Backpropagation for Deep Learning

Activation Functions

In a neural network, an activation function specifies how the weighted sum of the input is turned into an output from a node or nodes within a layer.

Occasionally, the activation function is referred to as a "transfer function." If the activation function's output range is restricted, it is referred to as a "squashing function." Numerous activation functions are nonlinear, which is referred to as the layer or network design's "nonlinearity."

The activation function chosen has a significant impact on the neural network's capabilities and performance, and different activation functions may be utilised in different portions of the model.

Although networks are meant to employ the same activation function for all nodes in a layer, technically the activation function is applied within or after the internal processing of each node in the network.

A network may have three types of layers: input levels that accept raw domain data, hidden layers that accept input from another layer and pass it along to another layer, and output layers that produce predictions.

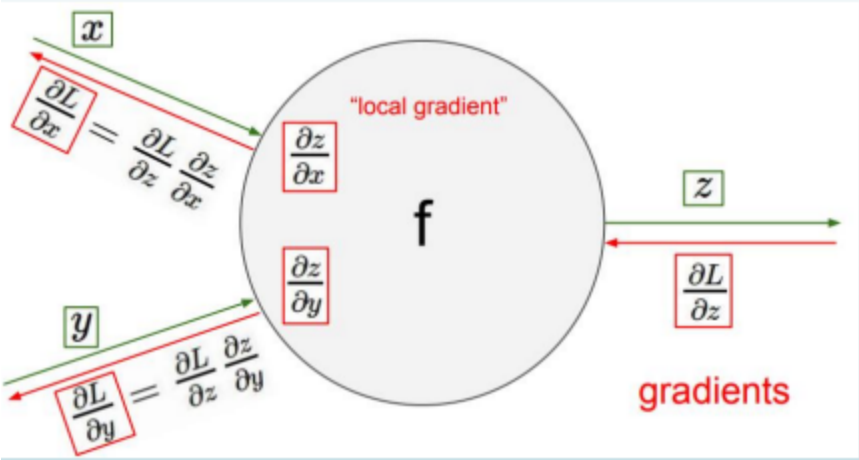
Typically, all buried layers share the same activation function. The output layer's activation function is often different from that of the hidden layers and is determined by the type of prediction required by the model.

Additionally, activation functions are often differentiable, which means that the first-order derivative for a given input value may be determined. This is necessary because neural networks are often trained using backpropagation of error, which requires the derivative of the prediction error to update the model's weights.

There are numerous types of activation functions used in neural networks, although only a few of these functions are likely to be employed in practise for the hidden and output layers.

Backpropagation

To refer to "the backward propagation of errors," the term "backpropagation" is used. This is because an error is calculated at the output and distributed backwards via the network layers as they are created.



