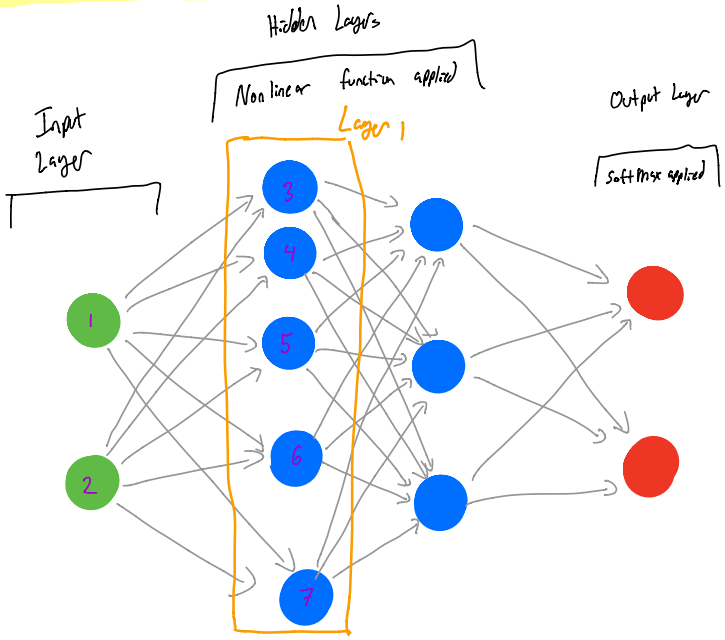
Neural Networks from Scratch



The more nodes here the more complex functions we can fit. Careful, you can also overfit data.

Forward Propagation → - Calculating subsequent layer values.

Each path has a weight, $W_{i,j}$, and each node has a bias, b_i.

So for example at Layer 1

$$\text{Values} = \text{activation\_function}\left(\underbrace{\text{weight}}_{\text{matrix of weights}} \cdot \underbrace{\text{inputs}}_{\text{Vector of Values from previous layer}} + \underbrace{\text{bias}}_{\text{Vector of current node layer}}\right)$$

$\cdot$ → element wise application of activation_function.

$$\begin{bmatrix} \text{Value\_3} \\ \text{Value\_4} \\ \text{Value\_5} \\ \text{Value\_6} \\ \text{Value\_7} \end{bmatrix} = \text{activation function}\left(\begin{bmatrix} W_{13} & W_{23} \\ W_{14} & W_{24} \\ W_{15} & W_{25} \\ W_{16} & W_{26} \\ W_{17} & W_{27} \end{bmatrix} \begin{bmatrix} \text{Value\_1} \\ \text{Value\_2} \end{bmatrix} + \begin{bmatrix} b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \right)$$

Activation functions - Nonlinear function for our hidden layer applied at each node so our hypothesis can take on nonlinear functions. - $\tanh(x)$, $\frac{e^x}{e^x+1}$ (Sigmoid), etc...

- Softmax function for output layer to transform to probabilities. Turns a vector into a proportional one that sums to 1. $\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{\text{len}(\vec{z})} e^{z_j}}$

Learning - How do we find weights and biases that minimize error?

A nice choice for a loss function of softmax outputs:

Cross-entropy loss, AKA negative log likelihood.

$$L(y, \hat{y}) = \frac{-1}{N} \sum_{n \in N} \sum_{i \in C} y_{n,i} \log(\hat{y}_{n,i})$$

- y: True value
- $\hat{y}$: Prediction value
- N: training examples
- C: classes (output nodes)

Backwards Propogation (Gradient descent)

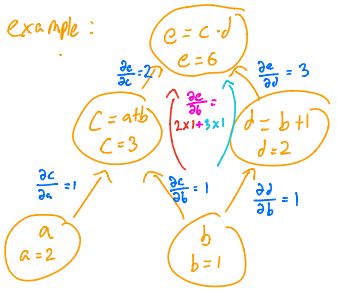
We use gradient descent to find minimum of loss function with respect to the parameters.

Thus we need the derivatives. $\frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial b_2}, \ldots \frac{\partial L}{\partial w_{1,3}}, \frac{\partial L}{\partial w_{2,3}}, \ldots$

We do this by starting from the output layer and working backwards via the backpropogation algorithm.

Backwards Propogation, formally - reverse-mode differentiation

Derivatives on computational graphs



C is directly effected by a, how does it change if we change a?

$$\frac{\partial c}{\partial a} = \frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1 + 0$$

Also need product rule: (u & v are independent)

$$\frac{\partial}{\partial v} uv = u \frac{\partial v}{\partial v} + v \frac{\partial u}{\partial v} = 0 + v$$

General Rule - sum over all possible paths, multiplying the derivatives on each edge of a path together.

