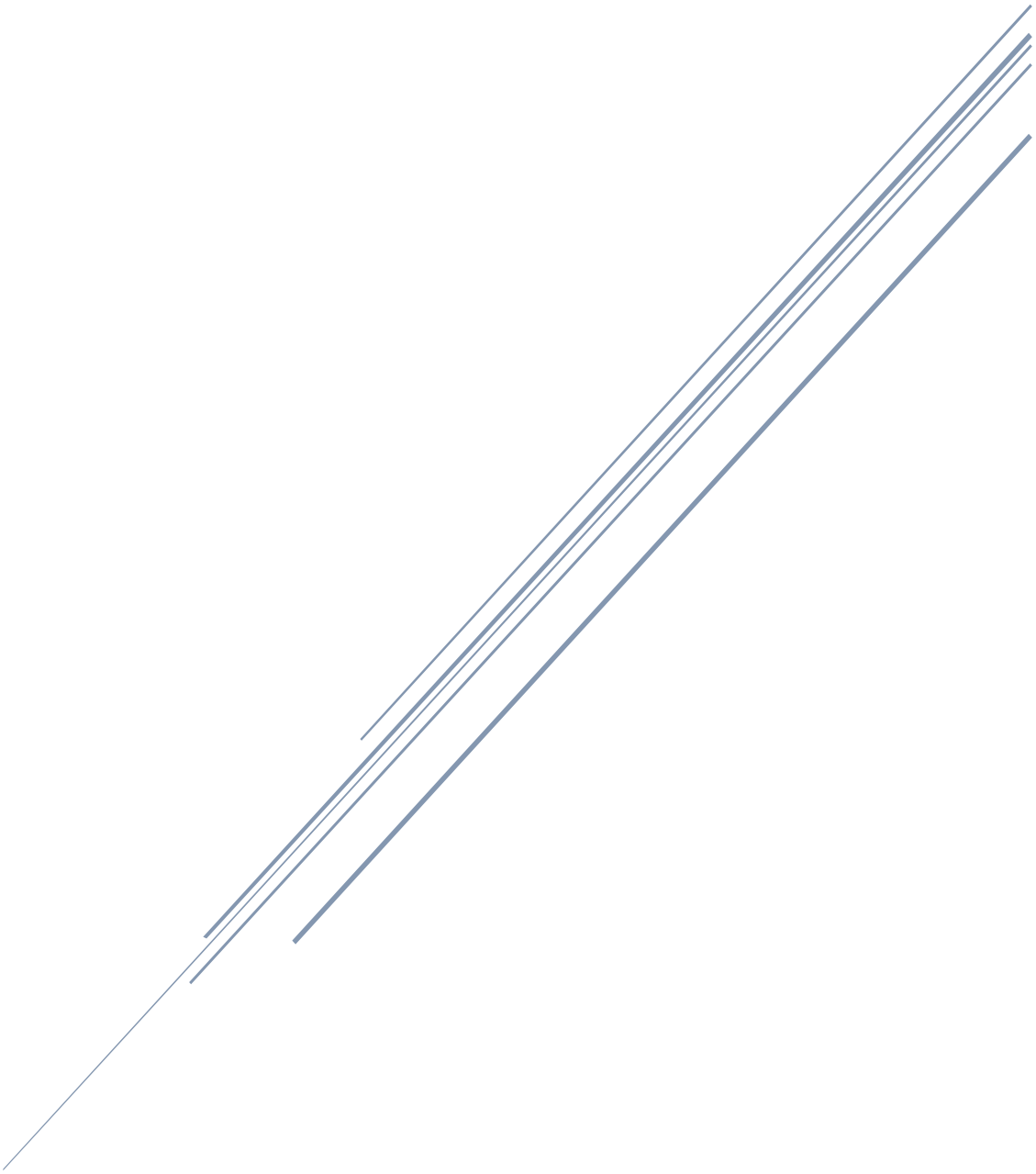
ARTIFICIAL NEURAL NETWORKS

Module 2



v-cardona

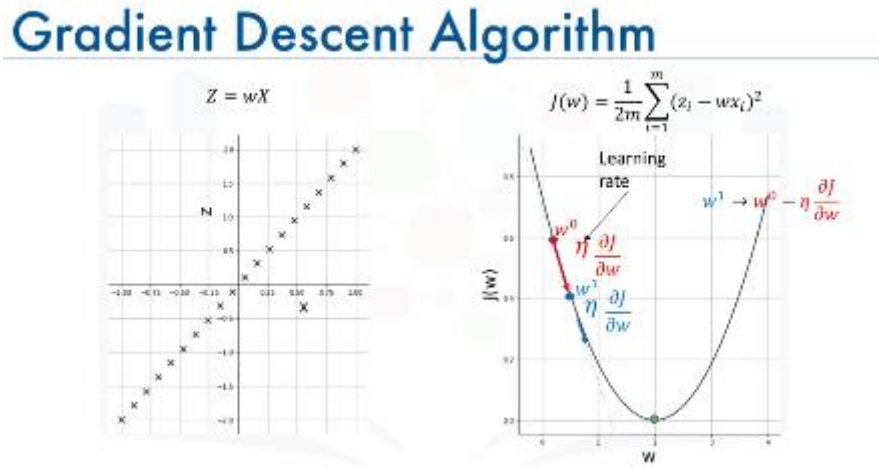
Deep Learning Fundamentals with Keras

Contenido

Gradient descent

Gradient descent is an iterative optimization algorithm for finding the minimum of a function. To find the minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point.

We start at a random initial value of w. Let us call it w0, and say it's equal to 0.2, and we start taking steps towards the green dot which is w = 2. To determine in which direction to move, we compute the gradient of the loss function at the current value of w, which is 0.2. The slope of the tangent at w gives the gradient = 0.2, and then the magnitude of the step is controlled by a parameter called the learning rate. The larger the learning rate, the bigger the step we take, and the smaller the learning rate, the smaller the step we take. Then we take the step and we move to w1. W1 is essentially computed as w0 minus the learning rate times the gradient at w0. This represents the first iteration of the algorithm. At w1, we repeat the same process of computing the gradient at w1 and using the same learning rate to control the magnitude of the step towards the minimum. We keep repeating this step repeatedly until we hit the minimum or a value of the cost function that is very close to the minimum, within a very small-predefined threshold.



When choosing the learning rate, we have to be very careful as a large learning rate can lead to big steps and eventually missing the minimum. On the other hand, a small learning rate can result in very small steps and therefore causing the algorithm to take a long time to find the minimum point.

Backpropagation

