youtube

first principles of CV - Neural Networks Series

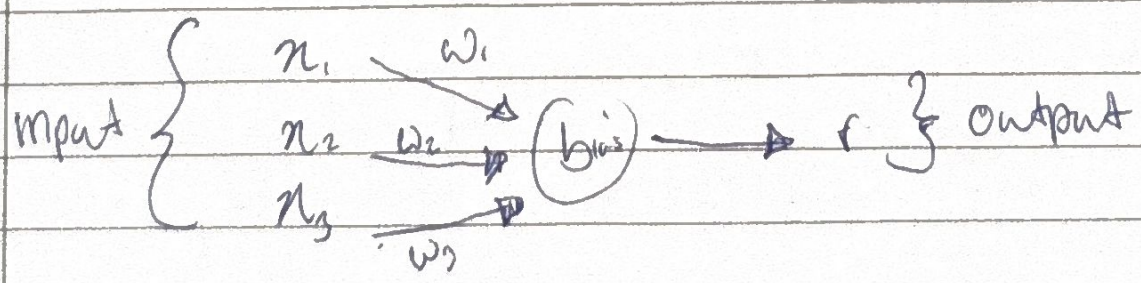
(1) overview
(2) Perceptron → Perceptron network multilayP
(3) Activation Function
(4) Neural network
(5) Gradient Descent
(6) Backpropagation
(7) Example applications

Overview

NN
(1) Perceptron → v similar to linear model
(2) Perceptron Network
(3) Activation functions
(4) Neural network → very big, how to train? GD
(5) Backprop → inner because GD is xpen
~~Grad Desc is very comp~~
(6) Example applicatns

Perceptron

very simple type of neuron



$$f = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

let $z = w \cdot x + b$

→ this is what activ func gets applied to

$$a = f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

the activation function of a perceptron is a step function (0, 1)

Example Q: Will you go to movies

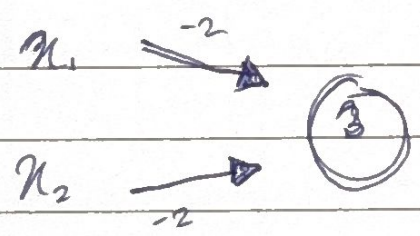
x_1 is the weather good?

x_2 Do you have company?

