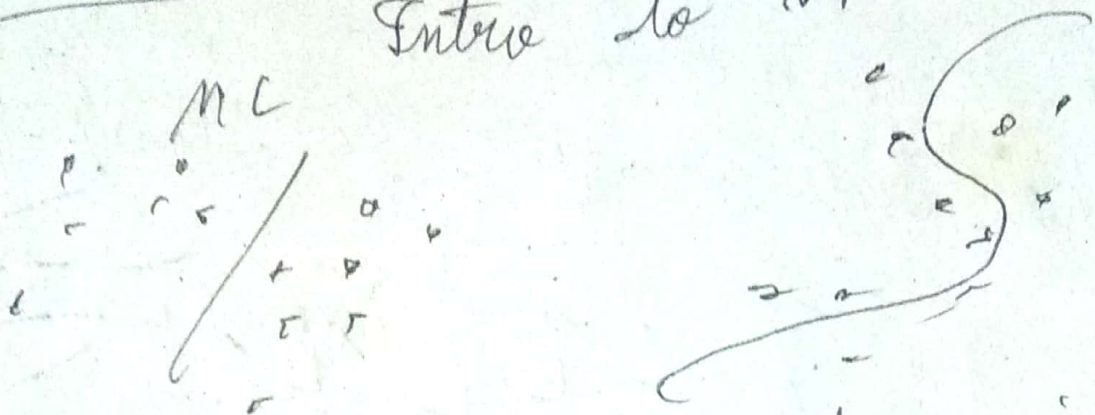
the people rather than Height of all the people of just one person.

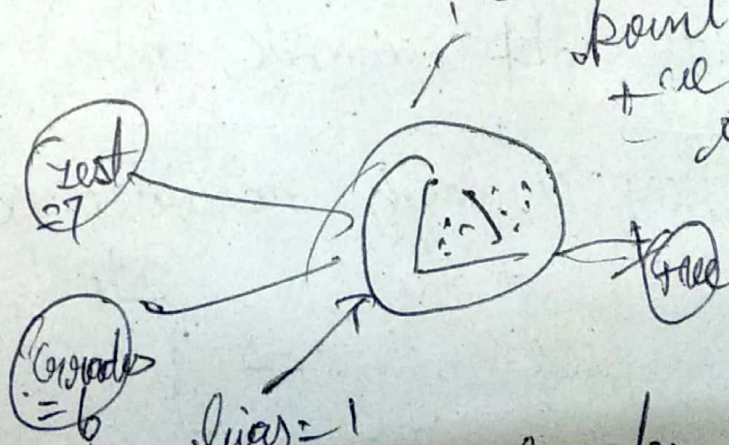
Intro to NN : Neural network



n-dimensional space, boundary is a n-1 dim hyperplane which is a high dimensional equi of line or plane in 2D & 3D respectively.

$w_1x_1 + w_2x_2 \dots w_nx_n + b = 0$ & it can be abbreviated to $wx + b = 0$.

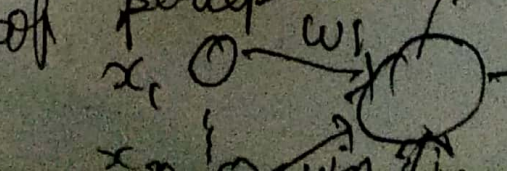
Perceptron :



$2 * Test + 1 * Grades - 18$

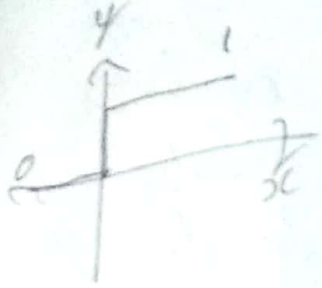
$wx + b = \sum_{i=1}^{n} w_i x_i + b$

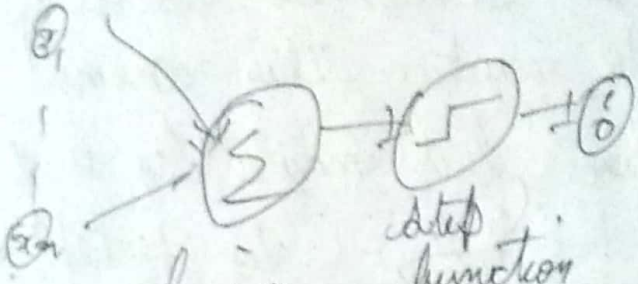
Generic case of perceptron



do here we use a step function i.e.

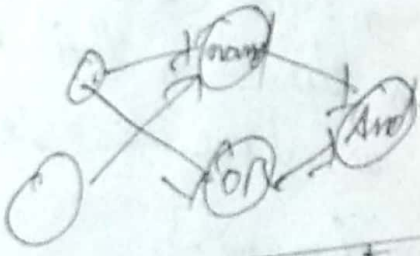
$$y = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$





Neurons takes nervous impulses as input through dendrites & output them using axon.

nucleus → axon.



Perceptron algorithm

1) start with random weights:
$w_1, \ldots, w_n, b$

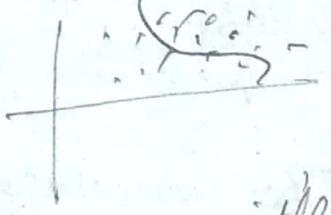
2) For every misclassified point
$(x_1, \ldots, x_n)$:

- If prediction is 0
  for $i = 1 \ldots n$
  change $w_i + \alpha x_i$
  change $b + \alpha$
- If prediction is 1:
  for $i = 1 \ldots n$
  change $w_i - \alpha x_i$
  change b to $b - \alpha$

4 5 -10

Non-linear regions :-

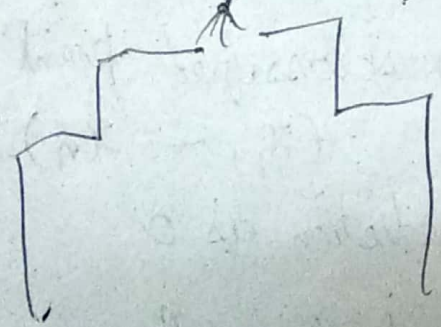
What if grade is 1 but test is 9. This would be classified as positive but we do not want to accept a person with low test scores. So we can classify it to be negative. This cannot be done by a line but maybe by a curve -,



- do perceptron won't work & we have to generalize so it'll work for other types of curves.

Log-loss error function: Mountain example - to get down we look at directions & take step in direction that helps us descend the most. It is possible to get stuck at valley (local minimum).

Discrete Vs continuous



- here we always see constant errors & we are confused & not sure what to do.

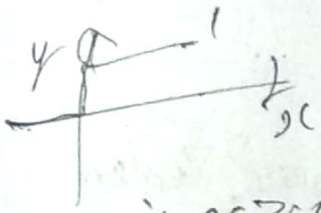
But in a continuous scenario one can detect small variations in height & we can figure in what direction it decreases the most.

Do our error function cannot be discrete but continuous (i.e. differentiable).

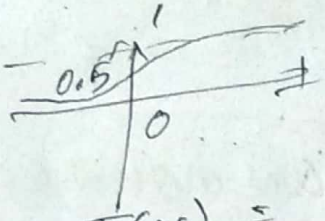
discrete vs continuous: eg: discrete - yes or no answer

continuous - a number probably between 0 to 1. (continuous).

A simple way to move from discrete to continuous predictions is to shift from step function to sigmoid activation function.



$$y = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$



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

Multi class classification

It is equivalent to sigmoid function but for problems with 3 or more classes.

exp (exponential) is a function that returns the number for every input given for - ve numbers.

	score	
beaver	2	these do not add to 1, But prob must add to 1. do we can divide each number by sum of all numbers. eg: $\frac{2}{2+1+0} = \frac{2}{3}$
duck	1	
weasel	0	

or $\frac{1}{2+1+0}$. But there's a probability

that scores are -ve & hence could lead to division by 0 error. So instead we use exp as it turns all numbers to +ve numbers. So e^2 - This gives us the softmax $e^2 + e^1 + e^0$ function.