Training is done in a supervised learning setting, where each data point has a corresponding label or ground truth. Training is needed when the predicted value by the neural network obviously does not match the ground truth for a given input. The training process:

1. Calculate the error E between the predicted values and the ground truth labels. This error now represents the cost or the loss function.
2. the next step would be to propagate this error back into the network and use it to perform gradient descent on the different weights and biases in the network, to optimize them using the same equations

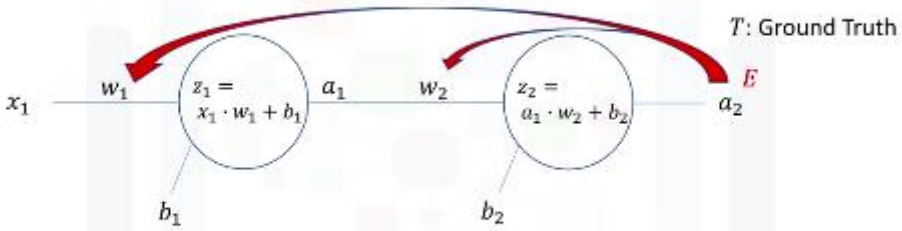
Backpropagation Algorithm

1. Calculate the error between the ground truth and the estimated output. Let's denote the error by E.
2. Propagate the error back into the network and update each weight and bias as per the following equations:

$$w_i \rightarrow w_i - \eta \cdot \frac{\partial E}{\partial w_i}$$

$$b_i \rightarrow b_i - \eta \cdot \frac{\partial E}{\partial b_i}$$

Backpropagation Algorithm



1. Calculate the error: $E = \frac{1}{2}(T - a_2)^2$ $\qquad E = \frac{1}{2m}\sum_{i=1}^{m}(T_i - a_{2,i})^2$
2. Update w_2, b_2, w_1, and b_1

Updating w_2

$$w_2 \rightarrow w_2 - \eta \cdot \frac{\partial E}{\partial w_2}$$

$$E = \frac{1}{2}(T - a_2)^2 \longrightarrow \frac{\partial E}{\partial a_2}$$

$$a_2 = f(z_2) = \frac{1}{1 + e^{-z_2}} \longrightarrow \frac{\partial a_2}{\partial z_2}$$

$$z_2 = a_1 \cdot w_2 + b_2 \longrightarrow \frac{\partial z_2}{\partial w_2}$$

Updating w_2

$$w_2 \rightarrow w_2 - \eta \cdot \frac{\partial E}{\partial w_2}$$

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

$$= \left(-(T - a_2)\right) \cdot \left(a_2(1 - a_2)\right) \cdot (a_1)$$

$$w_2 \rightarrow w_2 - \eta \cdot \left(-(T - a_2)\right) \cdot \left(a_2(1 - a_2)\right) \cdot (a_1)$$