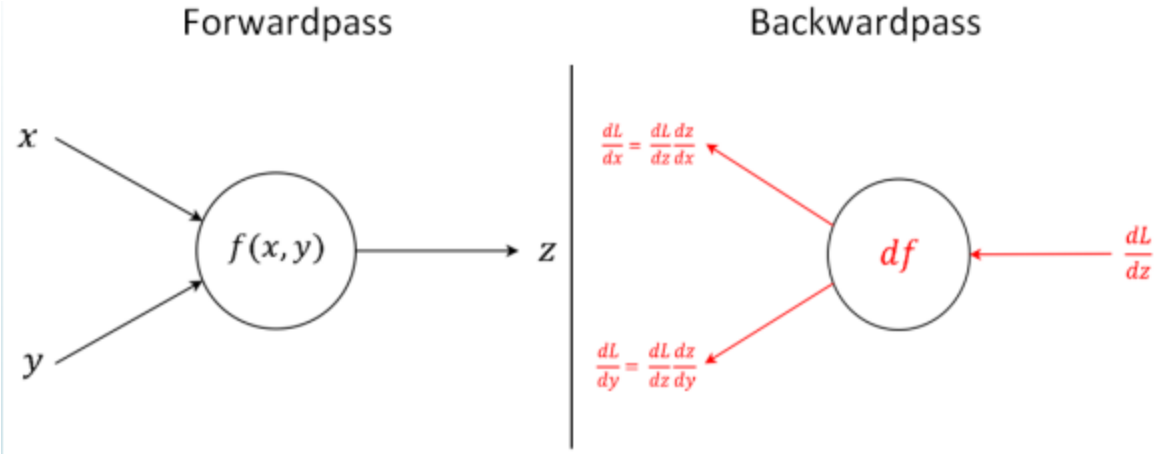
History

- The backpropagation algorithm was first developed in the 1970s, but it wasn't until David Rumelhart, Geoffrey Hinton, and Ronald Williams published a seminal work in 1986 that its significance was realised.
- Deep neural networks rely heavily on backpropagation nowadays.
- Other than error correction in Deep Neural Network Layers, it's difficult to envisage any other technique replacing it.
- People doubt its ability to solve AGI (Artificial General Intelligence), yet it's still the best we have right now.

What is the purpose of Backprop?

- This cannot be done in a deep neural network context due to gradient descent
- As a result, we will be unable to update all of the neurons at once.
- As a result, the weight update takes a long time to complete.
- Additionally, there is no requirement for a separate memory for each layer.

What takes place, in reality?



First, let's define the model's components.

Assume we have a deep neural network that has to be trained. The goal of training is to create a model that executes the XOR (exclusive OR) functionality with two inputs and three hidden units, resulting in a training set (truth table) that looks like this:

X1 | X2 | Y
0 | 0 | 0
0 | 1 | 1
1 | 0 | 1
1 | 1 | 0

Furthermore, an activation function that specifies the activation value at each node in the neural net is required. Let's start with a simple identity activation function:

f(a) = a

A hypothesis function is also required to identify what the input to the activation function is. This will be the standard, all-too-familiar:

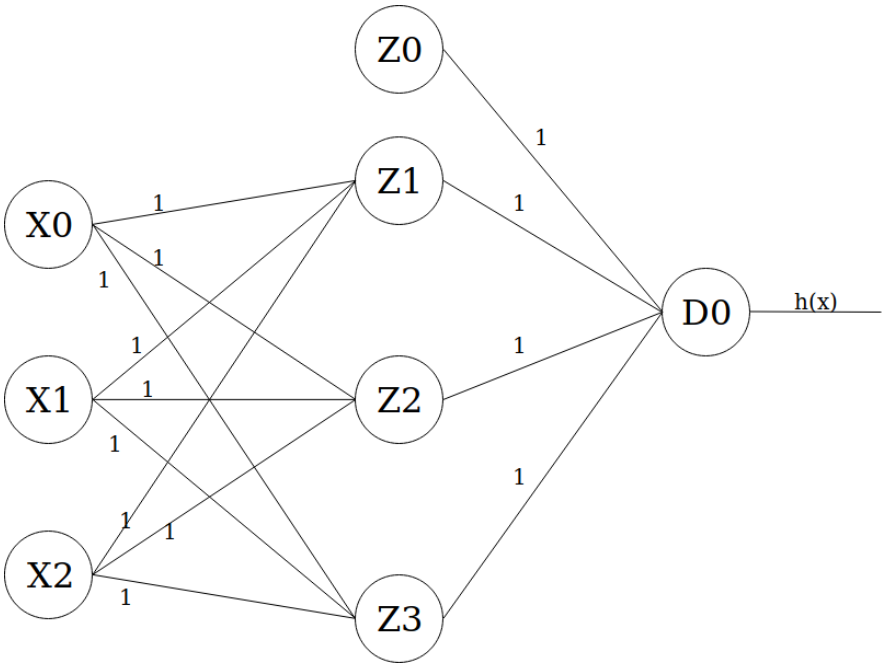
h(X) = W0.X0 + W1.X1 + W2.X2
or
h(X) = sigma(W.X) for all (W, X)

Let's also make the loss function the standard cost function of logistic regression, which appears hard but is actually rather simple:

Furthermore, we will utilise the Batch Gradient Descent optimization function to decide which weights should be adjusted in which direction to get a lower loss than the one we now have. Finally, the learning rate will be set to 0.1, and all weights will be set to 1.

Artificial Neural Network

Let's finally draw our long-awaited neural net diagram. This is what it should look like:



The input layer is the layer on the left, and it accepts X0 as the bias term of value 1, as well as X1 and X2 as input features. The middle layer is the initial concealed layer, which likewise has a bias term Z0 of value 1. Finally, the output layer contains only one output unit D0, whose activation value equals the model's actual output (i.e. h(x)).

