Backpropagation

Intuitive Explanation

- **Backpropagation** is a method for computing gradients
  - *Calculates how a single example would want to change the weights and biases*
    - How much and in which direction
  - **Gradient Descent -** carrying out backpropagation on all training data, averaging out gradients, each epoch
  - **SGD -** do backpropagation on each piece of **minibatch**, averaging fewer gradients, **faster** epochs
- Improve performance by considering changes to the **bias**, the **weights** and the **activation** of the previous layer
- We start from our output and calculate the weights backwards
- We want to most change the neurons that affect our output the most, i.e. the gradient of which is largest, i.e. are the most wrong
- At each node, we add all of the gradients / changes coming from the nodes in the next layer
  - I.e. at each node of the final hidden layer, we add up all of the gradients which are the opinions of each output node on how the output should change
  - This gives us the nudge that we want the nodes in the previous layer to result in

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

Backpropagation Calculus

- **z^L = W^TX+B**, the input into activation function at a neuron in layer L
- **L -** output layer
- **a^L -** the output of the activation function, **$\sigma(z^L)$**
  - **a^L_1 -** the activation of the first neuron in the output layer
- **C_0 -** the cost of 0th training example, **$(a^L - y)^2$**
  - **C = avg(Σ_k(dC_k/dw^L))**
  - ∇ **Nabla (gradient vector) -** a vector of **partial derivatives for f**
  - **∇C** - a vector [**$\partial C/\partial w^1$, $\partial C/\partial b^1$, ... , $\partial C/\partial w^L$, $\partial C/\partial b^L$**]
    - Tells us the gradient of every single parameter
- **Backpropagation** is finding out what is the gradient of the cost function with respect to the weights connected to the nodes in the output layer
- *We use chain rule to look to go from the derivative of C_0 wrt a^L to the derivative of z^L wrt w^L*
- **$dz^L/dw^L = a^{L-1}$** tells us that the influence of **w** on **z** depends on previous layer's activation **a^{L-1}**, which makes sense
- To get the gradients for **B**, we substitute **dz^L/dw^L** for **dz^L/db^L**
- To propagate errors backwards, we get the gradient at **a^{L-1}** in a similar way, then we find the gradients of **a^{L-1}** wrt **w^{L-1}**
- We compute one layer of the gradients at a time
-

$$\boxed{\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}}$$

Chain rule

$$\text{Cost} \longrightarrow C_0(\dots) = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

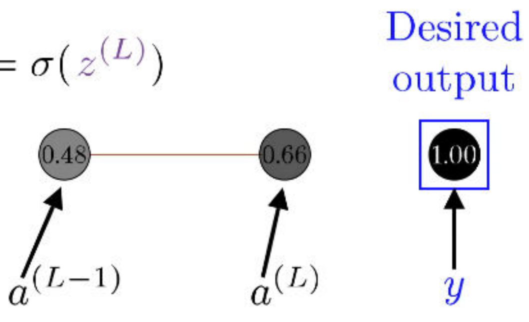
$$a^{(L)} = \sigma(z^{(L)})$$

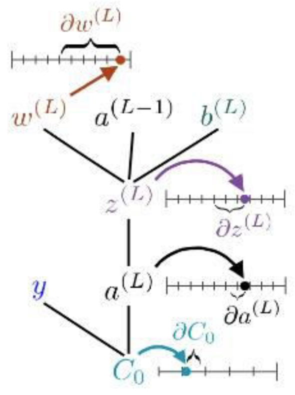
$$\frac{\partial C0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

Equations for a neural network with one node in each layer





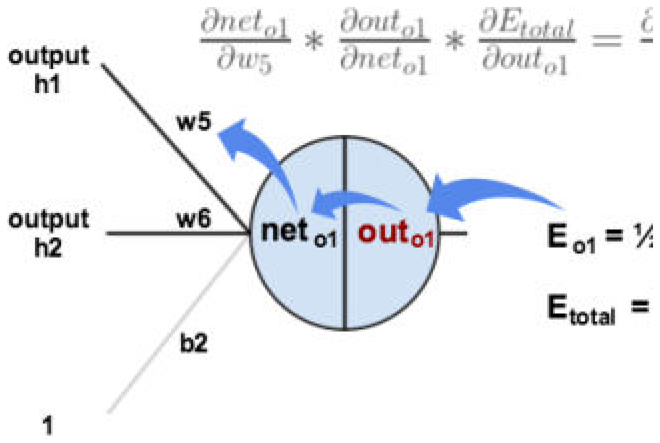
Equations for a normal neural network

- We must sum over the numerous neurons we have now
- **dC_0/da^{L-1}_k** becomes the sum of gradients over all nodes in L
- **C_0** becomes the sum of errors across all output nodes in L

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

$$C_0 = \sum_{j=0}^{n_L - 1} (a_j^{(L)} - y_j)^2$$

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \underbrace{\sum_{j=0}^{n_L - 1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}}_{\text{Sum over layer L}}$$



$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} =$
output h1
w5
output h2
w6
net o1
out o1
E o1 = ½
E total =
b2
1

Extra

Parametric vs Nonparametric

- **Parametric ML Algorithms -** Number of parameters is fixed w.r.t the data sample size
  - Doesn't change its mind about how many parameters it will model
  - Complexity is bound by the number of parameters even if the amount of data is unbounded
  - All of the knowledge the model will learn is encapsulated by the **finite set of parameters**, which means that it is **independent of previously observed data**
    - $$P(x|\theta, \mathcal{D}) = P(x|\theta)$$
  - E.g. In **OLS Regression**, number of parameters will always be length of β + 1 (variance)
    - E.g. simple linear regression
  - E.g. neural net with **fixed architecture** and **no weight decay**
  - *Greatly simplifies learning process, but limits what can be learned*
  - *Simple functions only*
  - E.g. linear regression, logistic regression, LDA, perceptrons
- **Nonparametric ML Algorithms -** The number of parameters can grow with the sample size
  - Assumes the data distribution cannot be defined in terms of a finite set of parameters
  - *Good when you have lots of data with no prior knowledge*
- **Argument they're the same -** a nonparametric algorithm can be approximated with a parametric model with an infinite number of parameters