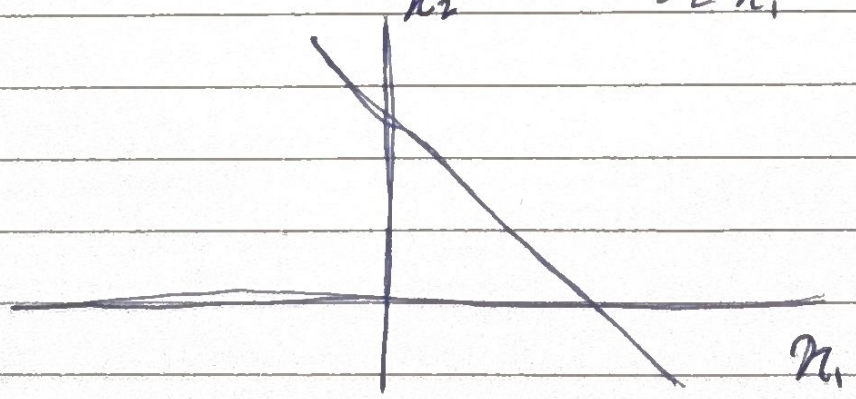
x_3 is the theater nearby

- Set weight $W_1 = 4$ $W_2 = 2$ $W_3 = 2$
- Set bias = Min threshold $b = -5$

  i.e. if w_1 & (w_2 or w_3) activ func > 0



$z = -2x_1 - 2x_2 + 3$



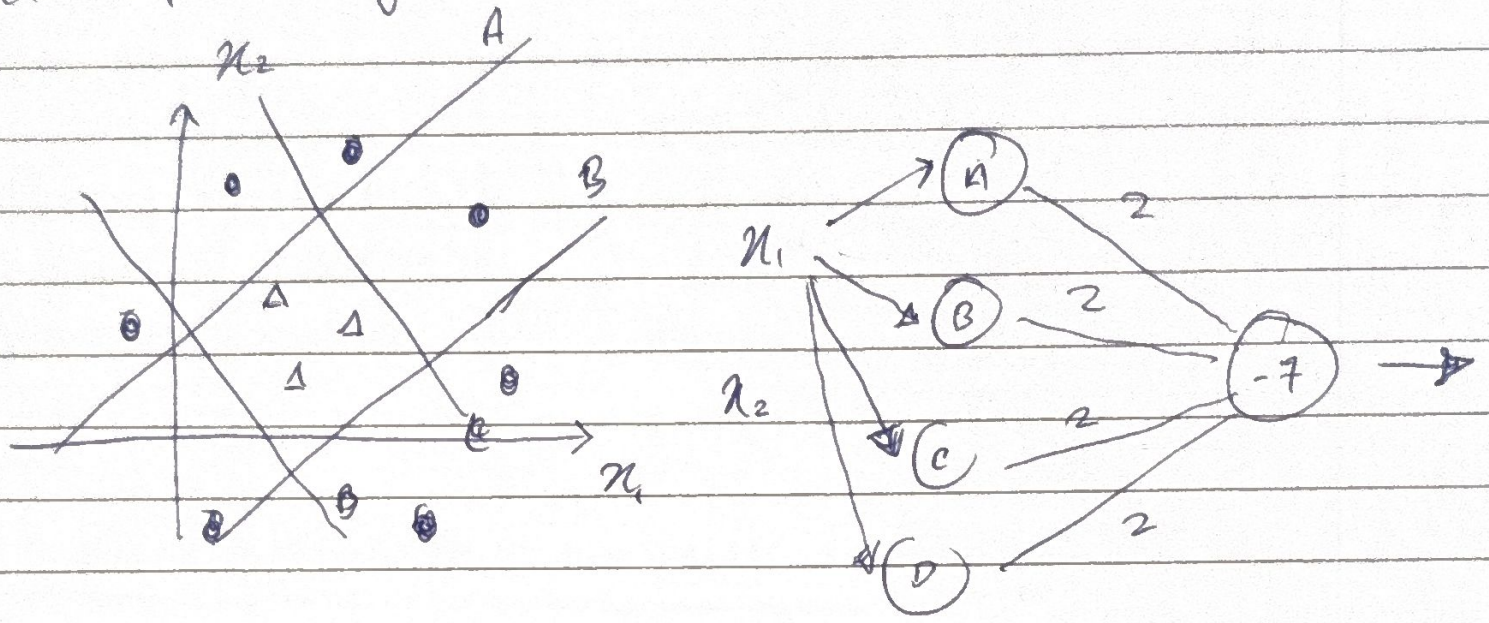
this is what the inside of a perc is doing

A lin classifier

Perceptron Network

what can we do with a network of perceptrons?

A network allows use to construct a linear classifier for complex regions



- each node create its own plane (0, 1)
- only when each node is switched on will the total cross the threshold

Multi-layer Perceptron Network

input layer → hidden layer → output layer

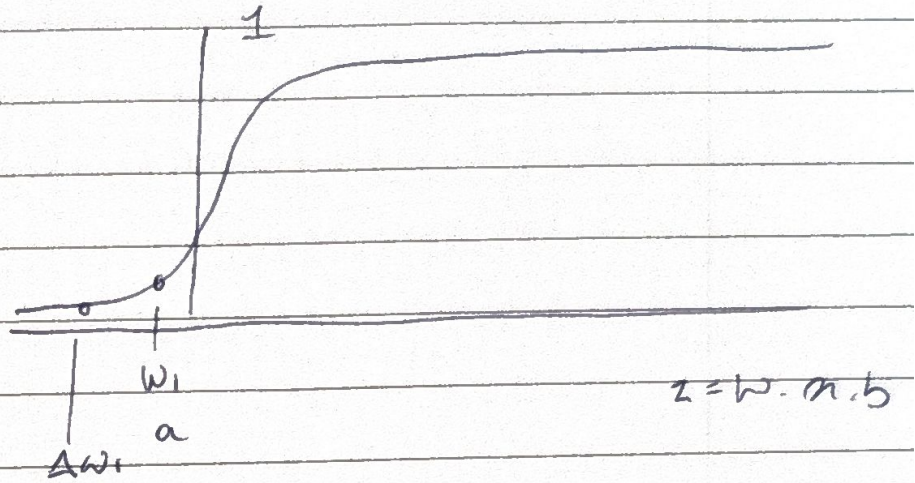
Activation function

- function that takes you from input → output
- For perceptron = Step function by default
- the task w/ perceptron is to find the correct weights & bias (Parameters)
- How does a change in w Δw change the output?
- this is an issue for the step function because of 0 & 1 output
- can't see the impacts of changing w & b, only when 0, 1 changes
- too unstable

Step function is too dramatic, would like it to be a smoother function

Sigmoid neuron

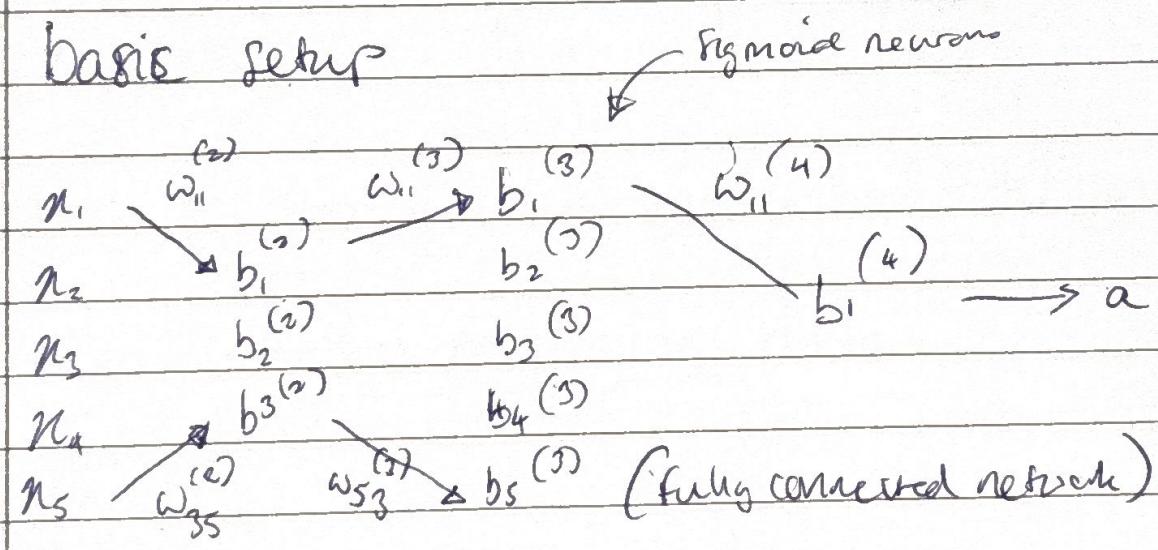
$$a = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Allow for measurable output w/ respect to change in w or b

Neural Networks - Construct & train

basic setup



Problem → recognise handwritten

MNIST → binarised, thresholded, bound 20×20 pixels within 28×28 image

Output → Digits 0-9 (10)

goal = high activation output for digits

Nielsen 2015 has model for this

Nielsen uses 784 inputs of 784 neurons = a neuron for each pixel in the 28 x 28

9 which go into 30 hidden layer neurons

Out into 10 output neurons for each digit

784 → 30 → 10

gets 95% Accuracy

if we dont have a trained ~~net~~ network we start by randomizing the weights & biases

this will ~~be~~ give a poor output & unpredictable outputs

how to change weights & bias to improve?

