Chapter 2:- Back Propagation Training & Keras

2.1 How to Train a Neural Network

- Put in Training inputs, get the output
- compare output to correct answers: Look at loss fn J
- Adjust & Repeat
- Backpropagation tells us how to make single adjustment using calculus

Gradient Descent Training:-

① Make prediction

② Calculate Loss

③ Calculate gradient of the loss fn wrt parameters

④ Update parameters by taking a step in the opposite direction

⑤ Iterate

Propagation: Feed Forward NN:-

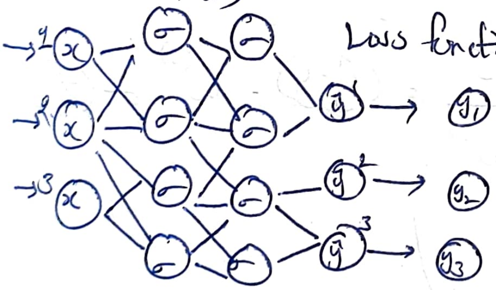
Pass Input (1st layer) → Calculate each layer (& hidden layers) → Get Output ($\hat{y}_i$) → Evaluate $J(y_i, \hat{y}_i)$

How to Train a Neural Net:-

How could we change the weights to make our Loss function Lower?

1. Think Neural Net as a function: $F: X \rightarrow Y$
2. F is a complex computation involving many weights W_k.
3. Given the structure, the weights "define" the function & therefore define our model.
4. Loss function is $J(y, F(x))$

F(x)



Loss function

weights w

$\hat{y}_1 \leftarrow z_1 = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$

$\hat{y}_2 \leftarrow z_2 = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$

$\hat{y}_3 \leftarrow z_n = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$

~~Each~~

Each layer calculating the value of (z)

Training:-

→ Get $\frac{\partial J}{\partial W_k}$ for every weight in the network

→ This tell us what direction to adjust each W_k if we want to lower our loss function

→ Make an adjustment & Repeat!

2.2 Backpropagation : Feedforward Network

Backpropagation is an algo that is designed to test for errors working back from output nodes to input nodes

Backpropagation:

We obtain desired changes to input using calculus:

- Functions are chosen to have 'nice' derivatives
- Numerical issues are to be considered.

Punchline:-

$$\frac{\partial J}{\partial w^{(3)}} = (\hat{y}-y) \cdot a^{(2)} \longrightarrow \frac{\partial J}{\partial w^{(2)}} = (\hat{y}-y) \cdot w^{(3)} \cdot \sigma'(z^{(3)}) \cdot a^{(1)}$$

$$\downarrow$$

$$\frac{\partial J}{\partial w^{(1)}} = (\hat{y}-y) \cdot w^{(3)} \cdot \sigma'(z^{(3)}) \cdot w^{(2)} \cdot \sigma'(z^{(2)}) \cdot X$$

→ The values for the weights of final layer in NN will be updated using that partial derivative in regards of that final layer.

→ Then from there, in order to calculate the weights of second layer, the layer before the final layer, we take what we learned from that final layer & take the dot product of ω of that final layer, multiplied by derivative of activation of z from that final layer again. & the dot product of $a^{(1)}$ too

→ And finally we add on the further steps needed & take the dot product with X our initial input in order to get the derivative in respects to our intial layer

*→ So the larger smaller errors will affect the size of each one of our gradients.

[Remember:- $\boxed{\sigma'(z) = \sigma(z)[1-\sigma(z)]}$]

Main idea behind Back propagation:-

① First run our NN with our intialized weights

② Then moving back through our layers, we're going to take the derivative of each of our weights in our final layer with respect to our loss function.

③ Then use that to again get our partial derivatives in respect to our layer two of our weights & then layer one weights finally.

(4) We will use these to update our initialized values & then again feed these updated weights through on their own net & repeat the process

Vanishing Gradients:-

This for updating out 3 layer feed for NN

$$\frac{\partial J}{\partial w^{(1)}} = (\hat{y}-y) \cdot w^{(3)} \cdot \sigma'(z^{(3)}) \cdot \sigma'(z^{(2)}) \cdot X$$

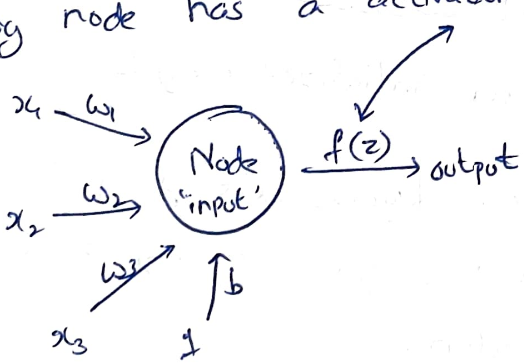
- Remember : $\sigma'(z) = \sigma(z)\left(1-\sigma(z)\right) \leq 0.25$

$$(0 \leq z \leq 1)$$

- As we have more layers, the gradients get very small at the early leayers (eg:- $w^{(3)}, w^{(2)}, w^{(1)}$)
- This is known as Vanishing gradient problem
- For this reason, other activations (such as ReLU) have become more common.

