Final Paper: Backpropagation

Why I chose this topic?

When we trained a two-layer neural network for XOR gate in our lab session, we did it using PyTorch. It made me curious about how it's working behind the scenes since with PyTorch you can get away with using autograd, but if you construct a neural network using NumPy then things are much clearer. Backpropagation is a complex topic on its own and because of that I've only attempted to understand it at a surface level due to my own limitations with mathematical and technical understanding. This paper explains backpropagation using a neural network created for implementing XOR gate as an example.

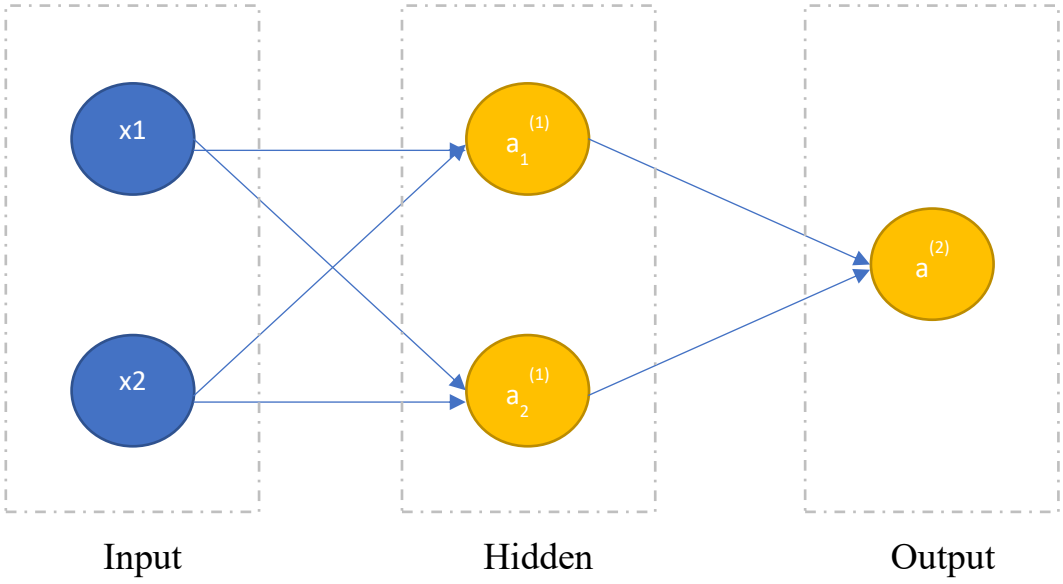
Prashant Kapoor
TLP – 2023
C-A 004: Introduction to AI
Plaksha University

Introduction

Backpropagation is a widely used algorithm for training feedforward neural networks. It uses gradient descent to update the parameters, i.e., weights and biases, of the model iteratively.

The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule.

Neural Network used for XOR gate



The Neural Network we're using for implementing XOR gate consists of an input layer (0^{th} layer), a hidden layer (1^{st} layer), and an output layer (2^{nd} layer). The superscript for any variable represents which layer it's a part of, and the subscript represents which neuron it's a part of. When we say $a_1^{(1)}$, this means we're referring to the output of the first neuron of the first layer. x1 and x2 are inputs. Outputs are calculated as follows:

$$z_1^{(1)} = w_1^{(1)} \cdot x_1 + b_1^{(1)}$$

$$z_2^{(1)} = w_2^{(1)} \cdot x_2 + b_2^{(1)}$$

While I've mentioned $z_1^{(1)}$ and $z_2^{(1)}$ separately, both can be calculated at the same time using NumPy dot products.

$$\overrightarrow{a^{(1)}} = \left\{\sigma\left(z_2^{(1)}\right) \middle| \sigma\left(z_2^{(1)}\right)\right\}$$

$\overrightarrow{a^{(1)}}$ is our vector consisting of results from both neurons of first layer. This is used as an input for the second layer which is our output layer. σ is the sigmoid function that is applied to $z_1^{(1)}$ and $z_2^{(1)}$.

