Josh Starmer / Stat Quest
Neural Networks Series

15 video to cover
(More videos but outside of scaper for NNs, CNNs, RNNs, LSTM)

1	the main ideas of NNs	19
2	the chain rule	19
3	Gradient Descent	24
4	Backprop main ideas	18
5	Backprop details I - opt 3 params simult	19
6	Backprop details II - chain rule	14
7	Relu	9
8	Multi-inputs & Outputs	14
9	Argmax & Softmax	15
10	Softmax derivative	8
11	Cross Entrophy	10
12	Cross Entrophy Derivatives & Backprop	23
13	Convolutional NNs	16
14.	Recurrent NNs	17
15	Long Short-term memory	21
		246
		4.1 hours

The main ideas of NNs

can fit to non-linear distributions

Activation function is what allows this
Softplus, Relu, Sigmoid

Input → Hidden → Output nodes

linear calc of input * weight + bias = x-axis coord

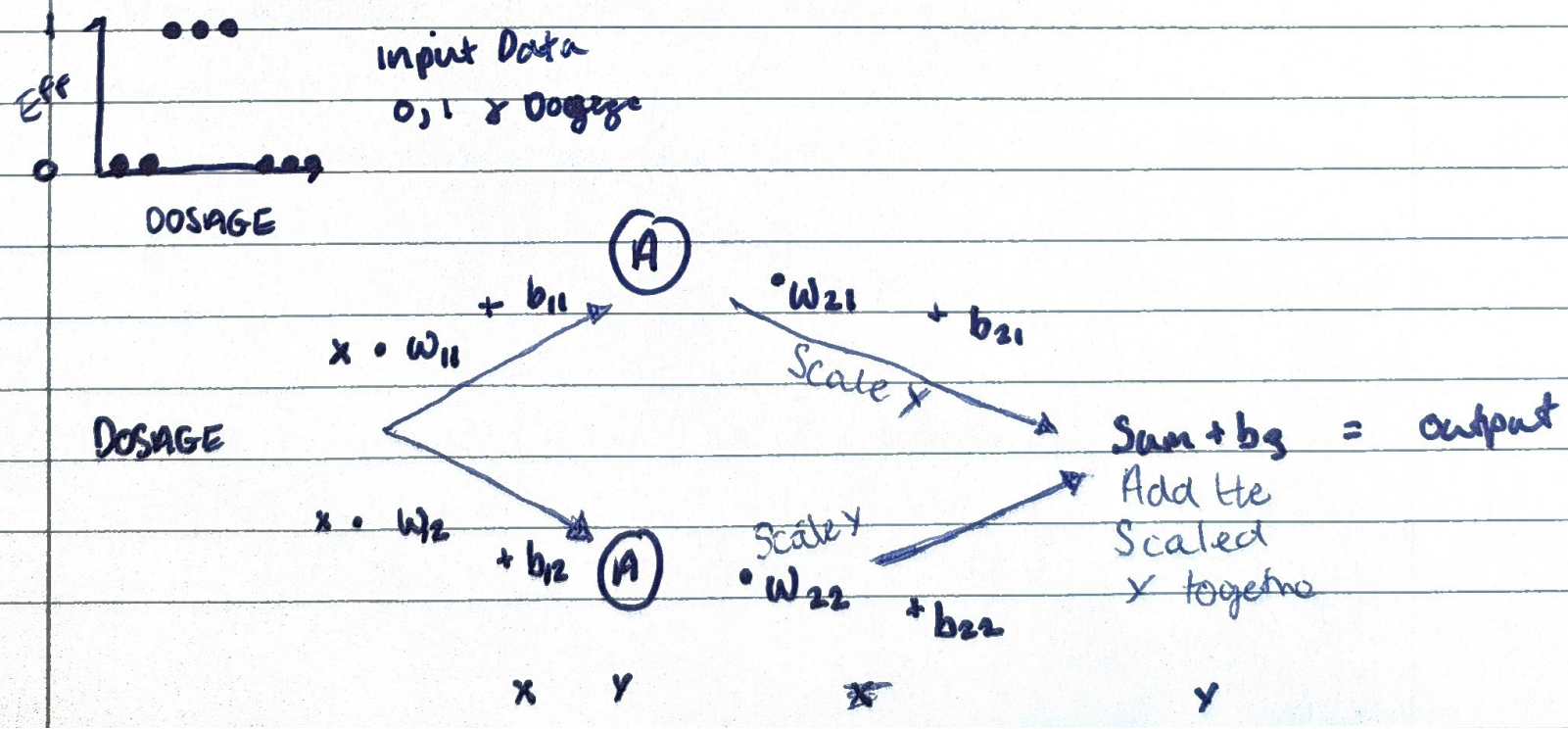
Put x into activation function = y-axis coord

the y-axis value is what is then carried into the next layer & the process repeats