2.3 Types of Activation Functions:

Every node has a activation function



(1) without this activation function this would be as a linear model

(2) The activation improves our ability to determine non-linear outcomes

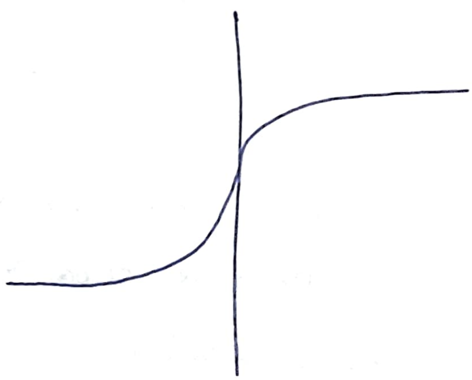
what we've discussed so far about activation function,

Logistic Regression is as linear regression with sigmoid activation function

1. Sigmoid Function:-

-> we use sigmoid functions because we want outputs zero & one & we want a non-linear model. And it also gives flexibility in our outputs.

-> The main advantage is that it produces the derivative of itself which also ranges from 0 to 1.



-> The disadvantage is something that we can see graphically here is that the derivative can tend to be a very low value. i.e for higher values of x, there is unnoticible change in 'y'. The derivatives are going to be very small.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

2. Hyperbolic Tangent Function :

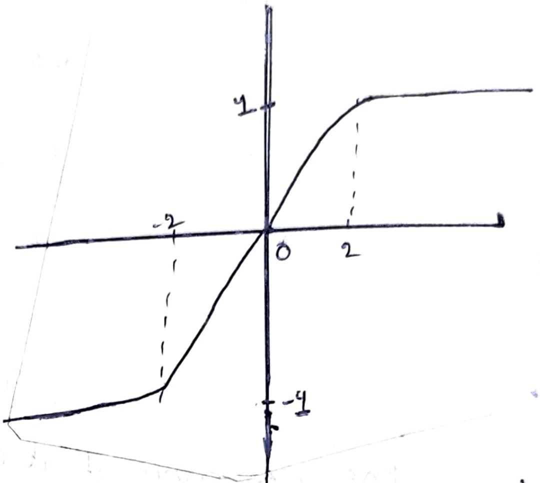
$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2x}-1}{e^{2x}+1}$$

* similar to sigmoid fn but this is bit stressed out

$\tanh(0) = 0$

$\tanh(\infty) = 1$

$\tanh(-\infty) = -1$



In graph :-

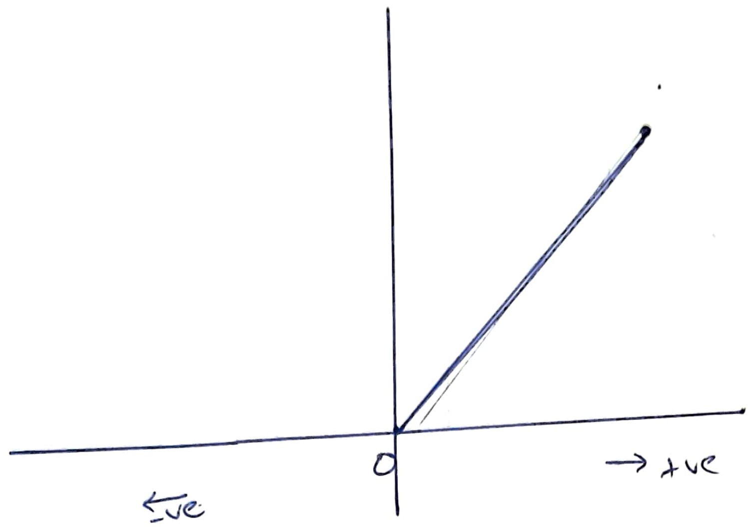
we see that for values between negative two & two, we have a sharper slope i.e. a small change in x will lead to large change in y. & thus derivative is going to be larger & gradient descent may be optimized.

* It's powerful if for any reason you want your values to be between one & negative one rather than 0 & 1.

But as we discussed early, we still have the same problem as sigmoid function, small derivative for higher values of x & that's a gun face that vanishing gradient problem.

3. Rectified Linear Unit (ReLU):

$$ReLU(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$



→ The graph is still non-linear, as the transition b/w less than zero & greater than zero introducing non-linearity.

Right side of the graph; the tiny derivative problem is solved, so vanishing gradient problem is solved by ReLU

Left side of graph, there is zero changes.