For N classes & a linear model that gives us scores.

for i in range(len(L))
np.exp(L[i])

$$P(\text{class}_i) = \frac{e^{z_i}}{e^{z_1} - e^{z_n}}$$

np.sum(np.exp(L))

---

one hot encoding

example	Duck?	Beaver?	Walrus?
Duck	1	0	0
Beaver	0	1	0
Walrus	0	0	1

Maximum likelihood :- usually the best model is the one that assigns higher probabilities to events that happened to us. This method is called a maximum likelihood.

Maximizing probabilities using math

Maximizing the probability is equivalent to minimizing the error function.

For finding the best model we consider prob of colors as independent events & multiply them - The product that yields

highest prob is the last one. However products of millions of data points are dead. Instead it can be converted to a sum problem using log.

cross-entropy

log(1) = 0 so log (number between 0 to 1) as -ve. so if we take -ve of log of probabilities we get a +ve number.

eg: ln(0.6) = -0.51 but -ln(0.6) = 0.51.

sum of -ve log of numbers is called cross entropy. good algo - low cross entropy.

bad algo - low high cross entropy.

cross entropy is simple. If we have a bunch of events & probability. how likely is that the events happen based on the probabilities. If it's very likely we have low cross entropy. less likely? high entropy.

	Door 1	Door 2	Door 3	$0.8 \times 0.7 \times 0.1$
P(right)	(0.8)	(0.7)	0.1	$= 0.056$
P(wrong)	0.2	0.3	(0.9)	5%

Events with high prob - have low cross entropy & with low prob have high

Door 1 Door 2 Door 3 ✓ - right, x - no right

✓ 0.8 ✓ 0.7 x 0.9 $-\ln(0.8) - \ln(0.7) - \ln(0.9)$

$y_i = 1$ if present on door i

Door 1 $y_1 = 1$ $p(\text{right}) = p_i$

Door 2 $y_2 = 1$ $p_1 = 0.8$

Door 3 $y_3 = 0$ $p_2 = 0.7$

$p_3 = 1 - p_3 = 0.1$

So formula cross entropy $= -\sum_{i=1}^{m} y_i \cdot \ln(p_i) + (1-y_i) \ln(1-p_i)$

cross entropy tells us how similar are the vectors.

Multi-class cross entropy:

Animal	Door 1	Door 2	Door 3
Duck	0.7	0.3	0.1
Beaver	0.2	0.4	0.5
Walrus	0.1	0.3	0.4

The numbers in columns should add to 1 but nos. in rows need not add to 1.

Door 1 Door 2 Door 3

Duck Walrus Walrus

$P = 0.7 * 0.3 * 0.4 = 0.084$

$CE = -\log(0.7) * -\ln(0.3) * -\ln(0.4) = 2.48$

y_{ij} = 1 if Duck behind door j

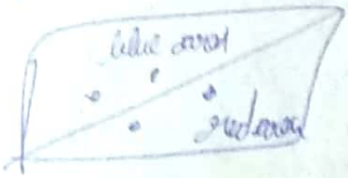
y_{2j} = 1 if Beaver , ,

y_{3j} if wolf , , ,

Cross entropy $= -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} \ln(p_{ij})$

m - number of classes.

Logistic regression :



If $y = 1$

$P(\text{blue}) = y^?$

error $= -\ln(y^?)$

If $y = 0$ (point is red) $P(\text{red}) = 1 - P(\text{blue})$

$= 1 - y^?$

error $= -\ln(1 - y^?)$

So error $= -(1-y)\ln(1-y^?) - y\ln(y^?)$

Error function $= \frac{-1}{m}\sum_{i=1}^{m} (1-y_i)\ln(1-\hat{y}_i) + y_i \ln(\hat{y}_i)$

Here we are considering average.

If activation is sigmoid then our error function should be in terms of w's & b.

$$E(w,b) = \frac{-1}{m}\sum_{i=1}^{m} (1-y_i)(\ln(1-\sigma(ax^{(i)}+b)) + y_i \cdot \ln(\sigma(ax^{(i)}+b))$$

Above is for binary class.. For multiclass it is

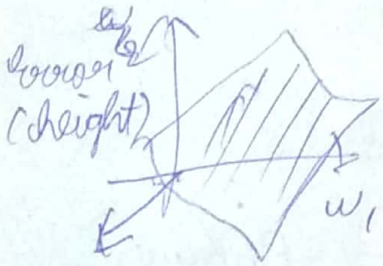
$$\frac{-1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_{ij} \ln(\hat{y}_{ij})$$

Minimizing error function.

$$E(w, b) = -\frac{1}{m}\sum_{i=1}^{m} (1-y_i)\ln(1-\sigma(wx^{(i)}+b)) + y_i \ln(\sigma(wx^{(i)}+b))$$

We update weights & biases such that we get reduced error $E(w', b')$.

Gradient descent



∇E = vector sum of gradient partial derivatives of E w.r.t w_1 & w_2

Gradient gives us the direction in which error increases more. So taking -ve of ∇E tells us in which direction we must move to decrease the error.

dE/dw_2 ↑ → ∇E

$\hat{y} = \sigma(wx+b)$

$\hat{y} = \sigma(w_1 x_1 + \dots + w_n x_n + b)$

dE/dw_1 →

$\nabla E = (\partial E/\partial w_1, \dots, \partial E/\partial w_n)$

$-\nabla E$

$\alpha = 0.1$ Taking a step is same as updating w & b.

$$w_i' \leftarrow w_i - \alpha \, \partial E/\partial w_i$$

$$b' \leftarrow b - \alpha \, \partial E/\partial b$$

$\hat{y} = \sigma(w'x + b')$ - better.

$\sigma'(x) = \frac{\partial}{\partial x}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{(1+e^{-x})^2}$

$= \frac{1 \cdot e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \cdot \frac{1}{(1+e^{-x})^2}$

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

$= \sigma \cdot \left(1 - \frac{1}{1+e^{-x}}\right)$

$= \sigma(1-\sigma)$

For m points labelled $x^{(1)} \ldots x^{(m)}$

$E = \frac{-1}{m} \sum_{i=1}^{n} y_i \ln(\hat{y}_i) + (1-y_i)\ln(1-\hat{y}_i)$

where $\hat{y}_i = \sigma(wx_i + b)$

$\nabla E = \left(\frac{\partial}{\partial w_1} E, \frac{\partial}{\partial w_n} E, \frac{\partial}{\partial b} E\right)$. To simplify we think of error each point produces & calculate derivative of error. The total error is average of error at all these points. Error at each point is $= E = -y \ln \hat{y} - (1-y)\ln(1-\hat{y})$

$\frac{\partial}{\partial w_j} \hat{y} = \frac{\partial}{\partial w_j} \sigma(wx+b) = \sigma(wx+b)(1-\sigma(wx+b)) \cdot \frac{\partial}{\partial w_j}(wx+b)$

$= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(wx+b) = \hat{y}(1-\hat{y}) \frac{\partial}{\partial w_j}(w_1 x_1 - w_n x_n + b)$

$= \hat{y}(1-\hat{y})\, x_j$

i.e. $w_j x_j$ is the only term (not constant) w.r.t w_j. So $\frac{\partial}{\partial w_j} E = \frac{\partial}{\partial w_j} [-y \ln \hat{y} - (1-y) \ln(1-\hat{y})]$

$$= \left[-y \frac{1}{\hat{y}} \frac{\partial}{\partial w_j} \hat{y} - (1-y) \frac{\partial}{\partial w_j} \ln(1-\hat{y}) \right]$$

$$= -\frac{y}{\hat{y}} (\hat{y}(1-\hat{y}) x_j) - \frac{(1-y)}{(1-\hat{y})} (\hat{y}(1-\hat{y}) x_j)$$

$$= -y(1-\hat{y}) x_j + \hat{y}(1-y) x_j$$

$$= -y x_j + y \hat{y} x_j$$

$$= -(y - \hat{y}) x_j$$

$\therefore \frac{\partial}{\partial b} E = -(y - \hat{y})$ (similar calculation)

In summary gradient is $\nabla E = -(y-\hat{y})(x_1, \ldots, x_n, 1)$

Gradient descent step :

$$w_i' \leftarrow w_i - \alpha(-(y-\hat{y}) x_i)$$

$$\Rightarrow w_i' \leftarrow w_i + \alpha(y-\hat{y}) x_i$$

$$b' \leftarrow b + \alpha(y-\hat{y})$$

Logistic reg algorithm

Pseudo code in next page

1) start with random weights:
$w_1, \dots w_n, b$

2) For every point $(x_1, \dots x_n)$:

2.1 For $i = 1$ to n
update $w_i' \leftarrow w_i - \alpha(y - \hat{y}) x_i$

$b' \leftarrow b - \alpha(y - \hat{y})$

Repeat until error is small.