Plot y-axis curves to see the space the area of the network is working in

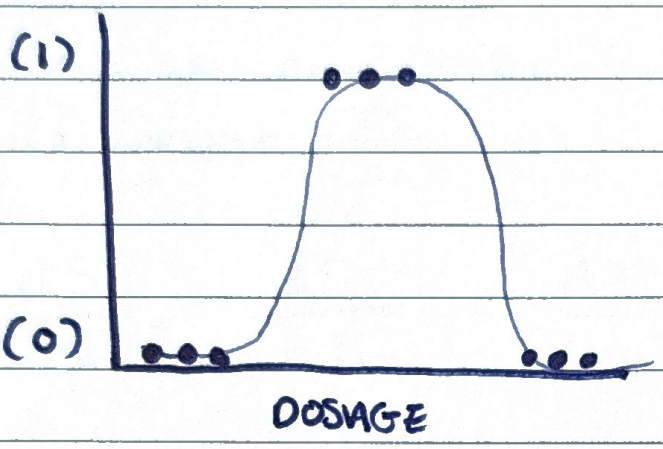
Add together the y-axis curves from different routes to create a new curve
(And minus the bias).

This is the output of network that fits the data



two options now:

- Plug new values into network to get output
- Plot the graph the network creates & read From it



output Plot of network

fix weights, bias, nodes

(2) Chain Rule

Assumes basic understood derivatives

Chain is uses for interwined relationships

Weight & height , height & shoe size

using ~~height~~ weight to predict shoe size

1 unit increase in weight = 2 unit increase in height

$\frac{dH}{dw} = \frac{2}{1} = 2$ So Height $= \frac{dH}{dw} \times W = 2 \times w$

1 unit increase in H $= \frac{1}{4}$ increase in W.

$\frac{1}{4}$ $\frac{DS}{DH} = \frac{1}{4}$ $S = \frac{DS}{DH} \times H = \frac{1}{4} * H$

the two can then be interwoven

$$S = \frac{dS}{dH} * H * \frac{dH}{dW} * W$$

now we can take the derivative of shoesize w/ resp to weight $\frac{DS}{DW}$

$$\frac{dS}{dW} = \frac{dS}{dH} * \frac{dH}{dW}$$

(3) Gradient Descent

Alot of ML is about optimization

Pred = Intercept + Slope * weight

to start, ~~Pick~~ an Plug in a Slope

then pick an intercept @ random. We are going to use GD to find the correct intercept

Provides a guess for GD to improve from

GD: Calculate loss function of ~~for~~ data points
- Against the prediction line

Residual = ~~Pred~~ Actual - estimate

Sum of squared residuals = loss function

Plot graph of loss func w/ diff intercepts

this demonstrates that there is a curve & at some point exists an optimal intercept to achieve lowest loss function

But How to move to this point?
- GD

take the derivative of the loss function w/ respect to the intercept

e.g loss function =

$$(1.4 - (int + Slope \times Pred))^2 + +$$

(use chain rule)

$$\frac{d}{d int} \text{Sum of Squared residual}$$

GD uses the derivative to find the optimal point

Plug the intercept (0) into the derivative to get the Slope of the curve at that point

note when close to the optimal point the Slope will be close to 0

higher Slope when further away

this provide the size of the "Step" for GD to move the intercept for a better value

GD takes the slope, e.g. 5.7, & applies a "learning rate" e.g. 0.1

Step Size = -5.7 * 0.1 = -0.57

new Intercept = old - Step
0 - (-0.57) = 0.57

then iterate to learn again

How many steps to take? e.g.

Stop when step size = 0.0001

How to estimate Slope & Intercept?

Same as before but this take the Deriv of the loss function & also w/ resp to intercept AND ALSO the Slope

$\frac{d}{d\,inter}$ & $\frac{d}{d\,Slope}$

having to derivatives of the same function is known as a gradient

use gradient to descend to lowest point in Loss F
- Hence the name

Pick random intercept (0) & Slope (1)

Plug these into the 2 Derivatives & get out 2 Slopes

turn these Slopes into Steps w/ LR

Step Size$_I$ = Slope * LR
Step Size$_S$ = Slope * LR

use the Slope Size to update both I & S

Repeate steps until Step Size is really ~~small~~ small

Steps:

1. take Derivative of the Step Function for each Param inside of it
2. Pick Random starting values for the parameters
3. Plug the params into derivatives
4. Calculate the Step Sizes
5. calculate/update the parameters

   New P = old P - Step Size

6. keep repeating 3-5 until ~~same~~ small Step Size

(4) Backprop Main ideas

(1) use chain rule to calc Derivs
(2) use gradient Desc to update

Backprop starts w/ the last param

Recall that activation funcs output y-axis coords which we can plot

each ~~net~~path in the net creates its own y-axis curve

- A curve is obtained by plugging in several inputs, e.g 0-1 in 0-1

A curves FROM DIFF routes together to get final fitting curve

+ Add final bias
  - what this example wants to optimize

Let as random value 0

Calculate loss function against network

Plot loss against bias (0)
  - increment intercept to demonstrate how intercept can change loss func

what to use gradient descent to make this movement

Find the derivative of the loss funchan w/ respect to intercept

$$\frac{dLf}{db_3} = \frac{dLf}{dPred} * \frac{dPred}{db_3}$$

Chain rule to get to b_3

After follow known gradient desc rules

(5) Backprop Details Pt 1 - Opt 3 Paras Simultaneously

How to get ~~adjust~~ move optimal params
- not just the last bias

Calc b_3, w_3, w_4 @ once

Start init w_3, w_4 w/ random vals

Init B_3 as 0
- not it is common to init B_3 intercepts as 0

Run network w/ input & get output using inited params

Calculate the loss funcnan

use same steps as before to GD b_3

Important point here is that even though we are optimizing further down the tree the calcs or before are the same

how to calc Derivatis for w_3, w_4?

breakdow how we get predicheat vale

Pred = $Activ_1 * w_3 + Activ_2 * w_4 + b_3$

↳ This valer is used to create the LF

And writting it this way Demonstrates how to link the LF to w_3, w_4

Can now use chain rule to:

$\frac{dSSR}{dw_3}$ $\frac{dSSR}{dw_4}$

$\frac{dSSR}{dw_3} = \frac{dSSR}{dPred} * \frac{dPred}{dw_3}$ → easy to Derive - Pred Derived by w_n

$\frac{dSSR}{dw_4} = \frac{dSSR}{dPred} * \frac{dPred}{dw_3}$

↓

note uses the same calc as eachother but Also the same from the b_3 Calc

Now we have 3 Derivatis to plug into a gradient Descend

$\frac{dSf}{dw_3} = \sum_i -2 \times (Obs_i - Pred_i) \times Activ_1$

Expand, Plug observed, Plug Observed, Plug vale

Output is a slope value
→ Step Size

Repeat for w_4, b_3

Apply GD updating process

(6) Solving all parameters

Optimize all weights & bias

route to get to w1:

$$\frac{dLF}{dw_1} = \frac{dLF}{dPred} * \frac{dPred}{dy_1} * \frac{dy_1}{dx_1} * \frac{dx_1}{dw_1}$$

dy_1: $y_1 \cdot w_3 + y_2 \cdot w_4 + b_3$

dx_1: activ funct $\log(1+e^x)$

dw_1: input $* w_1 + b_1$

follow the connections back

note ALL derivatives are the same structure regardless of the number/function seen

this is why backprop can be easily coded

Plug all derivs together to get a new functn of the Deriv $\frac{dLF}{dw_1}$

Repeat process for b_1

Note the steps repeat MOST of the same steps as w_1

Same steps for w_2 & b_2

then use GD to opt all params together

Starts by init weights & bias (many ways to do this)
- weights = rand norm dist
- bias = 0's

Expand Derive summations and plug values to get predicted values (from network)

Solve each derivative & turn into step size
Step size = Derivative * learning rate

& update params
new = old - Step Size

Repeat until criterea met
(Step Size < x)

(7) Relu in action

Activation function

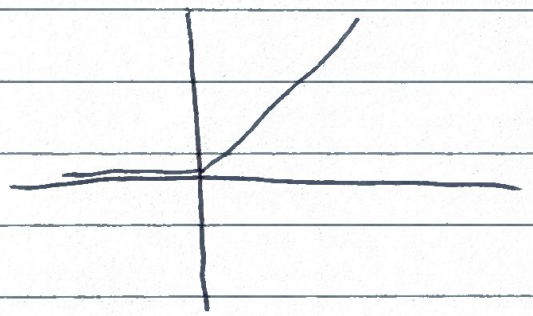
Note can also use activation functions at the final output too

$y = f(x)$

Relu = Max(0, x)

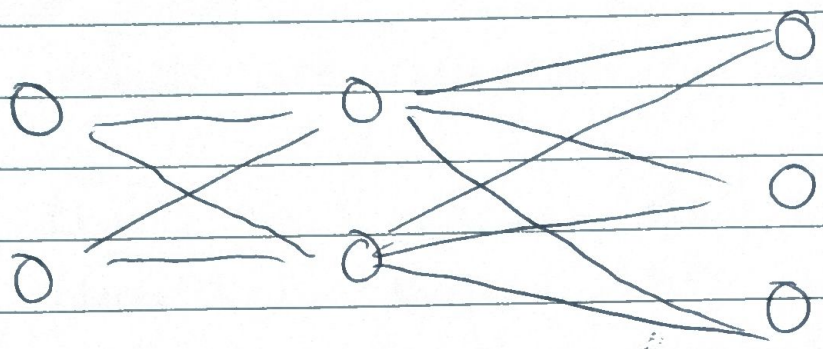
if less than 0 [redacted] then Relu = y = 0

if >0 they = y = x



visual

(3) Multiple inputs & outputs



All interconnected w/ weights

for the output active func common to covert values to 0-1

this 1 is more likely to be the correct value

Select the highest (Argmax or Softmax)

(9) Argmax & Softmax

labels are denoted by a 1

thus nn outputs for categorical data should be below this hence normalizing activ func between 0 & 1

or use Argmax

Argmax

- sets all outputs to 0 or 1
- largest 1, rest = 0
- then can be used w/ loss function
  - (obs - Pred)
- this can't be used to optimize a nn become the deriv of 0 is 0
- no info ⇒ no grad desc step

Softmax

- Argmax for output, Softmax for training

$$\frac{\text{values} \leftarrow \text{one cat}}{v_1 + v_2 + v_3 \leftarrow \text{all cats}}$$

- Always ≥ 1 & sums to 1 over all cats
- can be interp as "probs"
- Dont trust as true probs

$$\frac{e_i}{\sum e_j}$$

- Derive of SfMax is not always 0 & hence suitable for gDesc

(10) Soft-Max Derivative

Derivative of category X

with respect to the

Derivative of the raw output value of Cat X

$$\frac{d \text{ "P cat x"}}{d \text{ Raw Setosa}} = \text{"P cat x"} \times (1 - \text{"P cat x"})$$

"P cat x" is the probability from the SF

↳ Derive of this use Quotient rule to calc (complicated)

raw output is just the pre-SF value

Important = each SF outvalue relys on all of the categories

$$\frac{d \text{ CAT A}}{d \text{ raw A}} \qquad \frac{d \text{ CAT B}}{d \text{ raw B}} \qquad \frac{d \text{ CAT C}}{d \text{ raw C}}$$

CAT A + CAT B + CAT C

So need to take the deriv w/ respect to ea

P(CAT A) × P(CAT B) easier than own Cat

(11) Cross Entrophy Loss function

W/ one output we often use Sum Sireds

W/ >1 we want to use Argmax as the output activation

↳ has a Bad Derivative so cant use fro Backprop

So we use Softmax which outputs values as probabilities, 0-1

when outputs are bounded between 0-1 we use cross entrophy CF

Raw output → Softmax Probs

cross ent = $-\log(\text{Softmax})$

note $-\log(x)$ isnt the Standard form of cross entrophy

NNs only need a Simplified version of it

obtained by expanding & summing

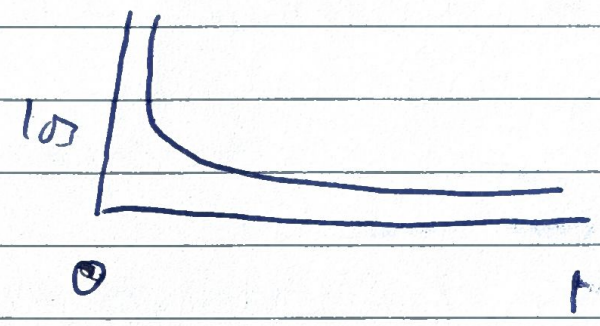
$-\sum_{n}$ observed$_i$ x log(Predicted) = full

total error = Sum($-\log(SF)$)
loss func ^

From the indiv cross ent values we are meant to calc the Sum of Square resids

why not just calc w/ Softmax probs?

Softmax probs = 0, ϕ

Cross ent = 

As a cross Ent gets more bad, it gets an exponentially higher value

this provides more info to learn / penalise

the slope of the derivative is larger @ bad preds

= larger step size to learn

(12) Cross Ent & Backprop

raw → softmax Prob → entrophy value

cross ent depends of max

but softmax has a different calc
Depending on the observed value

Cat (n) / cat r, n, m

thus different derivs to be calced

$CE = -\log(\text{Pred Cat } n)$

Pred Cat n = Softmax(cat a, cat b, cat c)

Linking cross ent to paras
↳ find the link via softmax
Category

Derive uses chain rule

$$\frac{d\,Cat}{db_3} = \frac{d\,Cat}{d\,Prob} \times \frac{d\,Prob}{d\,Raw} \times \frac{d\,raw}{d\,b_3}$$

Plug in & simplify

When obs cat is used to calc the deriv then = prob - 1

For other cats still need to follow the route for the correct cat

this means taking the Deriv via the softmax

$$\frac{d\ cat\ 2}{d\ bz} = \frac{d\ cat\ 2}{d\ Prob\ C2} * \frac{d\ Prob\ C2}{d\ raw\ C1} * \frac{d\ R\ C1}{d\ bz}$$

Plug & simple

Result = Prob cat 1

How to optimize?

Only plug in the correct derive route for each data entry w/ respect to param

Derive of cross ent = slope

Step size = slope * learn rate

GD iterate

(13) convolutional NNs

Mainly used for image classification

convert Matrix of image into vector

this vector then proxies as the input layer

each pixel = input neuron

Plain NNs are poor for images because they do not scale well

each pixel / neuron has a weight to the next layer

big / detailed images = many pixel = many weight = too comp heavy

for large & complicated images → CNNs

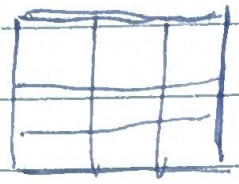
CNNs apply a filter to the image which is 3×3 pixels (kernal)

0	0	1
0	1	0
1	0	0

looks like this but is estabs via Backprop (???)

this kernal is then applied iteratively to the image

the result is a small image as

kernal  = n , i.e. 1 Pixel

q Pixels to

q Pixels overlayed w/ kernal & Dot product (each Pixel times & added together)
→ Sum of Products

After applying the kernal add a bias term

Next Slide the kernal over 1 Pixel & repeat
- the Slide of 1 Pixel can vary
- Some methods Slide more to reduce the computation cost

why is compressing the image okay?
→ Areas of a image are often correlated so Info is not really lost

After kernal conversion an activation func is applied (ReLU)

And then apply another kernal method called Max Pooling

this takes a kernal, usually of 4, and selects the highest value

usually this method has no overlap & Slides to the next empty Slot

Filter kernal → ReLu → Max Pool kernal
9 to 1 >0 4 to 1

↓ ↓

the best filter mates the input cluster

Pooling selects the most relevant // strongest value from the filter

the exists an alt method called mean pooling

finally the Max Pooled image is be converted into a vector & used as an input to a standard NN