We are now forward-propagating

It's now time to pass the information from one layer to the next. This is accomplished through two steps that occur at each node/unit in the network:

1- Using the h(x) function we defined before, obtain the weighted sum of inputs for a specific unit.

2- In this example, we plug the value from step 1 into the activation function we have (f(a)=a) and use the activation value we get (i.e. the output of the activation function) as the input feature for the connected nodes in the next layer.

It should be noted that units X0, X1, X2, and Z0 have no units attached to them and supplying inputs. As a result, the preceding steps do not occur in those nodes. However, for the remainder of the nodes/units in the neural net, this is how it all happens for the first input sample in the training set:

Unit Z1:

h(x) = W0.X0 + W1.X1 + W2.X2
= 1 . 1 + 1 . 0 + 1 . 0
= 1 = a
z = f(a) = a => z = f(1) = 1

The same is true for the remaining units:

Unit Z2:

h(x) = W0.X0 + W1.X1 + W2.X2
= 1 . 1 + 1 . 0 + 1 . 0
= 1 = a
z = f(a) = a => z = f(1) = 1

Unit Z3:

h(x) = W0.X0 + W1.X1 + W2.X2
= 1 . 1 + 1 . 0 + 1 . 0
= 1 = a
z = f(a) = a => z = f(1) = 1

Unit D0:

h(x) = W0.Z0 + W1.Z1 + W2.Z2 + W3.Z3
= 1 . 1 + 1 . 1 + 1 . 1 + 1 . 1
= 4 = a
z = f(a) = a => 4 = z = f(4)

As previously stated, the activation value (z) of the last unit (D0) is the same as that of the entire model. As a result, for the set of inputs 0 and 0, our model projected a result of 1. The current iteration's loss/cost would be calculated as follows:

Loss = actual y - predicted y

= 0 - 4
= -4

The actual y value is derived from the training set, whereas the predicted y value is the result of our model. As a result, the cost at this iteration is -4.

According to our example, we now have a model that does not provide accurate predictions (it returned a value of 4 instead of 1), which is due to the fact that its weights have not yet been calibrated (they are all equal to 1). We also know the loss, which is -4. Back-propagation is the process of feeding this loss backwards in order to fine-tune the weights based on which. The optimization function (in our case, Gradient Descent) will assist us in determining the weights that will, ideally, result in a reduced loss in the following iteration. So let's get started!

If the following functions were used to feed forward:

f(a) = a

The partial derivatives of those functions will then be used to feed backward. There is no need to go through the process of calculating these derivatives. All we need to know is that the functions listed above will occur:

f'(a) = 1
J'(w) = Z . delta

where Z is just the z value acquired from the activation function computations in the feed-forward step, and delta is the unit loss in the layer.

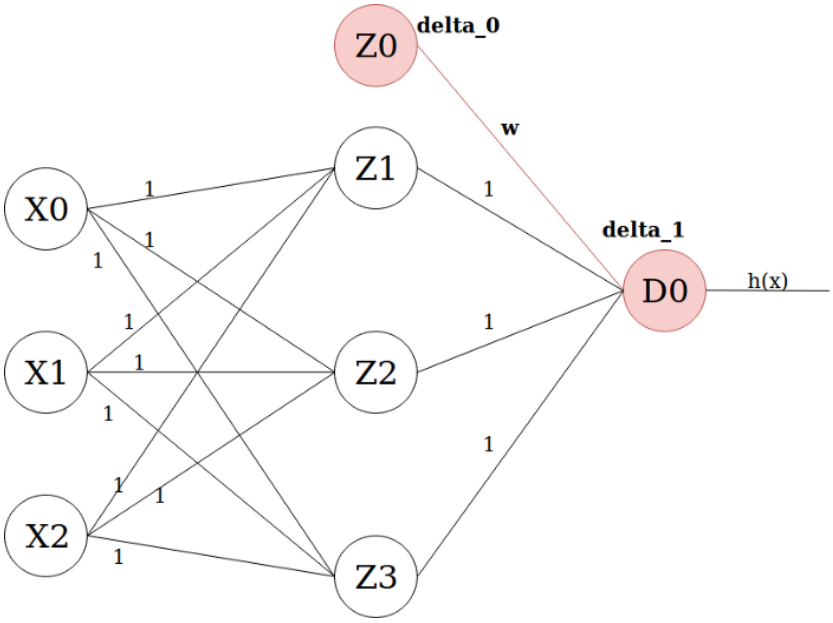
Calculating the deltas

We must now calculate the loss at each unit/node in the neural network. Why is this the case? Consider it this way: every loss that the deep learning model encounters is the result of all the nodes accumulating into a single number. As a result, we must determine which node is responsible for the majority of the loss in each layer so that we can penalise it in some way by assigning it a lower weight value and thereby reducing the total loss of the model.

Calculating the delta of each unit can be difficult. Mr. Andrew Ng, on the other hand, provided us with a shortcut formula for the entire thing:

delta 0 = w . delta 1 . f'(z)

where delta 0, w, and f'(z) are the same unit's values, and delta 1 is the loss of the unit on the other side of the weighted connection. As an example:



To calculate the loss of a node (for example, Z0), multiply the value of its corresponding f'(z) by the loss of the node it is connected to in the next layer (delta 1), and by the weight of the connection connecting both nodes.

Back-propagation works exactly like this. We perform the delta calculation step for each unit, back-propagating the loss into the neural net and determining which node/unit is accountable for the loss.

Let's get those deltas calculated and out of the way!

delta D0 = total loss = -4
delta Z0 = W . delta D0 . f'(Z0) = 1 . (-4) . 1 - 4
delta Z1 = W . delta D0 . f'(Z1) = 1 . (-4) . 1 - 4
delta Z2 = W . delta D0 . f'(Z2) = 1 . (-4) . 1 - 4
delta Z3 = W . delta D0 . f'(Z3) = 1 . (-4) . 1 - 4

There are a few things to keep in mind here:

- The loss of the last unit (D0) equals the loss of the entire model. This is because it is the output unit, and its loss equals the sum of all the units' losses, as previously stated.

- No matter what the input (i.e. z) is, the function f'(z) will always return the value 1. This is because, as previously stated, the partial derivative is as follows: 1 = f'(a)
- The delta values of the input nodes/units (X0, X1, and X2) are not present because those nodes control nothing in the neural net. They just serve as a bridge between the data set and the neural network. This is simply why the entire layer is not normally included in the layer count.

Weights are being updated.

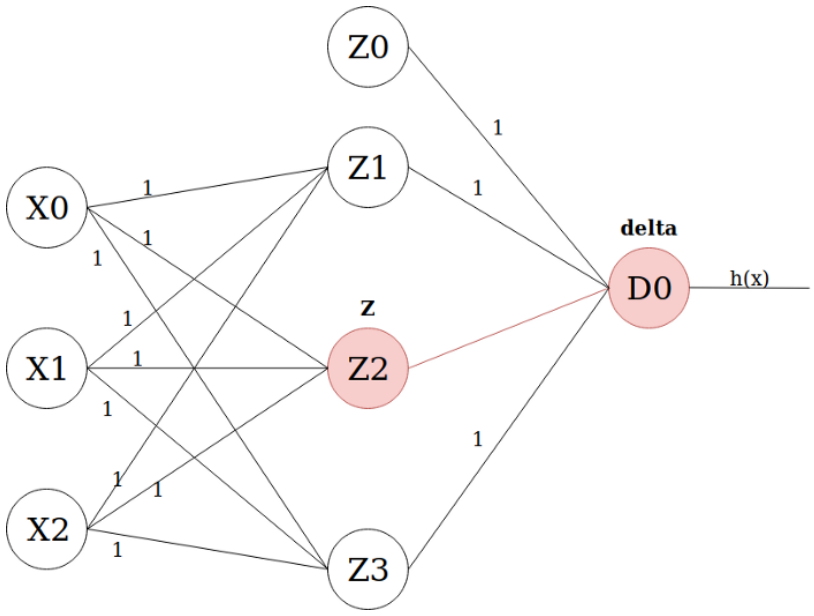
All that remains now is to update all of the neural net weights. This is based on the Batch Gradient Descent formula:

W := W - alpha . J'(W)

Where W is the weight in question, alpha is the learning rate (in our case, 0.1), and J'(W) is the partial derivative of the cost function J(W) with respect to W. Again, we don't need to dive into the math. As a result, let us employ Mr. Andrew Ng's partial derivative of the function:

J'(W) = Z . delta

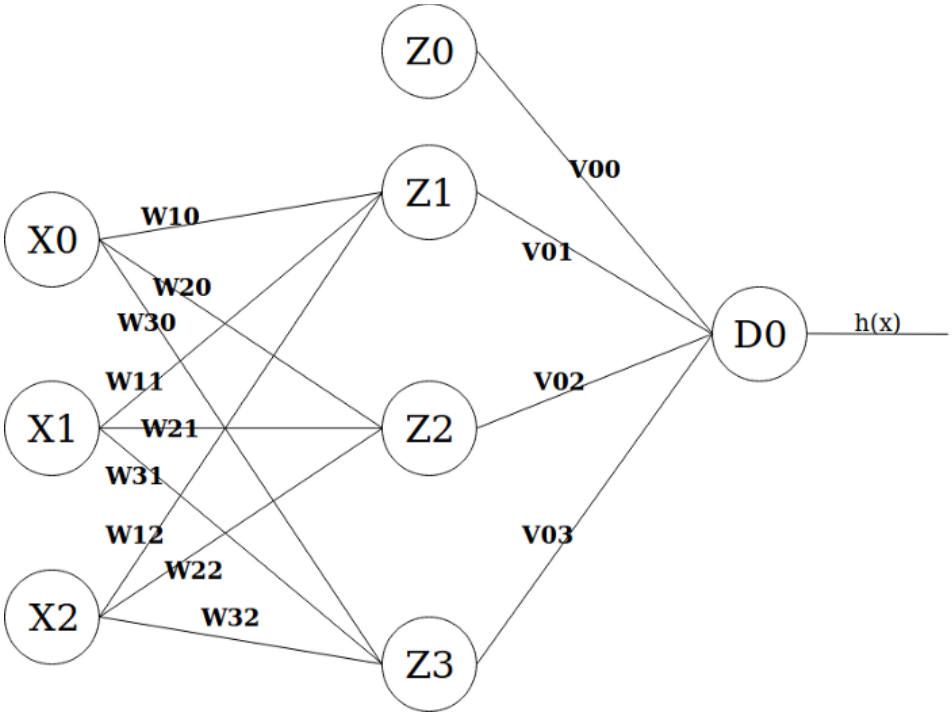
Where Z is the forward-propagation Z value and delta is the loss at the unit on the opposite end of the weighted link:



We now apply the Batch Gradient Descent weight update to all of the weights, utilising the partial derivative values obtained at each step. It is worth noting that the Z values of the input

nodes (X0, X1, and X2) are all equal to 1, 0, 0. The 1 represents the bias unit value, whereas the zeros represent the feature input values from the data set. Finally, there is no particular order in which the weights should be updated. You can update them in whatever order you like, as long as you don't update any weight twice in the same iteration.

Let us name the links in our neural nets in order to calculate the new weights:



New weight calculations will happen as follows:

W10 := W10 - alpha . Z_X0 . delta_Z1

= 1 - 0.1 . 1 . (-4) = 1.4

W20 := W20 - alpha . Z_X0 . delta_Z2

= 1 - 0.1 . 1 . (-4) = 1.4

. . . . .

. . . . .

. . . . .

W30 := 1.4

W11 := 1.4

W21 := 1.4

W31 := 1.4

W12 := 1.4

W22 := 1.4

W32 := 1.4

V00 := V00 - alpha . Z_Z0 . delta_D0

= 1 - 0.1 . 1 . (-4) = 1.4
V01 := 1.4
V02 := 1.4
V03 := 1.4

It's crucial to note that the model hasn't been adequately trained yet because we simply back-propagated through one sample from the training set. Doing what we did for all of the samples again will result in a model that is more accurate as we go, with the goal of getting closer to the least loss/cost at each stage.

It may seem strange to you that all of the weights have the same value. However, repeatedly training the model on different samples will result in nodes with varied weights based on their contributions to the total loss.