3. Backprop and Neural Networks

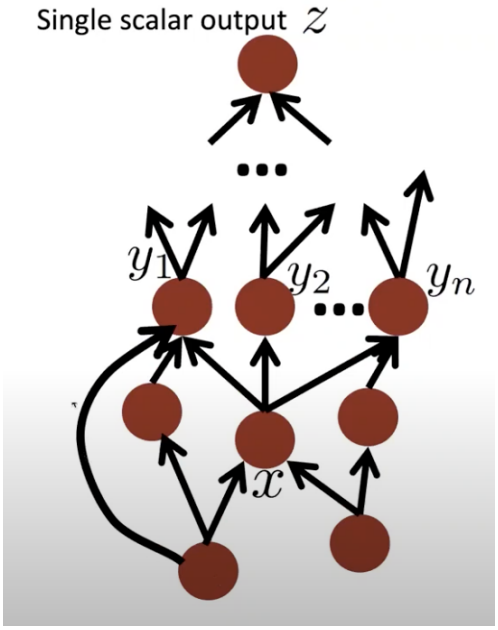
- Gradient descent math
    - This is basically matrix calculus
        - These gradients have a LOT of partial derivatives (n)
        - The gradients computed via the chain rule are also extremely similar (only the last partial derivative in the chain rule is different)
        - It's essentially that we avoid repeated computation (local error signal, delta, can be reused to prevent repeated calculations)
    - The shape of the derivatives
        - Disagreement between the best form
            - Jacobian form makes the chain rule easy
            - Shape convention is better for SGD
- Steps of NN training
    - 1). Forward propagation
        - Compute the initial desired value using current input parameters
    - 2). Backpropagation
        - Send back gradients to tell us how to update the model parameters

- Backpropagation Algorithm
    - That's essentially doing what was happening above!
    - Reuse derivatives of higher layers to be as computationally efficient as possible
        - i.e. Compute all the gradients at once (analogous to using delta when computing by hand)
    - Done correctly, the big O() complexity is actually the same for both forward and backward propagation
    - Backprop steps on one node:
        - 1. You have a node f (where some computation happens) and a downstream gradient flowing into it
        - 2. Work out the gradient of the node f
        - 3. Downstream gradient = upstream gradient * local gradient (chain rule!)
        - 4. Continue this all the way back up the tree
        - 4. The final backpropogated values tell us how much the overall output will change if we "wiggle" the values of each output
            - i.e. The slope at each point via the gradient

- If a node has multiple outward branches, you sum the slopes to see the effect that wiggling the value has

■ Modern DL frameworks (TensorFlow / PyTorch etc.) do 90% of the work, but the layer node derivatives still need to be calculated by hand

FProp:



BProp: