Econ 425 Week 7

Manu Navjeevan

February 16, 2021

1 Neural Net Review

As we went over last time, a neural network consists of nodes and edges, arranged into layers.

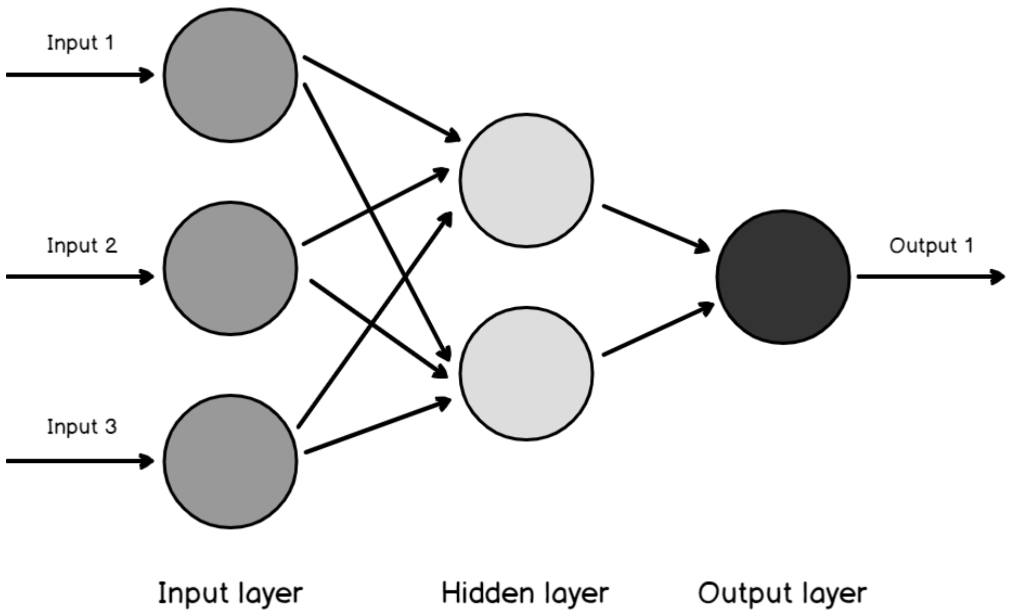


Figure 1: A simple neural network

Each neuron takes some inputs (the outputs of the neurons that are pointing to it), $a_1, \ldots, a_p$ and outputs a vector of the form:

$$\pi\left(\underbrace{\beta_0}_{\text{Bias Term}} + \beta_1 a_1, \ldots, \beta_p a_p\right).$$

where $\pi(\cdot)$ is some pre-specified activation function. The bias term β_0 and each of the weights $\beta_1, \ldots, \beta_p$ are parameters that we'd like to "learn" using our data.

Specifically, we would like to "learn" the parameters that minimize some cost function of our output. An example of this would be to choose the parameters that minimize the mean squared error on our training data:

$$MSE(\cdot) = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y}_i - y_i)^2.$$

In order to do this minimization, we essentially need to do a gradient descent on our model parameters. However, computing this gradient is not straightforward. This brings us to back propogation, which is a way of computing the gradient that takes advantage of the feed-forward nature of the neural network.

2 Back Propogation

To begin with, consider a very simple neural network.

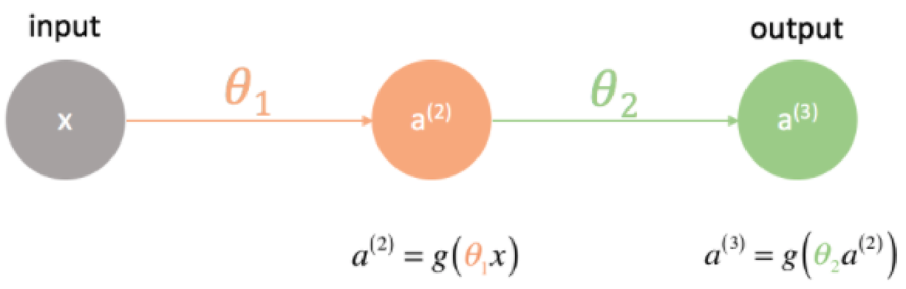


Figure 2: A very simple neural net

The output can be written as $\widehat{y}_i = g\left(\theta_2 \cdot g(\theta_1 \cdot x_i)\right)$. My cost function is the MSE:

$$\begin{aligned}\mathcal{C}\left(\theta_1, \theta_2\right) &= \sum_{i=1}^{n} \left(\widehat{y}_i - y_i\right)^2 \\ &= \sum_{i=1}^{n} \Big(g\left(\theta_2 \cdot g(\theta_1 \cdot x_i)\right) - y_i\Big)^2\end{aligned}$$

