WHY ARE DEEP NEURAL NETWORKS HARD TO TRAIN

There are two challenges we might encounter when training our deep neural networks.

Vanishing Gradients

A problem with training networks with many layers (e.g. deep neural networks) is that the gradient diminishes dramatically as it is propagated backward through the network. The error may be so small by the time it reaches layers close to the input of the model that it may have very little effect. As such, this problem is referred to as the "vanishing gradients" problem. Like the sigmoid function, certain activation functions squish an ample input space into a small output space between 0 and 1.

Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small. For shallow networks with only a few layers that use these activations, this isn't a big problem.

However, when more layers are used, it can cause the gradient to be too small for training to work effectively.

Exploding Gradients

Exploding gradients are problems where significant error gradients accumulate and result in very large updates to neural network model weights during training.

An unstable network can result when there are exploding gradients, and the learning cannot be completed.

The values of the weights can also become so large as to overflow and result in something called NaN values. Exploding gradients can be avoided in general by careful configuration of the network model, such as choice of small learning rate, scaled target variables, and a standard loss function.

There are many approaches to address exploding gradients. Some of them are listed below:

1. Re-Design the Network Model

In deep neural networks, exploding gradients may be addressed by **redesigning the network to have fewer layers**.

There may also be some benefit in **using a smaller batch size** while training the network.

In recurrent neural networks, updating across fewer prior time steps during training, called truncated Backpropagation through time, may reduce the exploding gradient problem.

2. Use Long Short-Term Memory Networks

In recurrent neural networks, gradient exploding can occur given the inherent instability in the training of this type of network, e.g. via Backpropagation through time that essentially transforms the recurrent network into a deep multilayer Perceptron neural network.

Exploding gradients can be reduced by using the Long Short-Term Memory (LSTM) memory units and perhaps related gated-type neuron structures.

Adopting LSTM memory units is a new best practice for recurrent neural networks for sequence prediction.

3. Use Gradient Clipping

Exploding gradients can still occur in very deep Multilayer Perceptron networks with a large batch size and LSTMs with very long input sequence lengths.

If exploding gradients are still occurring, you **can check for and limit the size of gradients during the training** of your network.

This is called gradient clipping.

Dealing with the exploding gradients has a simple but very effective solution: clipping gradients if their norm exceeds a given threshold.

Specifically, the values of the error gradient are checked against a threshold value and clipped or set to that threshold value if the error gradient exceeds the threshold.

To some extent, the exploding gradient problem can be mitigated by gradient clipping (thresholding the values of the gradients before performing a gradient descent step).

4. Use Weight Regularization

Another approach, if exploding gradients are still occurring, is to **check the size of network weights and apply a penalty to the network's loss function for large weight values**.

This is called weight regularization and often an L1 (absolute weights) or an L2 (squared weights) penalty can be used.

Keras provides a weight regularization API that allows you to add a penalty for weight size to the loss function. A regression model that **uses L1 regularization technique is called Lasso Regression** and model which **uses L2 is called Ridge Regression**. The key difference between these two is the penalty term. Ridge regression adds "squared magnitude" of coefficient as penalty term to the loss function. Three different regularizer instances are provided; they are: L1: Sum of the absolute weights. L2: Sum of the squared weights. L1L2: Sum of the absolute and the squared weights.

We apply **kernel_regularizer** to penalize the weights which are very large causing the network to overfit, after applying kernel_regularizer the weights will become smaller.

If you want the output function to pass through (or have an intercept closer to) the origin, you can use **the bias regularizer.**

If you want the output to be smaller (or closer to 0), you can use the **activity regularizer**