$$z^{(2)} = \overrightarrow{w^{(2)}} \cdot \overrightarrow{a_1} + b_2$$

Here $\overrightarrow{w^{(2)}}$ is a vector. $\overrightarrow{w^{(2)}} \cdot \overrightarrow{a_1}$ is a simpler way of writing $w_1^{(2)} \times a_1^{(1)} + w_2^{(2)} \times a_2^{(1)}$ since the dot product computes exactly that. Our final output, $a^{(2)}$ is given by:

$$a^{(2)} = \sigma\left(z^{(2)}\right)$$

Training the Network

For the loss function, we've used Binary Cross Entropy.

$$L = -\frac{1}{m}\sum_{i=1}^{m}(y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i))$$

Using Gradient Descent, we'll update our weights and biases iteratively

$$w^{[i+1]} = w^{[i]} - \alpha\frac{\partial L}{\partial w^{[i]}}$$

$$b^{[i+1]} = b^{[i]} - \alpha\frac{\partial L}{\partial b^{[i]}}$$

Here, 'α' is the learning rate, 'w' and 'b' are weights and biases. Attach subscripts to them and you'd know which layer it's pointing to. The superscript refers to iteration.

Now, computing these gradients is where Backpropagation Algorithm shows its magic and makes the entire process more efficient. The gradients will be computed using chain rule of differentiation:

Second Layer:

$$\frac{\partial L}{\partial w^{(2)}} = \frac{\partial L}{\partial a^{(2)}} \times \frac{\partial a^{(2)}}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial w^{(2)}}$$

$$\frac{\partial L}{\partial b^{(2)}} = \frac{\partial L}{\partial a^{(2)}} \times \frac{\partial a^{(2)}}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial b^{(2)}}$$

First Layer:

$$\frac{\partial L}{\partial w^{(1)}} = \frac{\partial L}{\partial a^{(2)}} \times \frac{\partial a^{(2)}}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial a^{(1)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial z^{(1)}}{\partial w^{(1)}}$$

$$\frac{\partial L}{\partial b^{(1)}} = \frac{\partial L}{\partial a^{(2)}} \times \frac{\partial a^{(2)}}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial a^{(1)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}} \times \frac{\partial z^{(1)}}{\partial b^{(1)}}$$

To reduce the number of steps in the algorithm, we can compute and store

$$\frac{\partial L}{\partial z^{(2)}} = \frac{\partial L}{\partial a^{(2)}} \times \frac{\partial a^{(2)}}{\partial z^{(2)}}$$

This makes it easier to calculate second layer gradients:

$$\frac{\partial L}{\partial w^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial w^{(2)}}$$

$$\frac{\partial L}{\partial b^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial b^{(2)}}$$

For the first layer, we compute and store

$$\frac{\partial L}{\partial z^{(1)}} = \frac{\partial L}{\partial a^{(2)}} \times \frac{\partial a^{(2)}}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial a^{(1)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}}$$

Computing this can be made easier by using $\frac{\partial L}{\partial z^{(2)}}$ we computed earlier

$$\frac{\partial L}{\partial z^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial a^{(1)}} \times \frac{\partial a^{(1)}}{\partial z^{(1)}}$$

And our gradients can be calculated much more efficiently

$$\frac{\partial L}{\partial w^{(1)}} = \frac{\partial L}{\partial z^{(1)}} \times \frac{\partial z^{(1)}}{\partial w^{(1)}}$$

$$\frac{\partial L}{\partial b^{(1)}} = \frac{\partial L}{\partial z^{(1)}} \times \frac{\partial z^{(1)}}{\partial b^{(1)}}$$

So essentially, we're going backwards from the last layer to the first layer and storing the results to reduce the complexity of our algorithm. By calculating gradients using this method and updating the parameters using gradient descent for a certain number of iterations, or EPOCHS, we reach a point where our loss is minimum, and our outputs are predicted perfectly.

When we're calculating gradients for a multi-layered neural network, Backpropagation algorithm helps a lot in reducing the complexity.

The same neural network created for XOR gate using NumPy along with an implementation of backpropagation can be found here in this notebook:

https://colab.research.google.com/drive/1PR4VDAR_xLwxPb1se4bw_vzbN5-Uifwm?usp=sharing