need labelled trainging data

Steps

1. pick random weights & bias
2. computed network activations (outputs) for each of the training images
3. compute the COST for whole train data set

$$C_x(w,b) = \|\hat{a}(x) - a(x|w,b)\|^2$$

actual — Pred

euclidean distance

how this looks per image →

$$\left\| \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0.2 \\ 0.3 \\ 0.0 \\ 0.9 \\ 0.1 \\ 0.2 \end{bmatrix} \right\|^2$$

for whole data: $C(w,b) = \frac{1}{n} \sum C_x(w,b)$

4. update weights & bias ~~&~~ Average to min COST

Gradient Descent

Adjust weights & biases using optimization process (GD)

Want to min cost function using params (w, b)

Start w/ w & b values
- adjust until min cost

How do you measure change in a function?
Taylor Series
- This tells us →

$$f(x+\Delta x, y+\Delta y) \approx f(x,y) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y$$

$$\Delta f(x,y) \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y$$

Ultimately shows us a way to measure the change in a function w/ respect to changes in its params

Applied to cost function:

$$\Delta C \approx \frac{\partial C}{\partial w_i}\Delta w_i + \frac{\partial C}{\partial b_j}\Delta b_j$$

vector form $= \Delta C = \begin{bmatrix} \frac{\partial C}{\partial w_i} & \frac{\partial C}{\partial b_j} \end{bmatrix} \cdot \begin{bmatrix} \Delta w_i \\ \Delta b_j \end{bmatrix}$

∇C : gradient of C
Δv = change in vars

Gradient / cost pens: ∇C
Para changes: Δv

$\nabla C \cdot \Delta v = \Delta$ Cost

∇C gives you value to increase cost function, we want opposite to decrease

let $\Delta v = -\nabla C$ $\Delta C = -\|\nabla C\|^2$

η = learn rate = how fast to descend

let $\Delta v = -\eta \nabla C$ $\Delta C = -\eta \|\nabla C\|^2$

the change in the cost is equal to minus eta(η) time the magnitude of the gradient ∇C

to change paras in each step

$$w_i \rightarrow w_i' = w_i - \eta \frac{\partial C}{\partial w_i}$$

$$b_j \rightarrow b_j' = b_j - \eta \frac{\partial C}{\partial b_j}$$

① to update the paras, find the gradient of the cost w/ respect to para

② times by learning rate

③ & subtract from current para value

learning rate:
- bigger = Quicker
- but too big = overshoot
- might jump to another values which isn't the minima global

intuition → on a mountain, identify the steepest direction & go the opposite - eventually get to the bottom

What to do when there are many paras?
$\Delta v = -\eta \nabla C$ is still valid
↳ Scale to all layers, params & dimensions

Gradient Computation

using finite Differences:

$$\frac{\partial C}{\partial w^L_{jk}} \approx \frac{C(w+\Delta w, b) - C(w,b)}{\Delta w^L_{jk}}, \quad \Delta w = \begin{bmatrix} 0 \\ \vdots \\ \Delta w^L_{jk} \\ \vdots \\ 0 \end{bmatrix}$$

Change in cost over change in Param

Change just one param

Problem is the number of gradient compactors is equal to the number of parametes

note: $C(w,b)$ is a shared value so not repeated but the rest is

Complexity Calculation →
network w/ 784 inputs, 30 hid, 10 output
weights: $784 \times 30 + 30 \times 10 = 23,820$
Bias: $30 + 10 = 40$

each gradient comp needs 23,861 cost functn calcs per gradient
& there is many iterations of gradient

Cost per single image 1 ~~gradient~~ Cost: 23,820 (feedforward)
images = 60,000

$$23,820 \times 60,000 = 1.4 \times 10^9$$

then 1 gradient = 23,861 times the two for each iteration

very big number !!!