Perceptron vs gradient descent

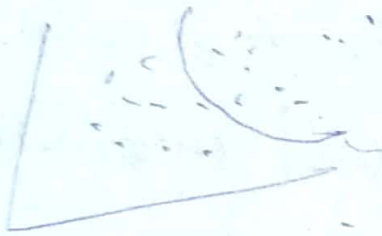
Gradient descent
change
w_i to $w_i' + \alpha(y - \hat{y}) x_i$

Perceptron algo-
In perceptron not every point changes the weight only misclassified ones
If x is classified
change w_i to $w_i + \alpha x_i$ if +ve
$w_i - \alpha x_i$ if -ve
If correctly classified
$y - \hat{y} = 0$
If misclassified.
$y - \hat{y} = 1$ if positive
$y - \hat{y} = -1$ if negative

Are both same?
The main diff is that in gradient descent $\hat{y}$ can take any values.

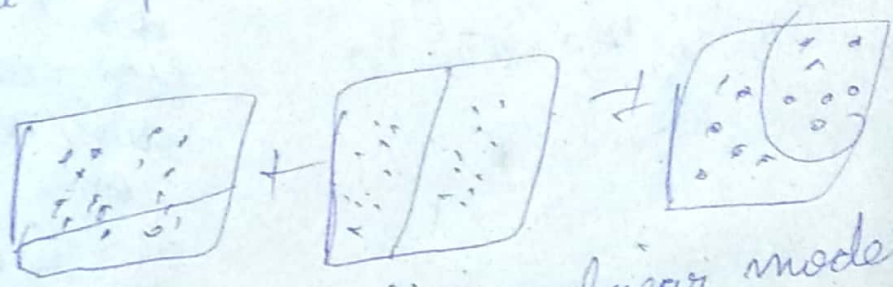
If a point is correctly classified perceptron algorithm stops. But in gradient descent a misclassified point tells the line to come closer & correctly classified point tells it to go farther away.

Non linear regions



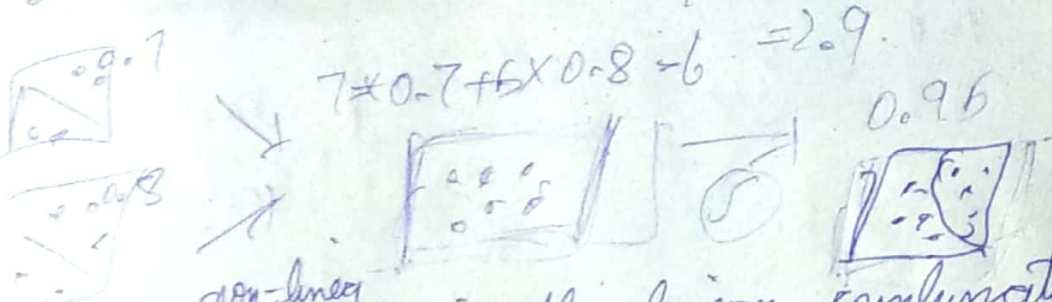
set of points equally likely. Here is where neural networks come into play.

Combining regions : We are gonna combine 2 perceptrons into a 3rd complicated one.



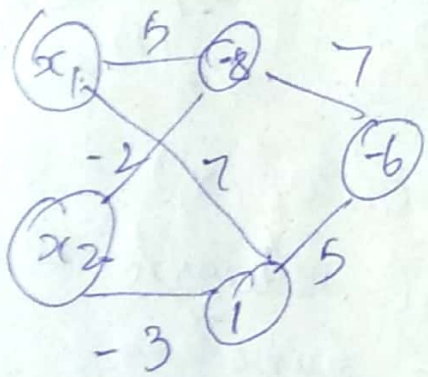
We are gonna combine 2 linear models to non linear model. We combine as follows. Consider a point in blue region in 1st diag above. It's prob in 2nd diag in blue space is 0.7. When we combine both we get 1.5 > 1. So squash it between 0 to 1. We can use sigmoid so $\sigma(1.5) = 0.81$

If we want we can add weights too to the 2 linear models.



The third model is the linear combination of the existing 2 models.

non-linear



$$\frac{1}{1+e^{-x}} = 0.88$$

$$0.88 + 0.88e^{-x} = 1$$

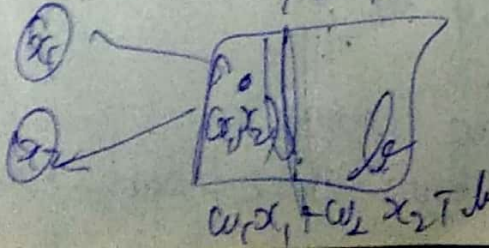
$$0.88e^{-x} = \frac{1}{0.88}$$

$$-ln(1+b) = +0.107$$

We can keep adding more layers & nodes to form complex networks. DNN's are where linear models combine to form non linear models which further combine to form even more non-linear models.

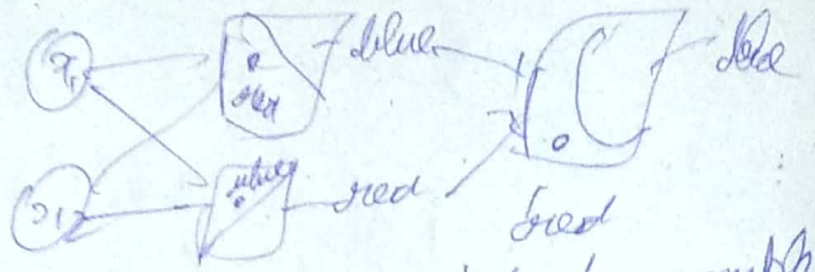
Feedforward

Feedforward is the process neural networks use to turn input to output. ↳ plots & outputs probability.



$x = (x_1, x_2)$

$y = 0$



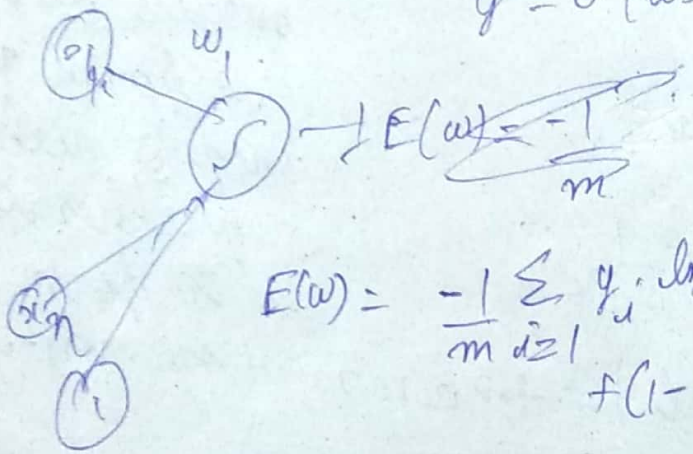
For a neural net with 2 layers

$$y^1 = \sigma \begin{pmatrix} w_{11}^{(2)} \\ w_{21}^{(2)} \\ w_{31}^{(2)} \end{pmatrix} \sigma \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

$$y^1 = \sigma\, w_2\, \sigma \cdot w_1 (x)$$

To train neural network we need the define error function.

$$y^1 = \sigma \cdot (wx + b)$$



$$E(w) = \frac{-1}{m} \sum_{i=1}^{} y_i \cdot \ln(\hat{y}_i) + (1 - y_i) \ln(1 - \hat{y}_i)$$

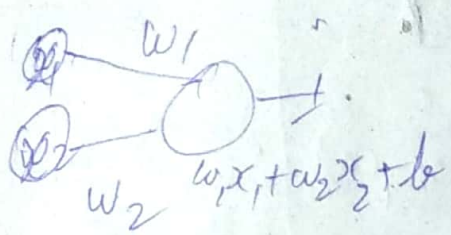
Backpropagation :-

In a nutshell backprop consists of :-

- doing a feedforward operation.
- comparing o/p of model with desired output.
- calculating error.
- feedforward or backwards to spread error

to each of the weights.
- use this to update weights.
- continue till model is good.

