out network gets updated in the same way as a single perceptron, i.e

$$\theta^{t+1} = \theta^t - \alpha \partial_{\theta^t} \mathcal{L},$$

however, in the context of an MLP, we now have $\theta \in \{w_{ij}^{[k]}, b_i^{[k]}\}$.

Again, we will begin by calculating the partial derivatives for the weights $w_{ij}^{[k]}$. We have by the chain rule

$$\partial_{w_{ij}^{[k]}} \mathcal{L} = \partial_{a_j^{[k]}} \mathcal{L} \, \partial_{w_{ij}^{[k]}} a_j^{[k]} = \delta_j^{[k]} \partial_{w_{ij}^{[k]}} a_j^{[k]},$$

where

$$a_j^{[k]} = \sum_l w_{lj}^{[k]} x_l^{[k-1]} + b_j^{[k]}.$$

Looking at the partial derivative term, we thus have

$$\partial_{w_{ij}^{[k]}} a_j^{[k]} = \partial_{w_{ij}^{[k]}} \sum_s w_{sj}^{[k]} x_s^{[k-1]} + b_j^{[k]} = x_i^{[k-1]}.$$

Hence, plugging this back into our previous equation yields

$$\partial_{w_{ij}^{[k]}} \mathcal{L} = \delta_j^{[k]} x_i^{[k-1]} \tag{3}$$

3.2.1 Calculating the deltas

In the last section we defined

$$\delta_j^{[k]} = \partial_{a_j^{[k]}} \mathcal{L}.$$

Using the chain rule once again, we obtain

$$\partial_{a_j^{[k]}} \mathcal{L} = \sum_l \partial_{a_l^{[k+1]}} \mathcal{L} \, \partial_{a_j^{[k]}} a_l^{[k+1]}.$$

Notice that the first term in this equation is, by definition, $\delta_i^{[k+1]}$, and hence

$$\partial_{a_j^{[k]}} \mathcal{L} = \sum_l \delta_l^{[k+1]} \partial_{a_j^{[k]}} a_l^{[k+1]}$$

Again, looking at the partial derivative term, we have

$$\partial_{a_j^{[k]}} a_l^{[k+1]} = \partial_{a_j^{[k]}} \sum_h w_{hl}^{[k+1]} x_h^{[k]} + b_l^{[k+1]} = \partial_{a_j^{[k]}} \sum_h w_{hl}^{[k+1]} \phi(a_h^{[k]}) + b_l^{[k+1]}$$

setting $h \to j$ and removing the summation (since the partial derivative vanishes for all terms with $h \neq i$), then gives

$$\partial_{a_j^{[k]}} a_l^{[k+1]} = \partial_{a_j^{[k]}} \left[w_{jl}^{[k+1]} \phi(a_j^{[k]}) + b_l^{[k+1]} \right] = w_{jl}^{[k+1]} \phi'(a_j^{[k]})$$

Thus, we obtain an expression for $\delta_j^{[k]}$

$$\delta_j^{[k]} = \phi'(a_j^{[k]}) \sum_l w_{jl}^{[k+1]} \delta_l^{[k+1]}. \tag{4}$$

This now shows how the error is propogated backwards in the network, since the deltas for layer k is dependent on the deltas for layer $k+1$. Of course, one cannot use this formula without first knowing the deltas for the output layer, which can easily be shown to be

$$\delta_j^{[N+1]} = \delta_j^* = \sum_l (\hat{y}_l - y_l) \phi_0' \big(a_l^{[N+1]}\big) \tag{5}$$

We will now proceed put everything together to obtain the **governing equations for an MLP**.

3.2.2 The governing equations of MLPs

With the calculations complete, we can now write down the governing equations. For the forward feed we have

$$\hat{y}_j = \phi_0\Big(\sum_{i=1}^{P} w_{ij}^{[N+1]} x_i^{[N]} + b_j^{[N+1]}\Big), \tag{6}$$

$$x_j^{[k]} = \phi\Big(\sum_{i} w_{ij}^{[k]} x_i^{[k-1]} + b_j^{[k]}\Big) = \phi\big(a_j^{[k]}\big), \tag{7}$$

and for the backpropogation, we have

$$\partial_{w_{ij}^{[k]}}\mathcal{L} = \delta_j^{[k]} x_i^{[k-1]}, \tag{8}$$

$$\partial_{b^{[k]_j}}\mathcal{L} = \delta_j^{[k]}, \tag{9}$$

where

$$\delta_j^{*} = \sum_{l} (\hat{y}_l - y_l)\phi_0'\big(a_l^{[N+1]}\big), \tag{10}$$

$$\delta_j^{[k]} = \phi'(a_j^{[k]}) \sum_{l} w_{jl}^{[k+1]} \delta_l^{[k+1]}. \tag{11}$$

$$\tag{12}$$

Our parameters are then updated in the following way

$$w_{ij}^{[k]} \leftarrow w^{[k]_ij} - \alpha\partial_{w_{ij}^{[k]}}\mathcal{L} \tag{13}$$

$$b_j^{[k]} \leftarrow b^{[k]_j} - \alpha\partial_{b_j^{[k]}}\mathcal{L} \tag{14}$$

4 Convolutional Neural Networks

5 Using the Neural framework in Python