But just summing over all paths is extremely inefficient with large graphs.

Consider another graph

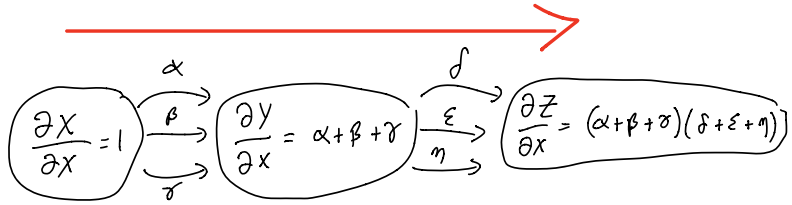
naive way → $\frac{\partial z}{\partial x} = \alpha\delta + \alpha\epsilon + \alpha\eta + \beta\delta + \beta\epsilon + \beta\eta + \gamma\delta + \gamma\epsilon + \gamma\eta$

how can we speed this up?

But we can just factor...

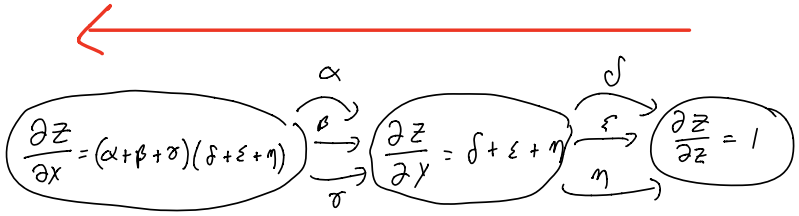
$$(\alpha + \beta + \gamma)(\delta + \epsilon + \eta)$$

Forward mode Differentiation



Calculates how one input effects every node $\left(\frac{\partial}{\partial x}\right)$

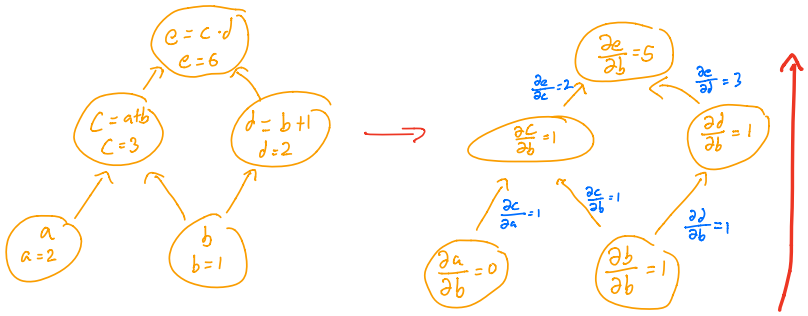
Reverse mode Differentiation



Calculates how every node effects one output $\left(\frac{\partial z}{\partial}\right)$

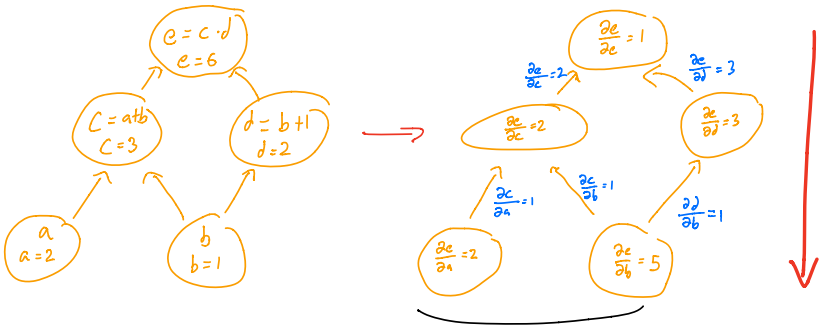
Ok back to the original graph... How is reverse mode computationally efficient?

with forward mode differentiation:



But, then we need to re-traverse the graph for a. What if we have 1000 inputs? That's a 1000x traversing the same graph.

with reverse-mode differentiation



We get all the derivatives we need in one graph traversal!!

Forward diff gives derivative of our output with respect to one input.

Reverse diff gives all of them!

Forward diff however gives us all of the derivatives of outputs with respect to our input.

If we have a large amount of outputs, forward mode can make more sense.

Overlook

1. Forward propogation, calculate weights and biases. Use auto diff to keep track of partial derivatives on paths.

2. Work your way back to calculate gradient using reverse-mode differentiation.

3. Finally, apply gradient descent updating weights and biases.

$$\hat{y}_{an} = \hat{y}_n - \gamma \nabla L\left(\hat{y}_1(w,b)\right)$$

$x \in \mathbb{R}$

b is irrational

$\sqrt{a} \in \mathbb{Z}^+$

$x \in \mathbb{R}$

x is rational