→ This zeroing out will allow for us to ignore nodes that may not be providing much extra info, & thus may be more efficient than sigmoid & hyperbolic fns, that always maintain at least some info at each node

→ On the other hand, there will be no-learning happening at each of those nodes that being zeroed out, & perhaps you want to ensure learning at nodes, with that in mind, we have the Leaky Rectified Linear Unit or LReLU

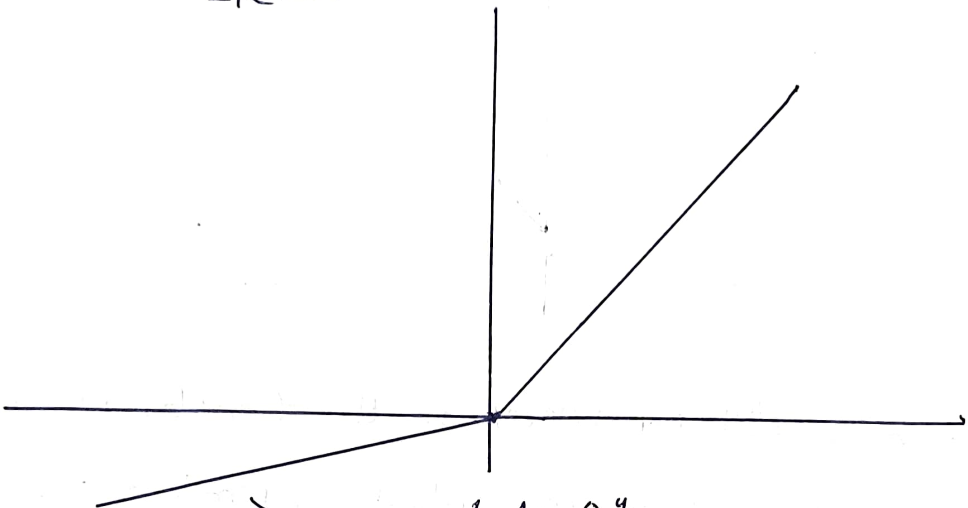
4. Leaky Rectified Linear Unit (LReLU):-

$$LReLU(z) = \begin{cases} az, & z < 0 \\ z, & z \geq 0 \end{cases} = \max(az, z) \; ; \text{ for } (a < 1)$$

$LReLU(0) = 0$

$LReLU(z) = z$ (for $z \geq 0$)

$LReLU(-z) = -az$



→ a is set to 0.1

This will solve the problem of nodes zeroing out throughout our network while keeping the advantage of a steady learning rate without that vanishing gradient problem

⇒ Just because it is solving a problem, it is need not to be necessary to be better than ReLU all time. Some times they are not neccessarily better all the time

Summary:-

Method	Use cases
Sigmoid Activation	useful when outcomes in (0,1) suffers from vanishing gradient.
Hyperbolic Tangent	useful when outcomes in (-1,1) suffers from vanishing gradient
ReLU	used to capture large effects, doesn't suffer from vanishing gradient
Leaky ReLU	Acts as ReLU & also allows -ve outcomes

2.4 Keras

Common Libraries for DL include:

i) TensorFlow → by Google

ii) Theano → Grandfather of DL frameworks

iii) PyTorch → by Facebook

iv) Keras is a high-level library, can run on either TensorFlow or Theano

We will focus on running Keras, which will run TensorFlow "under the hood"

2.5 Keras Workflow

Typical Command Structure:

- Build the structure of your network

1. Compile the model, specifying your loss function, metrics & optimizer (this includes learning rate)
2. Fit the model on your training data (specifying batch size, number of epochs)
3. Predict on new data
4. Evaluate your results

Building the model:

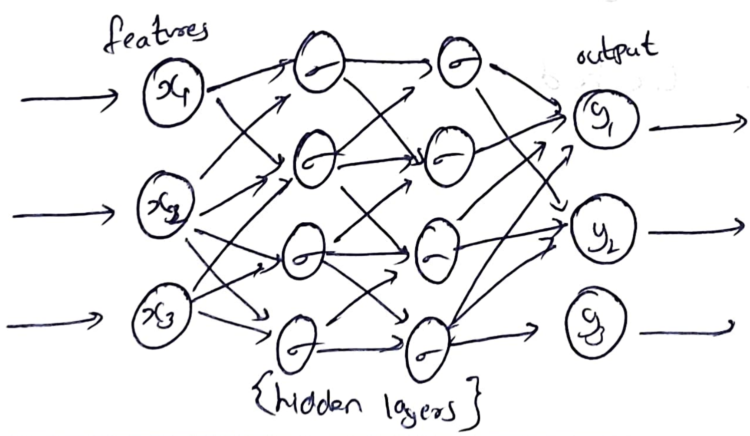
- Keras provides 2 approaches to building the structure of your model:

① Sequential Model: allows a linear stack of layers - simpler & more convienient if model has this form

② Functional API: more detailed & complex, but allows more complicated architectures.

Implementing an example NN in Keras:-

Let's build this NN structure in Keras:



Sequential Model:

from keras.models import Sequential
model = Sequential()

First import the Sequential function & initialize

from keras.layers import Dense, Activation

Then we add layers to model one by one

model.add(Dense(units=4, input_dim=3))

For the first layer specify the input dimensions

model.add(Activation('sigmoid'))

specify activation function

model.add(Dense(units=4))
model.add(Activation('sigmoid'))

For subsequent layers, the input dimensions is presumed from the previous layer

* This is just a sample code to make keras understandable