Backpropagation

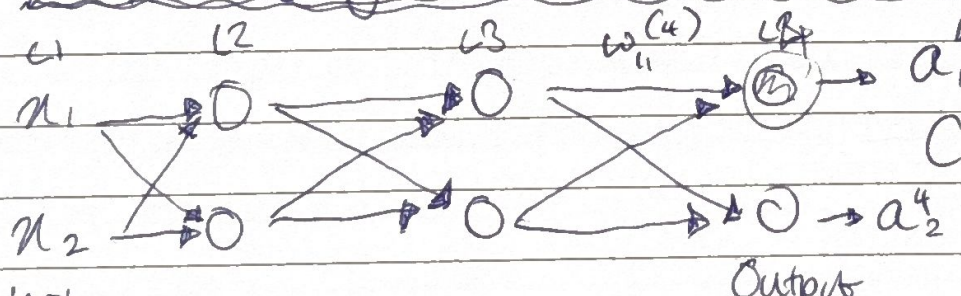
Problem w/ gradient descent is that for each iteration ~~too~~ is extremely expensive as it needs to keep calculating the cost function which is itself very comp expensive

- needs to find the derivative of the cost w/ respect to each param of the cost
  - → this requires computing the cost func with the new slightly change param $C(w + \Delta w, b)$

Key innovation = Backpropagation

Dramatically reduces cost of gradient desc

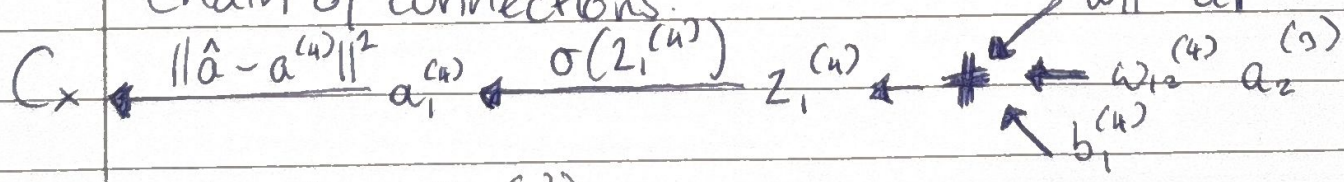
Computing gradient using chain rule



$$C_x = \|\hat{a} - a^{(4)}\|^2$$

this is the relation between a^4 & cost

Chain of connections:



$$\frac{\partial C_x}{\partial a_1^{(4)}} = 2(\hat{a}_1 - a_1^{(4)})$$

If we select a certain param we want to calculate, e.g $w_{11}^{(4)}$, we can use the chain rule to calculate

Chain Rule: $\frac{\partial C}{\partial w_{11}^{(4)}} = \frac{\partial C_x}{\partial a_1^{(4)}} \cdot \frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} \cdot \frac{\partial z_1^{(4)}}{\partial w_{11}^{(4)}}$

Here there are 3 derivatives to solve:

$\frac{\partial C}{\partial a_1^{(4)}} = 2(\hat{a}_1 - a_1^{(4)})$ from cost

$\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} = \sigma'(z_1^{(4)})$ from $\sigma(z_1^{(4)})$

$\frac{\partial z_1^{(4)}}{\partial w_{11}^{(4)}} = a_1^{(3)}$ from $w_{11}^{(4)} a_1^{(3)}$

By chain rule calc =

$\frac{\partial C_x}{\partial w_{11}^{(4)}} = 2(\hat{a}_1 - a_1^{(4)}) \sigma'(z_1^{(4)}) a_1^{(3)}$

All of these terms are calculated using the existing activation values, instead of recalcing a cost function

$\sigma'(z_1^{(4)})$ does not just use activation but it is the derivative of the sigmoid

Deriv of sigmoid function

$\sigma(z) = \frac{1}{1+e^{-z}}$

$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} \rightarrow$ Simplified $\sigma(z)[1 - \sigma(z)]$

So for a sigmoid neuron:

$a = \sigma(z)$ $\quad$ a = output of neuron

$\sigma'(z) = a(1-a)$

So

$$\frac{C_x}{\partial w_{11}^{(4)}} = 2\left(\hat{a}_1 - a_1^{(4)}\right) \cdot a_1^{(4)}\left(1 - a_1^{(4)}\right) \cdot a_1^{(3)}$$

~~cost~~ Deriv of cost w/ respect to $w_{11}^{(4)}$ using just values in the feedforward

Just calcing the product of activations that are already known

▲ call this $\delta_1^{(4)}$ w/out the $a_1^{(3)}$ component

$\delta_1^{(4)}$ can be reused for all $^{(4)}$ terms from the neuron

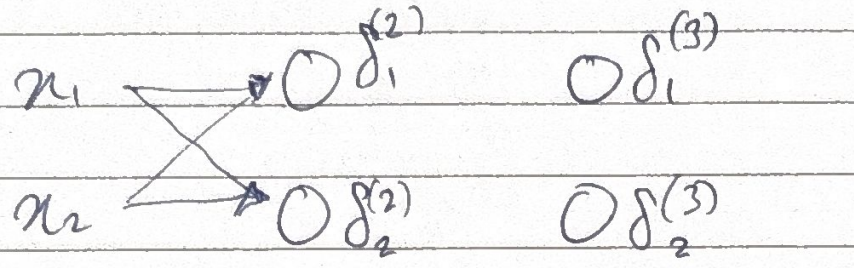
$$\frac{\partial C_x}{\partial w_{11}^{(4)}} = \delta_1^{(4)} a_1^{(3)} \qquad \frac{\partial C_x}{\partial w_{12}^{(4)}} = \delta_1^{(4)} a_2^{(3)}$$

& the bais $\frac{\partial C_x}{\partial b^{(4)}} = \delta_1^{(4)} \cdot 1$

* Repeat for the other output value $a_2^{(4)} \rightarrow \delta_2^{(4)}$

All computed without finite differences

now, if we know the δ for any ~~neuron~~ layer we can find the derivatives of the cost w/ resp to the params of that layer



thus for any weight or bais in net:

$$\frac{\partial C_x}{\partial w_{jk}^{(\text{layer})}} = \delta_j^{(L)} a_k^{(L-1)} \qquad \frac{\partial C_x}{\partial b_j^{(L)}} = \delta_j^{(L)}$$

(8)

How to find Deltas for previous layers? $\delta^{(3)}$

Can once again use chain rule

If you know δ for output layer
& Activations of the previous layer
& the known weights of the current layer
you can find deltas of previous layer

$$\delta_j^{(4)} = \sum_k \delta_k^{(L+1)} w_{kj}^{(L+1)} a_j^{(L)} \left(1 - a_j^{(L)}\right)$$

(1) In Backprop, you calc the deltas for output layer Ns
(2) usin these Deltas iteratively calc previous deltas using just deltas & activation values
(3) using deltas find derivatives of all weights & bias

No finite differences, just feedforward

Gradient Descent w/ Backprop

1. Init network w/ weights & biases (~~RE~~)
2. for each train image
   - Compute activations (FeedForward)
   - Compute δ Delta for output layer
     $\delta_j^{(4)} = 2(\hat{a}_j^{(L)} - a_j^{(4)}) a_j^{(L)}(1 - a_j^{(L)})$
   - Compute δ Deltas for all previous layers
   - Compute gradient of cost w/ respect to ~~the~~ all weight & bias for ~~Each~~ the train image
3. Average the gradient w/ resp to each weight & bias over the entire training set

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum \frac{\partial C_x}{\partial w_{jk}^L} \qquad \frac{\partial C}{\partial b_j^L} = \frac{1}{n}\sum \frac{\partial C_x}{\partial b_j^{(L)}}$$

4. update the weights & bias using gradDesc

$$w_{jk}^L \rightarrow w_{jk}^L = w_{jk}^L - \eta \frac{\partial C}{\partial w_{jk}^L}$$

$$b_j^L \rightarrow b_j^L = b_j^L - \eta \frac{\partial C}{\partial b_j}$$

5. repeat until COST reduces to desired level

Gradient compexity w/ Backprop (FF)

Calc ~~for~~ cost of single image = 23,820 (same)

Calcs for Backprop single image = 24,210

All grads for SI: 23,820 + 24,210 (sum not product)
∇C_x = 48,030

overall injs = 60,000 × 48,030
Per iteration
$= 10^4$ improvement