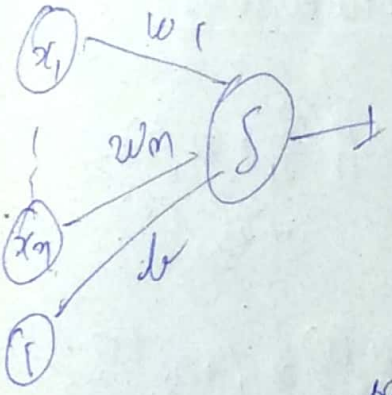
$$E(w) = \frac{-1}{m} \sum_{i=1}^{m} y_i \cdot \ln(\hat{y}_i) + (1-y_i) \ln(1-\hat{y}_i)$$



For MLP to reduce error. Calculate $E(w)$, & walk in direction of $-\nabla E$ to minimize the error.

Backprop math $\hat{y} = \sigma(wx + b)$

$$E(w) = \frac{-1}{m} \sum_{i=1}^{m} y_i \cdot \ln(\hat{y}_i) + (1-y_i) \ln(1-\hat{y}_i)$$



$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_m}, \frac{\partial E}{\partial b} \right)$$

Here $\hat{y}$

In MLP. $\hat{y} = \sigma w^{(3)} \circ \sigma w^{(2)} \ldots w^{(1)} x$

3 layers & error func is same but $\hat{y}$ is much more complex.

$$\nabla E = \left(\frac{\partial E}{\partial w_{ij}^{(k)}} \right)$$

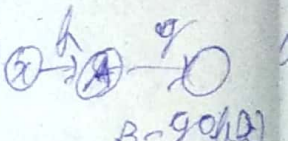
More formally for a MLP with n layers

$$\hat{y} = \sigma w^{(n)} \circ \sigma \circ \ldots \circ \sigma w^{(1)}(x)$$

$$w^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} \end{bmatrix} \quad w^{(2)} = \begin{pmatrix} w_{11}^{(2)} \\ w_{21}^{(2)} \\ w_{31}^{(2)} \end{pmatrix}$$

$$\nabla E = \begin{pmatrix} \frac{\partial E}{\partial w_{11}^{(1)}} & \frac{\partial E}{\partial w_{12}^{(1)}} & \frac{\partial E}{\partial w_{11}^{(2)}} \\ \frac{\partial E}{\partial w_{21}^{(1)}} & \frac{\partial E}{\partial w_{22}^{(1)}} & \frac{\partial E}{\partial w_{21}^{(2)}} \\ \frac{\partial E}{\partial w_{31}^{(1)}} & \frac{\partial E}{\partial w_{32}^{(1)}} & \frac{\partial E}{\partial w_{31}^{(2)}} \end{pmatrix}$$

Chain rule

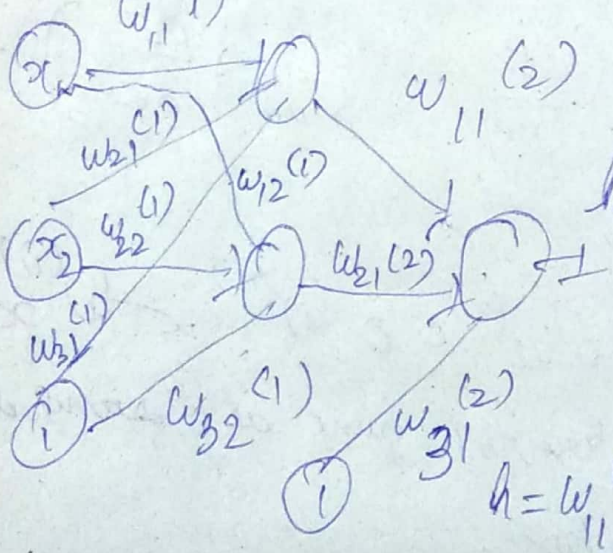


$$\frac{\partial B}{\partial x} = \frac{\partial B}{\partial A}\frac{\partial A}{\partial x}$$

$$w_{ij}^{(k)} \leftarrow w_{ij}^{(k)} - \alpha \frac{\partial E}{\partial w_{ij}^{(k)}}$$

when functions compose derivatives are multiplied. Feedforward is same as function composition & backprop is calculating derivatives & multiplication.

Backpropagation



$$h_1 = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)}$$

$$h_2 = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{32}^{(1)}$$

$$h = w_{11}^{(2)} \sigma(h_1) + w_{21}^{(2)} \sigma(h_2) + w_{31}^{(2)}$$

$$\hat{y} = \sigma(h)$$

i.e. $\hat{y} = \sigma \circ w^{(2)} \circ \sigma \circ w^{(1)}(x)$.

$$E(w) = -\frac{1}{m} \sum_{i=1}^{m} y_i \ln(\hat{y}_i) + (1-y_i) \ln(1-\hat{y}_i)$$

Now we need to calculate derivatives

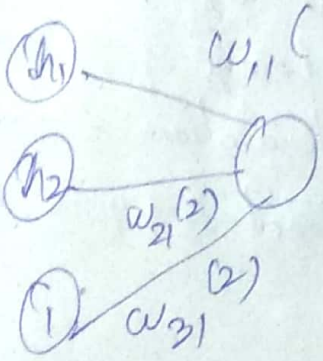
of error function w.r.t each of the weights.
using chain rule.

$$E(w) = E(w_{11}^{(1)}, \dots w_{31}^{(2)})$$

$$\nabla E = \left(\frac{\partial E}{\partial w_{11}^{(1)}} \dots \frac{\partial E}{\partial b_{31}^{(2)}} \right)$$

$$\frac{\partial E}{\partial w_{11}^{(1)}} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial w_{11}^{(1)}}$$ so

feedforward is just composition of functions
Eg: we take a small part of MLP



$$w_{11}(\cdot) \quad h = w_{11}^{(2)} \sigma(h_1) + w_{12}^{(2)} \sigma(h_2) + w_{31}^{(2)}$$

$$\frac{\partial h}{\partial h_1} = w_{21}^{(2)} \sigma(h_1)(1 - \sigma(h_1))$$

MSE vs log loss.

Sum of squared errors: $E = \frac{1}{2} \sum_{\mu} \sum_{j} (y_j^{\mu} - \hat{y}_j^{\mu})^2$

μ - set of all data points. j - output units.

First we perform inside sum over j - For each output unit we find diff between y & $\hat{y}$. sq & add up

second - for each data point find inner sum of squared diff for each output unit. Then sum of the squared diff for each data point. // Pros: 1) errors are always +ve
2) larger errors more penalized

$$y_j^{(\mu)} = f\left(\sum_i w_{ij} \cdot x_i^{(\mu)}\right)$$

$$E = \frac{1}{2} \sum_\mu \sum_j \left[y_j^{(\mu)} - f\left(\sum_i w_{ij} x_i^{(\mu)}\right) \right]^2$$

We want the network's error to be as small as possible & weights sets as tensors.

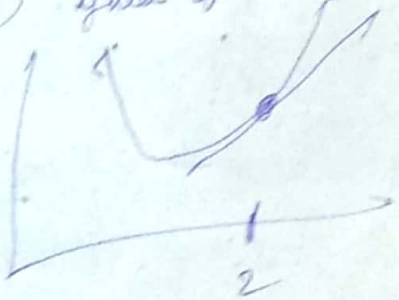
$$\hat{y}_j^{(\mu)} = f\left(\sum_i w_{ij} \cdot x_i^{(\mu)}\right)$$

We want to step in i direction that minimizes the error the most. This direction can be found by calculating gradient of squared error. Gradient is rate of change. To calculate rate of change we turn to derivatives & derivatives of a function $f(x)$ gives $f'(x)$ that returns slope of $f(x)$ at x.

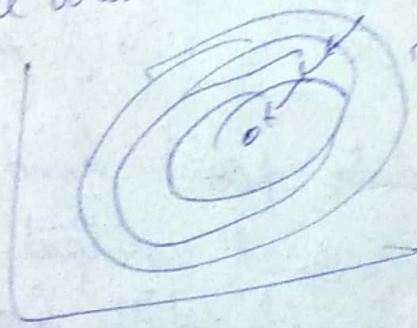
$f(x) = x^2$

$f'(x) = 2x$

$f'(2) = 4$



Gradient is just derivative extended to functions with more than one variable.