Complete Training Algorithm

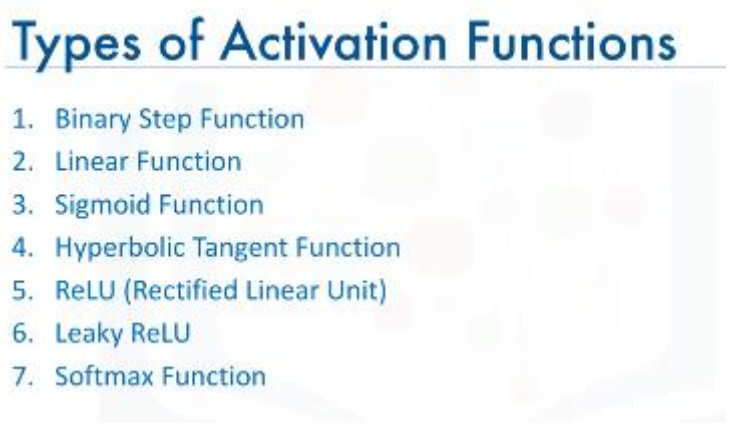
1. Initialize the weights and the biases
2. Iteratively repeat the following steps:
   1. Calculate network output using forward propagation
   2. Calculate error between ground truth and estimated or predicted output
   3. Update weights and biases through backpropagation
   4. Repeat the above three steps until number of iterations/epochs is reached or error between ground truth and predicted output is below a predefined threshold

Vanishing gradient

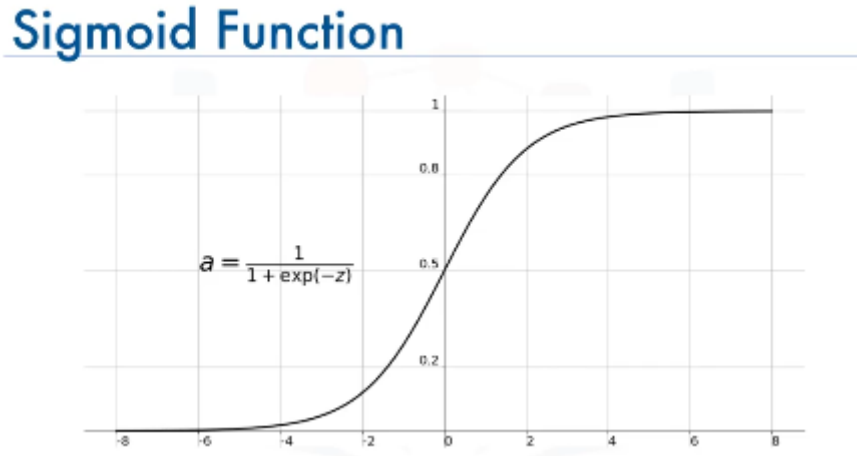
Vanishing gradient is a problem with the sigmoid activation function that prevented neural networks from blooming sooner. It occurs when the derivative it is too small so the difference with the new calculated value does not change significantly.

When we do backpropagation, we keep multiplying factors that are less than 1 by each other, and so their gradients tend to get smaller and smaller as we keep on moving backward in the network. This means that the neurons in the earlier layers learn very slowly as compared to the neurons in the later layers in the network. The earlier layers in the network are the slowest to train. The result is a training process that takes too long and a prediction accuracy that is compromised. Accordingly, this is the reason why we do not use the sigmoid function or similar functions as activation functions, since they are prone to the vanishing gradient problem.