this is fine as it is now essentially a small image

Output of NN is the classification of the data e.g. naughts or crosses

image → futer to feature Map → ReLu → Max Pool → input nodes

6x6 images is compressed down to 4 numbers

the futer kernal is the same as look @ a cluster of pixels rather then just 1

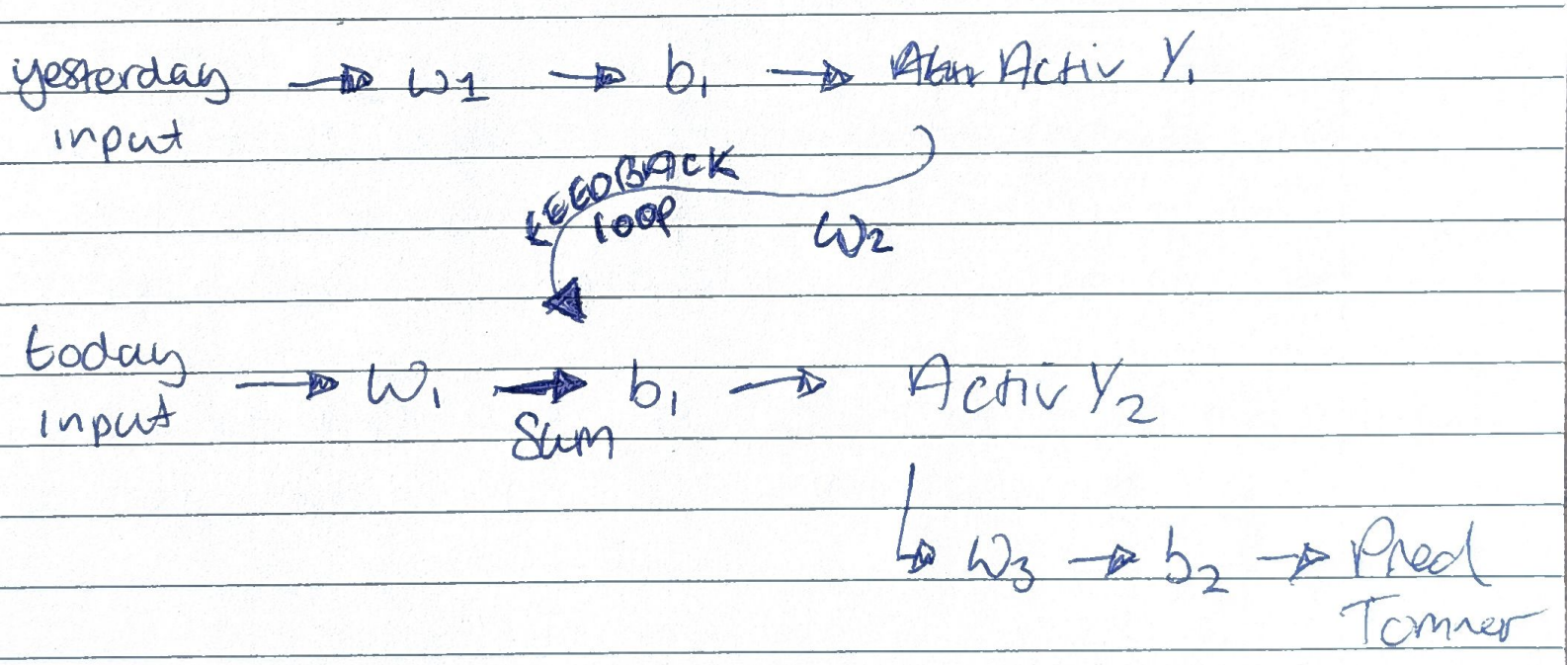
(14) Recurrent Neural Networks (RNNs)

RNNs are a stand alone thing but they also feed into
- long short term memory networks
- transformers

Varied input lengths?

RNNs have feedback loops

RNN looks like it has a single input value but the feedback loop allows for sequential or time series data



feedback loop allows additional layers to be threaded in a sequential data

the weights & biases W_1, W_2 are shared between layers

basic RNNs are not used often, why?

- the more we unroll a RNN the harder it is to train
  - known as vanishing / exploding gradient Descent problem

note W_2 is the feedback weight

if the W_2 is > 1 then the input value keeps getting multiped by factor n
= A huge number

huge number would find its way into the gradient Descent & step sizes wouldn't be incremental

Solution could be to limit W_2 to < 1
but this leads to vanishing gradient

= Steps are too small

LTSM NN are a solution & extension to this

(15) Long Short-term Memory (LTSM)

Solution to RNNs & Precursor to transformed

recap Vanilla RNN = feedback loop for seqdata
= explode or vanish gradient

RNNs = hard to train

LSTM designed to avoid

has two predicta Loop routes:

- feedback loop is used for short-term
- weights direct to output for long term

LTSM is a much mae complicated unit

uses Sigmoid & Tanh H activ funcs

Sigmoid = takes x-axis coord & creates y-axis between 0-1

tanh = x to y = -1 to 1

longterm has a lack of weights & bis

Short-term updates the long-term memory
- ST uss Sigmoid activ to be applied to LT
- So update can be thought as % being remembered

next unit of ST = Potential

- uses tanh to create a value -1 to 1
- & Sigmoid to % the potential value
- which then gets add to the LT

Updating the long term memory
- Sigmoid

Init input twice for LT & ST

for sequentials, only the ST is inputted

the NN rolls in "units" updating the LT & ST as it goes and inputting the seq ST data