Topographical map where points on a contour line have same error & darker contours have larger errors.

: weights will go wherever the error is steepest downhill so we won't get stuck at local minimum, so we can use momentum to

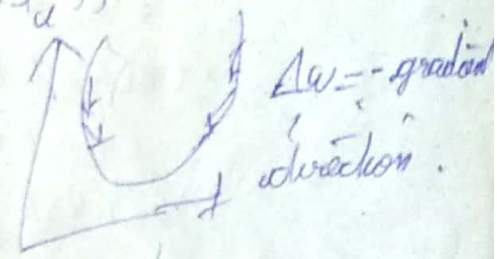
overcome it. In MSE why don't we take abs(y-ŷ) instead of square - One advantage of squaring is that it penalizes outliers more.

Gradient descent of SSE: math

$$E = \frac{1}{2}\sum_{\mu}(y^{\mu} - \hat{y}^{\mu})^2 = \frac{1}{2}\sum_{\mu}^{\mu}(y^{\mu} - f(\sum_i w_i x_i^{\mu}))^2$$

μ - data - We can think of scanning the below matrix one row at a time, calculating diff. & summing them up. Now let us consider just one example -

$$E = \frac{1}{2}(y - f(\sum a_i x_i))^2$$



$$w_i = w_i + \Delta a_i$$

$$\Delta w_i \propto -\frac{\partial E}{\partial w_i} \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

learning rate.

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}(y - \hat{y})^2 = \frac{\partial}{\partial w_i} * \frac{1}{2}(y - \hat{y}(w_i))^2$$

$$= (y - \hat{y})\frac{\partial}{\partial w_i}(y - \hat{y})$$

$$= -(y - \hat{y})\frac{\partial \hat{y}}{\partial w_i}$$ (as $\hat{y}$ depends on w_i & using chain rule $\frac{\partial}{\partial w_i}(y - \hat{y}) = \frac{\partial}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_i}$)

But $\hat{y} = h(h)$ where $h = \sum w_i x_i$

$$\frac{\partial E}{\partial w_i} = -(y - \hat{y})\frac{\partial \hat{y}}{\partial w_i}$$ Applying chain rule we get

$= (-(y - \hat{y})) \cdot f'(a) \frac{\partial}{\partial w_i} \sum w_i x_i$ as $a = w_i x_i$

$\frac{\partial}{\partial w_i} \cdot \sum w_i x_i = x_i \quad \therefore \frac{\partial E}{\partial w_i} = -(y - \hat{y}) f'(a) x_i$

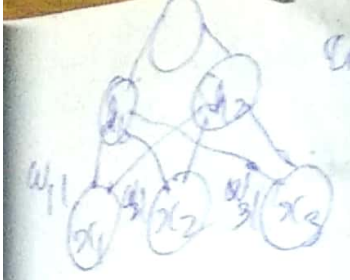
$w_i = w_i + \eta (+(y - \hat{y}) f'(a)) x_i$

as $\Delta w = -$ gradient

General algo for updating with gradient descent

- Set weights to zero.
- For each record in training data:
  - Make a forward pass through network $\hat{y} = f(\sum_i w_i x_i)$
  - Calculate error for output unit $\delta = (y - \hat{y}) * f'(\sum_i w_i x_i)$
  - Update weight step, $\Delta w_i = \Delta w_i + \delta x_i$.
- Update the weights $w_i = w_i + \eta \Delta w_i / m$ where m is no of records that enables to reduce large variations in training data.
- Repeat for e epochs.

Initializing weights - we have to initialize them to be small so they are in linear region & not squashed at high and low ends. Initialization should also be random to avoid symmetry - A good value for scale is $1/\sqrt{n}$, n - number of input units.



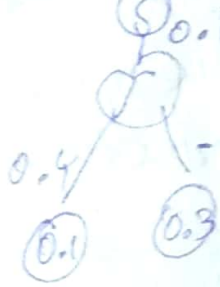
Input → hidden units

Inputs ↓

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \quad \delta_j^h = \sum w_{jk} \delta_k^o h'(q_j)$$

$\Delta w_{kj} = \eta \delta_j^h x_i$. w_{ij} weights between input and hidden layer.

$\Delta w_{kq} = \eta \delta_{output} x_{in}$



$h = \sum w_i x_i = 0.1 \times 0.4 - 0.2 \times 0.3$

$= -0.02.$

$a = f(h) = \text{Sigmoid}(-0.02)$

$= 0.496$

$y^2 = f(w.a) = \text{Sigmoid}(0.1 \times 0.496)$

$= 0.512.$

$f'(w.a) = f(w.a)(1 - f(w.a))$.

error term $\delta^o = (y - \hat{y}) f'(w.a)$

$= (1 - 0.512) \times 0.512 \times (1 - 0.512)$

$= 0.122.$

To calculate error term for hidden unit we scale error term from output unit by w.

For hidden error term, $\delta_j^h = \sum_k w_{jk} \delta_k^o f'(h_j)$

$\delta^h = w \delta^o f'(h)$. $= 0.1 \times 0.122 \times 0.496 \times (1 - 0.496)$.

$= 0.003$

Now since we have the errors we need to calculate gradient steps.

$$\Delta w = \eta \delta^o f(h) = 0.0902$$

For input to hidden units.

$$\Delta w_i = \eta \delta^h x_i$$

$$= (0.5 \times 0.003 \times 0.1, 0.5 \times 0.003 \times 0.3)$$

$$= (0.00015, 0.00045)$$

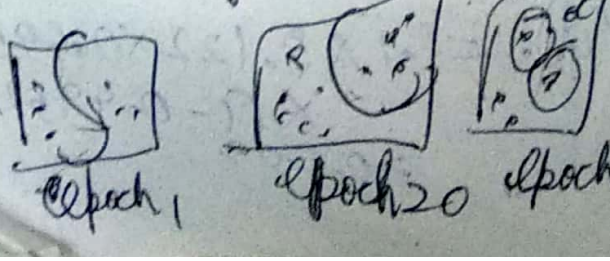
One advantage is that max derivative of sigmoid is 0.25, so errors can be reduced by atleast 75% & errors are scaled down by 93.75%.

If we have a lot of layers, using sigmoid will reduce weight steps to tiny values near the input. This is the vanishing gradient problem.

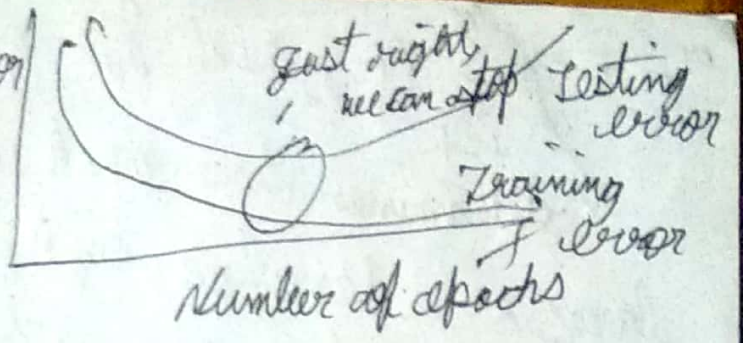
Learning optimization

Errors - Underfitting - trying a solution too simple & won't work. Overfitting - performs well on train set but not on test set. underfitting - bias, overfitting - variance

Early stopping | overfitting | classifier is too specific.



Model complexity graph



(1,1)
(-1,-1)

which equation gives smaller error?

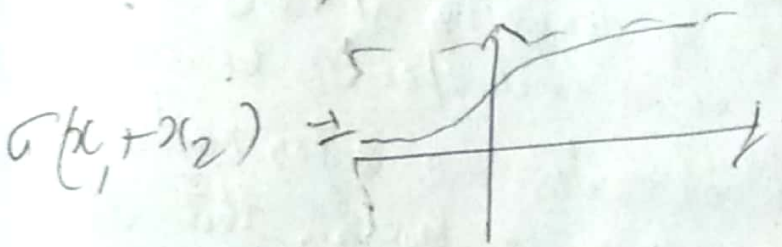
- $x_1 + x_2$
- $10x_1 + 10x_2$

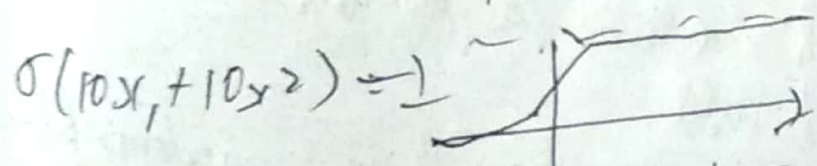
Answer is $10x_1 + 10x_2$. as $x_1 + x_2 = \sigma(1+1) = 0.8$

$\sigma(-1-1) = 0.12$

$10x_1 + 10x_2 = \sigma(10+10) = 0.99$ 4 (overall close to 1 great predictor)

$(-1,-1) \equiv \sigma(-10-10) = \sigma(-20) = 0.0000\,21.$





- function is closer to 0 & 1 but is much steeper & hard to do gradient descent. Bad models give good accuracy
- Solution to this issue is: regularization. Large coefficients → overfitting

Penalize large weights $(w_1, \ldots w_n)$.

$$E = \frac{-1}{m} \sum_{i=1}^{m} (1 - y_i) \ln(1 - \hat{y}_i) + y_i \ln(\hat{y}_i) + \lambda(|w_1| - |w_n|)$$

or $E = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)\ln(1-\hat{y}_i) + y_i \cdot \ln(\hat{y}_i) + \lambda(w_1^2 - w_n^2)$

λ determines how much we want to penalize
large λ - penalize high. First equation is
L1 reg & above one is L2.

L1

Sparsity = (1, 0, 0, 1, 0)

Good for feature selection
Makes many weights to zero

L2

Sparsity (0.5, 0.3, -0.2, 0.4, 0.1)

Here it does not aim on making the weights 0 but rather keeping the weights homogenously small.

Dropout

Train 

Present Train

Sometimes a part of the network has more weights & ends up dominating others.
So in each epoch we can randomly turn off nodes so other nodes can catch up. Each node has a prob. that it will be turned off.

Random restart: start from new random place & do gradient descent. This increases prob. that we will arrive at a global minimum or a good local minimum.

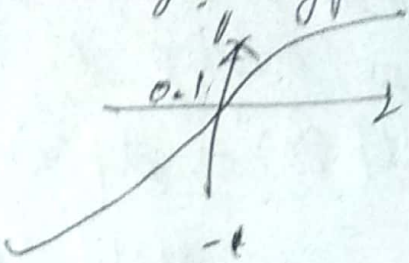
Vanishing grad: In sigmoid function gradient vanishes at the ends (becomes 0). For a MLP

$$\frac{\partial E}{\partial w_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}^1} \frac{\partial \hat{y}^1}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial w_1} \rightarrow \text{do}$$

gradients become really small. We make very tiny steps & never converge.

An alternative to avoid vanishing gradient is to use a different activation function eg:- hyperbolic tangent.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Relu  $\text{relu}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$

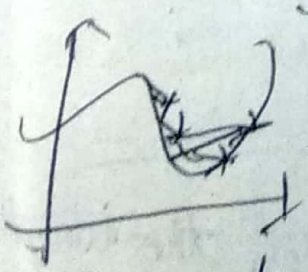
derivative is 1 if number is +ve.

Batch predictions :- In each epoch we take entire data and run it through the entire network we find the predictions & then backpropagate. If we have huge data it would take lots of matrix mul & huge memory for just 1 step. There are many such steps.

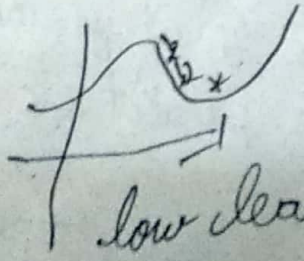
Stochastic gradient descent :- Split data into batches & run them through the network.

Learning rate decay:

high learning rate overshooting



may make model slow



low learning rate

A good model is where learning rate decreases as we approach convergence.

Rule:- 1) If steep : long steps

2) If plain : small steps.

momentum :- move with momentum to avoid getting stuck at local minimum. usually momentum could be taken as average of previous few steps.

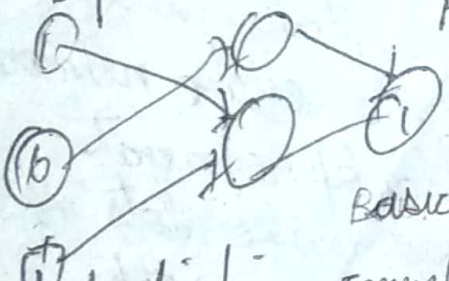
$$step(n) = step(n) + \beta\, step(n-1) + \beta^2 step(n-2) + \cdots$$

This way steps that happened long time ago may matter less than those that happened recently.

---

sentiment analysis

Input



Positive Building neural net for sentiment analysis :-

Basically sentiment classification works by finding correlation in training data. Sometimes we don't have better convergence due to noise in input data.

---

understanding neural noise.

Noise vs signal :- Each weight controls how much each input contributes the next layer. If it's say one of the words in our vocab has value 18 & it is nothing (' ') empty string or it is a period etc. These characters are noise.

Understanding inefficiencies

If a lot of inputs are 0 then the weights sum contributes nothing and computation leads to inefficiency.

Further noise reduction

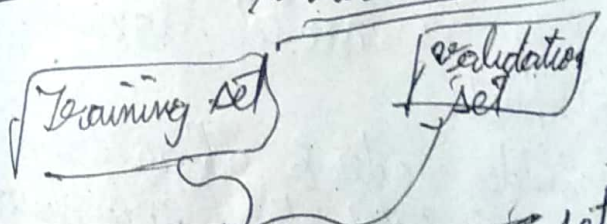
what if we can weigh postive words or negative words most common than other words. & we can remove words that do not have much predictive words like "a", "the"

CNN

Image is viewed as a 28×28 matrix, white pixels are encoded as 255, black as 0 etc. To convert our use MLP we have to convert image to vector. This is called flattening. eg:- 25×25 to 784.

Since image classification is multiclass we can use categorical cross entropy as loss function.

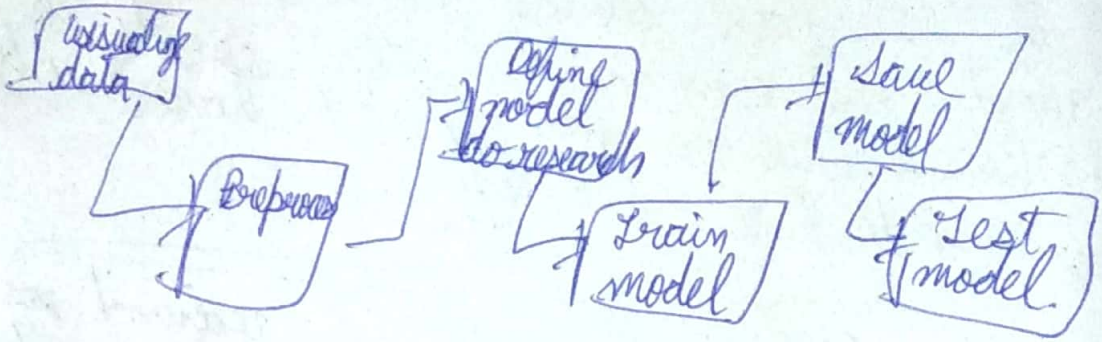
Model validation



Train on training set & calculate train loss & validation loss. Validation set is not used for backpropagation. It is only used to evaluate if model is generalizing well. Since it is not used to update weights it helps us identify if model is overfitting.

We also need a test set cause model selection is based on train and validation set. We need a test set so that we can truly see if model is generalized & not biased towards validation set.

Image classification steps



MLP vs CNN

Unlike MLPs, CNNs know that pixels in close proximity are heavily related than those far apart.

Local connectivity

- MLP - lot of parameters as only fully connected layers. * When we flatten the image to a vector for MLP we lose all 2-D information, knowledge of how pixels are located relative to each other.

CNN - • Uses sparsely connected layers.
• Accept matrices as input.

Instead of having more hidden nodes & all of them connected to input layer as in CNN there are lesser hidden nodes with each node only focusing on a smaller region of image. Thus we have fewer parameters. This will enable each hidden node share a common group of weights.

CNN's preserve spatial information. They process image as a whole or in blocks rather than input one by one as in MLP.

Spatial patterns - shape & color.
shape - intensity - (black or white).

edges can be detected by abrupt changes in intensity.

high pass filters ÷ high frequency edges

images are where intensity rapidly changes when moving from one pixel to others. high pass filters enhance such parts of image (emphasizes edges).

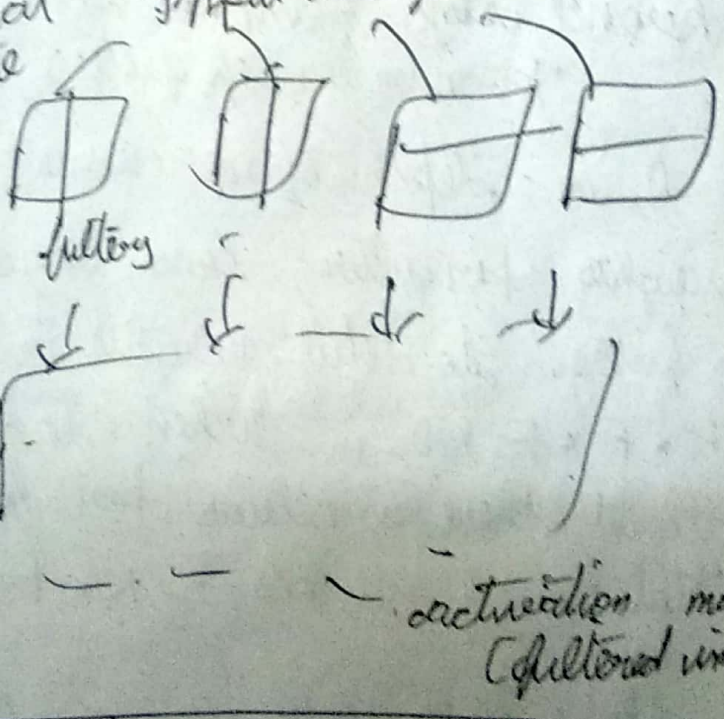
edge detection filter

0	-1	0
-1	4	-1
0	-1	0

It is important that values sum to 0. centre pixel is more important. Negative weights (-1) increase the contrast in the ~~image~~.

convolution layer

filters are initialized at random & so are the patterns detected. we define a loss function like categorical cross entropy (for multi class) & update the filters to learn the patterns needed.



stride & padding

stride - by how much the filter moves over the image.

Increasing depth

Resizing images is needed as CNN's need same size input. We resize such that spatial dimensions equal a power of 2 or divisible by a small power of 2.

Conv layers help increasing depth of the image & max pooling decreases x & y co-ordinates. Increasing depth helps extract useful features.

Number of parameters in a conv layer: It depends on filters, out channels, kernel size & input shape:

K - number of filters, F - height & width, D_in - depth of previous layer. F*F*D_in weights per filter, conv layer is composed of K filters. So total weights in conv layer is K*F*F*D_in. When taking bias into account, there is 1 bias per filter which means total no. of weights = K*F*F*D_in + K.

shape of conv layer: $\frac{W\_in - F + 2P}{S} + 1$

Image augmentation:

Scale invariant - We don't want pred. to depend on size of object.

Rotation invariance - not depend on angle of object.
Translation invariance - not depend on position of object.

Imagenet - 10 million hand labelled images drawn from 1,000 image categories.
Alexnet pioneered Relu & dropout to avoid overfitting.
VGG - 16 & 19 - 16 & 19 layers. - 3x3 conv, 2x2 pooling & 3 FC layers.
Alexnet - 11x11 windows.
Resnet - similar to VGG but latest version has 152 layers. More layers - vanishing gradient problem. Skip layers were added which enabled gradients to take a shorter route & avoid vanishing gradient problem.

Transfer learning

VGG-net - we can take an existing model & apply it to new dataset. If data set is small & has distinct shapes & features similar to Imagenet we can leverage most layers of VGG & just use final trainable layers.

Guide to use transfer learning

Size of data	Similar	Different
Large	Fine-tune	Fine tune or retrain
small	End of convnet	Startup convnet

weight initialization.

When we use transfer learning we use best pretrained weights for image classification problems. But for non-pretrained we need to figure out way to initialize weights.

All weights 0 initialization would have negative effect as it is impossible to train the weights or nodes are cause of error. This is because the gradients are all the same & it is difficult to figure out how much the weight should be updated.

If weights are all 1's same thing happens. But 1's are more harmful as they are a very large value & it is difficult for backprop to update & bring the weights down.

All this can be avoided by proper weight initialization by initializing weights to distinct values. This can be done by picking a random value from uniform distribution.

General rule: But initializing weights to just any value like 0 & 1 is not efficient. Rather since weights multiply with inputs, they should have some relation with weights i.e. inversely proportional. More # of numbers of units mean that the weights should be smaller.

A general rule for setting weights in network is to set them to be close to zero without being too small - i.e. weights should be in range $[-y, y]$ where $y = 1/\sqrt{n}$ (n - no. of inputs)

Autoencoders

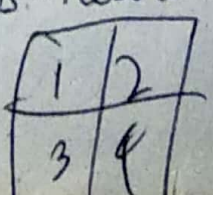
i.e N through its various layers discards spatial information & isolates high level info about the content of the image. This can be viewed as data compression (to feature map).

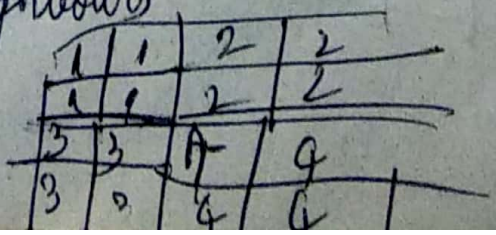
Now autoencoder has 2 parts: encoder & decoder. Autoencoder reduces the dimensionality of any input. This compressed format can be used for various purposes that are faster than original image. Autoencoders help in denoising image.

Learnable Upsampling

Conv layers preserve spatial information. A series of conv & max pooling layers downsample the spatial dimensions. This constitutes the encoder.

This can be reversed by reversing downsampling this can be used as decoder that gives the original image. We can use unpooling to get original vectors. This can be done by linear interpolation that is nearest neighbours





Linear interpolation is a crude idea as we are just copying numbers. A more realistic method a have variety and learnable parameters.

A better way to upsample the image is trans convolutions. This method does not use predefine convolutions but rather has learnable parame They are also called deconvolution layers Bu they do not undo convolutions but rather ups the input using learnable filter weights.

Transpose convolution.

Input 4x4 → 3x3 kernel padding 1 stride 2 → 2x2 output

stride is applied on inp to 0/1 dimensi

Now for transpose convolution

- we take one pixel from this 8 apply a 3x3 filter to it which in turn gi a 3x3 output.

Now for next pixel we take center & since stride is

in case of overlap values would be summed together.

- result is a 5x5

Upsampling & denoising

Sometimes blocky checkerboard patterns appear in images. They maybe due to overlap. We can apply denoising to remove such patterns. Eg. Nearest neighbours could be used to interpolate & avoid checkerboard pattern.

Style transfer

: A trained CNN has already learnt to represent content in later layers. But what about style. In case of a painting style would be brush strokes, etc.

VGG19

First the content image is passed through VGG19 & it goes through feedforward till it reaches the deepest convolutional layer. This layer gives the content representation.

When the style image is passed through VGG19 different layers extract different styles. The content & style image can be combined to give the output image.