Activation functions



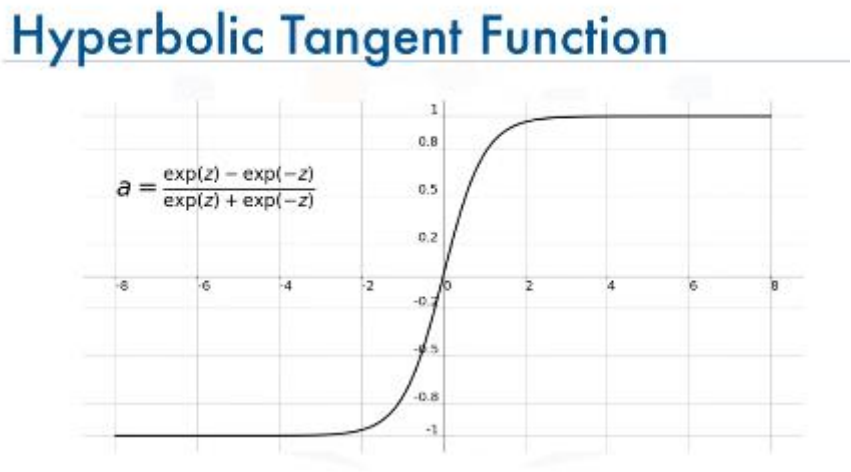
Sigmoid



Sigmoid functions used to be widely used as activation functions in the hidden layers of a neural network. However, as you can see, the function is pretty flat beyond the +3 and -3 region. This means that once the function falls in that region, the gradients become very small and as the gradients approach 0, the network doesn't really learn.

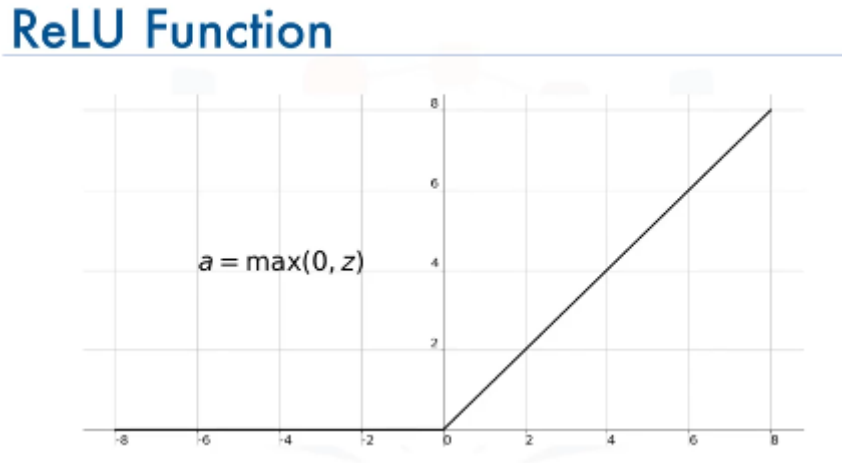
Another problem with the sigmoid function is that the values only range from 0 to 1. This means that the sigmoid function is not symmetric around the origin. The values received are all positive. Well, not all the times would we desire that values going to the next neuron be all of the same sign.

Hyperbolic tangent



It is actually just a scaled version of the sigmoid function, but unlike the sigmoid function, it's symmetric over the origin. It ranges from -1 to +1. However, although it overcomes the lack of symmetry of the sigmoid function, it also leads to the vanishing gradient problem in very deep neural networks.

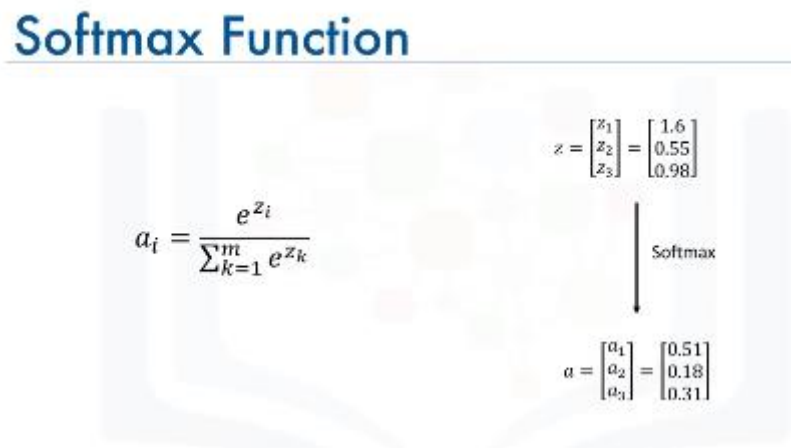
Rectified linear unit (ReLU)



It is the most widely used activation function when designing networks today. In addition to its being nonlinear, the main advantage of using the ReLU, function over the other activation functions is that it does not activate all the neurons at the same time.

Only a few neurons are activated, making the network sparse and very efficient. Also, the ReLU function was one of the main advancements in the field of deep learning that led to overcoming the vanishing gradient problem.

Softmax



The softmax function is also a type of a sigmoid function, but it is handy when we are trying to handle classification problems. The softmax function is ideally used in the output layer of the classifier where we are actually trying to get the probabilities to define the class of each input. It is easier to classify a given data point and determine to which category it belongs.

Conclusion