I would like to compute the gradient with respect to θ_1 and θ_2. Since this is fairly simple in this setting we do it by hand and then examine the results to gain some intuition for back propogation:

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial \theta_2} &= 2\sum_{i=1}^{n} \Big(\Big(g\left(\theta_2 \cdot g(\theta_1 \cdot x_i)\right) - y_i\Big)\Big) g'\left(\theta_2 \cdot g(\theta_1 \cdot x_i)\right) \cdot g\left(\theta_1 \cdot x_i\right) \\ \frac{\partial \mathcal{C}}{\partial \theta_1} &= 2\sum_{i=1}^{n} \Big(\Big(g\left(\theta_2 \cdot g(\theta_1 \cdot x_i)\right) - y_i\Big)\Big) g'\left(\theta_2 \cdot g(\theta_1 \cdot x_i)\right) \cdot g'\left(\theta_1 \cdot x_i\right) x_i\end{aligned}$$

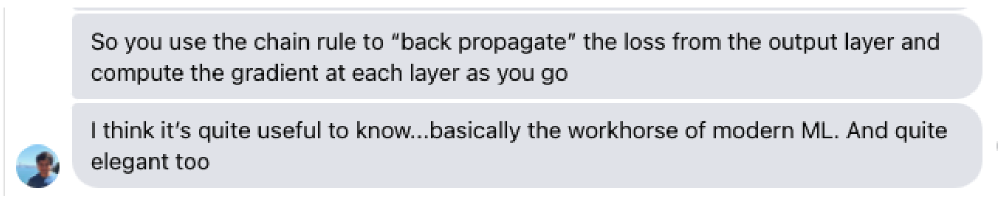
Comparing the partial derivatives with respect to θ_2 and θ_1 note that the partial derivative with respect to θ_1 can be computed iteratively.

1. First compute the derivative of the activation function in the last layer: $g'\left(\theta_2 \cdot g\left(\theta_1 \cdot x_1\right)\right)$. Use these to evaluate the derivative with respect to θ_2.
2. Compute the derivative with respect to the activation function in the first layer: $g'\left(\theta_1 \cdot x_2\right)$. Multiply these with the derivative with the with respect to the activation function in the last layer from the first step and use the resulting product to compute the derivative with respect to θ_1.

This provides the intuition for how back-propogation works to compute the total derivative with respect to all the weight/bias terms. To compute the total derivative we will start at the output layer, compute the derivatives of the activation function with respect to our parameters at this layer, and use these derivatives to calculate the derivative of the cost function with respect to the parameters in the last layer. To compute the derivative of the cost function with respect to parameters in the next layer back we will multiple these activation function derivatives from the final layer with the derivatives of the activation function from the next layer back layer. We then move to the second to last layer, calculate the derivatives of the activation function at each neuron and multiply these with the derivatives in the prior two layers and so on until we have the derivative with respect to every parameter.

Some notes:

- Note that the first term inside the summand in each of these derivatives is $\Big(g\left(\theta_2 \cdot g(\theta_1 \cdot x_i)\right) - y_i\Big)$. This means that observations for which our neural network prediction is far from the actual value will have a larger influence on our gradient, as we will try to correct for this and "fix" the loss function.
- In the very toy example we have drawn up here we are dealing with a (trivially) "fully connected" neural network. In practice, the multiplications at each step will depend on which nodes are connected to which.



So you use the chain rule to "back propagate" the loss from the output layer and compute the gradient at each layer as you go

I think it's quite useful to know...basically the workhorse of modern ML. And quite elegant too

Figure 3: Some thoughts from my college friend Shouvik, who is working in ML.

3 Overall Process

Finally, we combine our knowledge of the prediction process to come up with a unified process for "learning" the neural net:

1. Initialize a neural network with some structure (specify activation functions, layers, nodes, edges, etc.)
2. Start with an initial guess for your parameter values (bias terms and weights at each node)
3. Given your guessed parameter values, "forward propogate" your inputs through your neural network to get your predicted values
4. Given your predicted values "backward propogate" to compute the gradient of the cost function with respect to each parameter
5. Adjust your guess by moving in the direction of the negative gradient
6. Repeat steps 3-5 until you reach converge to a minimum
7. Try a few different starting guesses to make sure you have a global minimum