3B1B - Neural Networks

1. what is a neural network?
2. Gradient Descent, how neural networks learn
3. Backpropation, step-by-step
4. Backpropigation calculus

What is a neural network?

This video handles the structure of a NN - no learning

Plain NN = Multilayer perceptron

neuron = thing that holds a number

with image neuron = Pixel = activation

input → Hidden → output

input num of neurons = Pixel count

hidden neuron = arbitary choice

with Digit ~~recong~~ recognition the output neurons correspond to a possible digit 0-9

think about neurons in one layer being active & setting off ~~layers~~ neurons in the next layer ~~of~~

Connetors between neurons are weights

Activation would be better if between 0-1 (sigmoid)

Sigmoid is applied to weighted sum of neurons & weights

$$\sigma(w_1 a_1 + w_2 a_2 + \dots + w_n a_n)$$

bias is used to adjust activation of weighted sum, i.e. if weight = 10 - 10 bias = 0 so neuron isnt active

Can think of each layer as learning new characteristic of the problem

linear Algebra way to write weighted sums of ~~weights~~

neurons of input layer (0)

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} A_0^{(0)} \\ A_0^{(0)} \\ \vdots \\ A_n^{(0)} \end{bmatrix} \quad \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}$$

bias

Each row is the weights for a connecting neuron

$$a^{(1)} \sigma(W a^{(0)} + b)$$

Much easier way to write the connections between layers

Python: $a = \text{Sigmoid}(\text{np.dot}(w, a) + b)$

A better way to view a neuron might be as a function rather than a number

Sigmoid is almost never used now

instead the main activation used is Relu(a)

Relu = max(0, a)

i.e. if less than 0 then neuron is "turned off"

Gradient Descent, how NNs learn?

- Grad Desc
- Analyze network

Digit recognition — 28×28 image = 784

In → Hid → Hid → output

784 → 16 → 16 → 10

+16 Bias, +16 Bias

Bk weights & bias — These are the thing to change

MNIST database = Collection of handwritten Digits

to start the training process initialize the network by randomly assigning all weights & bias

Outputs will be rubbish but use these output to create a loss function

in the case of 0.-9 Digits

0: $(\text{pred} - \text{actual})^2$

1:

2:

n:

then do average cost over all training images

Sum

This total or average cost function is the performance of the network that needs to be optimize against

NN function:
Input: 784 number (pixels)
Output: 10 number (0-9)
Paras: 13,002 weights & biases

↳ Cost func = Save but 1 output metric over all train data

Cost function = performance
now need a way to set w. & b to improve perform

$C(w)$ = cost & we create a start point then we want to work out which way to step in order to make $C(w)$ lower

How to change the input to the cost function: w & b

- find the slope of the function
- if positive slope, move left : if neg, move right
- interate to achelve minimum

Could land in any ~~local~~ global minima

Gradient of a function gives ~~the~~ you the direction of steepest descent $\nabla C(x,y)$

the basic "algo" for ~~computing the~~ Minimizing the function:

1. compute gradient direction ∇C
2. Small step in $-\nabla C$ direction
3. Repeat

the concept is the same for $-\nabla C(\vec{w})$

$$\vec{w} = \begin{bmatrix} 2\times 4 \\ x_2 \\ x_3 \\ x_5 \end{bmatrix} \qquad -\nabla C(\vec{w}) = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

grad of cost w/ respect to weight

minus to reverse slope

x_n gets updated by $-\nabla C(w) = y_n$

Recall, the cost function is an average over all training data images

So an improvement to the cost is an improvement to the performance of the whole procedure

the Algo for computing this gradient explicitly is the main part of how NNs work

— Backprop (Next video)

learn = Min cost function

All of the weights to a neuron can be visualized to see the "pattern" of learning

Multi-layer perceptron is old tech from the 80s + 90s

But is the precursor to more modern tech

Visualizing learning weights in NNs shows you just how basic & often not good the learning is

---

BackProp, Step-by-Step

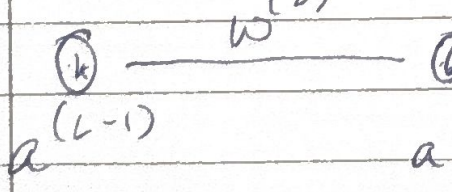
the negative gradient of the cost function is what is used to update the weights & biases

$$-\nabla C(\dots)$$

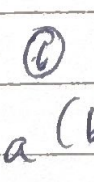
Computing the gradient for the entire training set is very clunky & slow so it done in "mini-batches"

Update weights @ each batch (2)
- Stochastic gradient descent

Backprop - Calculus

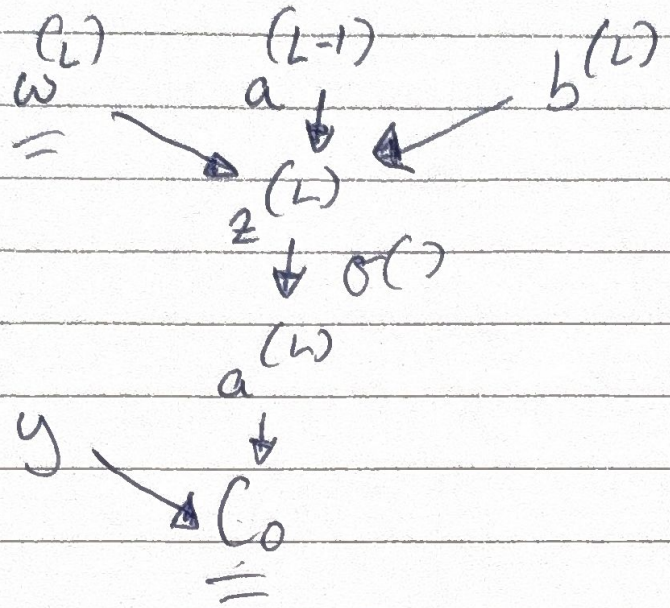


vs



$$C_0() = (a^{(L)} - y)^2$$

$$a^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)} \quad \text{where } z^{(L)} \text{ s.t. } a^{(L)} = \sigma(z^{(L)})$$



Connections of what is needed to calculate what

We want to know what happens to the cost func if we change a weight $w^{(L)}$ by a tiny amount

$\frac{\partial C_0}{\partial w^{(L)}}$ w/ respect to $w^{(L)}$

A change in $w^{(L)}$ changes $z^{(L)}$ changes $A^{(L)}$ changes C_0

there is a chain of intermediate changes that need to be follow to observe the chang

$$\frac{\partial C_0}{\partial w^{L}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z_L} \quad \frac{\partial C_0}{\partial a^{(L)}}$$

this is the chain rule — Multiply these 3 ratios to understand how the cost change w/ respect to small changes in $w^{(L)}$

Working through the derivative chain:

* $C_0 = (a^{(L)} - y)^2$

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

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

$$\frac{\partial C_0}{\partial a^L} = 2(a^{(L)} - y)$$

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

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

$$\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

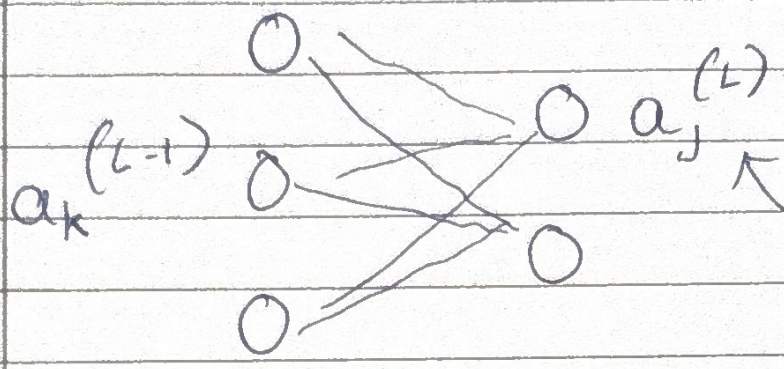
note this is just single example in net $= \frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum \frac{\partial C_k}{\partial w^{(L)}}$
* over all training examples

∇C = the true gradient is built up over all train examples & all partial derivatives of all weights and biases

the route is just the same for bias

$$\frac{\partial C_0}{\partial b^L} = \frac{\partial z^L}{\partial b^L} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L}$$

the concept remains much the same as we increase the complexity of the network



$$C_0 = \sum (a_j^{(L)} - y_j)^2$$

← Sum of nodes output across j times

$w_{jk}^{(L)}$ = weights are denoted by their end, start node

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

Cost function for each weight in the network

This gives you the value through which to "update" the layer of ~~weg~~ weights

Note network gets updated sequentially

nearest → Deepest layer

recalc is required after each update starting from the cost function

weights change activation values which change cost func