A content loss is defined to compare the content extracted from conv 4-2 with original content image.

content image - C_0 Target image content - T_0

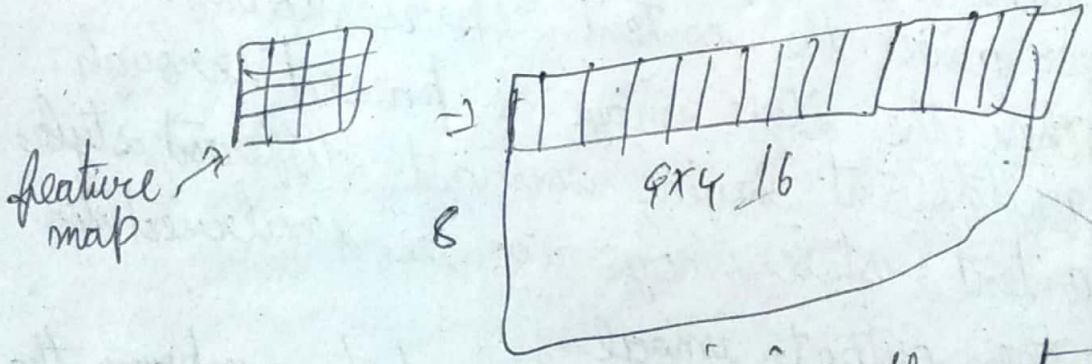
$$L_{content} = \frac{1}{2} \sum (T_0 - C_0)^2$$

Here we try to optimize but not by training but changing target image to match source image (content).

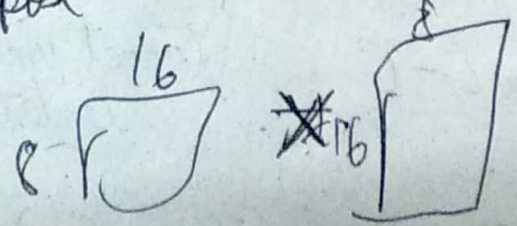
We want to do the same thing for styles i.e. compare the target image style representation style image. This is done by looking at correlation between features in individual layers. This includes calculating how similar features in single layer are. This gives a multi scale style representation correlations at each layer are given by gram matrix.

4 x 4 → 4 x 4 x 8 – it has 8 feature maps
input convolutional layer that we want to find correlations between.

First step involves vectorizing the layer.



Next step involves multiplying the outcome by its transpose.



→ 8 x 8

result contains localized info which means result won't vary if images were shuffled in space.

To compute style loss we need to find MSE between the gram matrices from style image & from target image.

$$L_{style} = \alpha \sum_i w_i \cdot (T_{s,i} - S_{s,i})^2$$

w_i — weights. — only that changes them.

total loss = $L_{content} + L_{style}$ - minimize them.

---

we need to add constant weights α & β as content and style loss are calculated differently and we need to weigh them differently.

total loss = $\alpha L_{content} + \beta L_{style}$.

$\frac{\alpha}{\beta}$ - lesser α, β ratio more stylized the image.

it is more as style is more important.

$L_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum (F_{ij}^l - P_{ij}^l)^2$ F - activation of i^{th} filter at position j in layer l.

derivative of loss w.r.t activations in layer l

$$\frac{\partial L_{content}}{\partial F_{ij}^l} = \begin{cases} (F^l - P^l)_{ij} & \text{if } F_{ij}^l > 0 \\ 0 & \text{if } F_{ij}^l < 0 \end{cases}$$

Then gradient w.r.t image $\vec{x}$ can be computed using back prop & $\vec{x}$ is changed till it reach

same response as a certain layer in the CNN, the original image $\vec{p}$.

Style :- $\vec{a}$ - original image. $\vec{x}$ - image that is generated. A^l & G^l - style representations, layer l. $E_l = \frac{1}{4 N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$

N_l - number of feature maps. M_l - height & width of feature maps.

Total style loss is $L_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$ - w_l weights of contribution of each layer to total loss.

$$\frac{\partial E_l}{\partial F_{ij}^l} = \begin{cases} \frac{1}{N_l^2 M_l^2} ((F^l)^T (G^l - A^l))_{ji} & \text{if } F_{ij}^l > 0 \\ 0 & \text{if } F_{ij}^l < 0. \end{cases}$$

Skin cancer detection :-

Precision is fraction of relevant instances among those retrieved & recall is the fraction of total number of relevant instances that were retrieved. For example is a image of 12 dogs if 8 were identified as dogs but ground truth is only 5 dogs, then precision = 5/8, recall = 5/12.

Sensitivity : Of all sick people how many did we diagnose as sick.

Specificity : Of all healthy how many we diagnosed as healthy.

	Diagnosed sick	Diagnosed healthy
sick	True positive	False negative
healthy	False positive	True Negative

TP: correctly diagnosed as sick.

TN: Healthy people diagnosed as healthy.

FP: Healthy people diagnosed as sick.

FN: Sick people incorrectly classified as healthy.

$$\text{Sensitivity} = \frac{TP}{TP+FN}$$

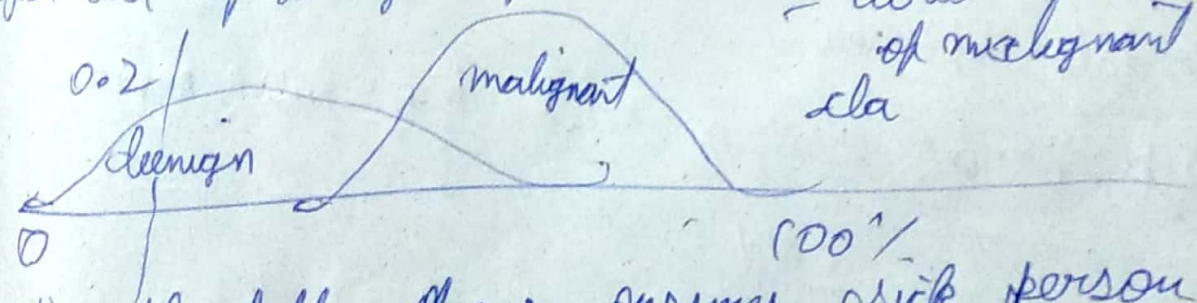
$$\text{Specificity} = \frac{TN}{TN+FP}$$

$$\text{Recall} = \frac{TP}{TP+FN}$$

$$\text{Precision} = \frac{TP}{TP+FP}$$

Threshold for detecting melanomas
0.2, 0.8 or 0.5.

Below is histogram of predictions model gives for set of images of lesions.



- covers most of malignant cla

No threshold of 0.2 ensures sick person gets treatment.

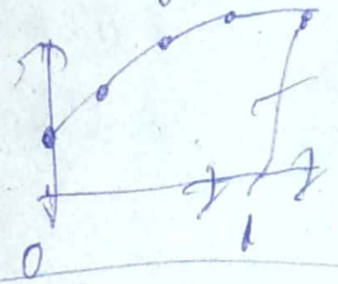
ROC curve

True positive rate = $\frac{\text{True P}}{\text{All positives}}$

Perfect split - 1
Good split - 0.8
random split - 0.5 or 0.2

F1 rate = $\frac{FP}{\text{All negatives}}$

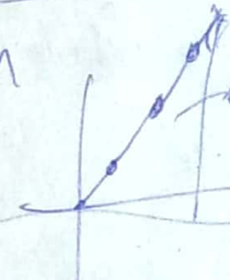
Plot diff points i.e. true positive rate in x axis & FP rate in y axis.



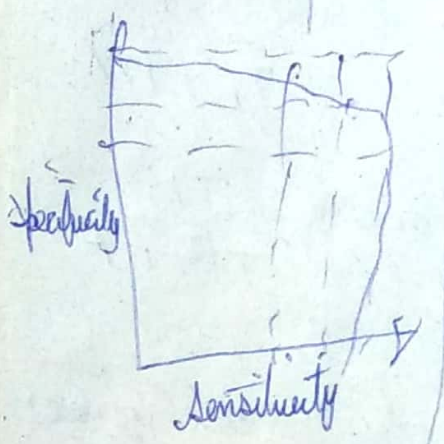
area under ROC curve - 1 for perfect split.

Good split - 0.8 AUC

Random split



area under curve = 0.5



Confusion matrices

	Diagnosed sick	Diagnosed healthy
Sick		
Healthy		

	Diagnosed sick	Diagnosed healthy
Sick	TP	FN
Healthy	FP	TN

Type 1 error: error of first kind, or false positive.

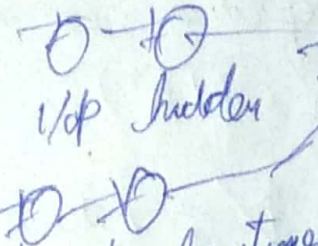
Type 2 error: error of 2nd kind, or false negative.

RNN

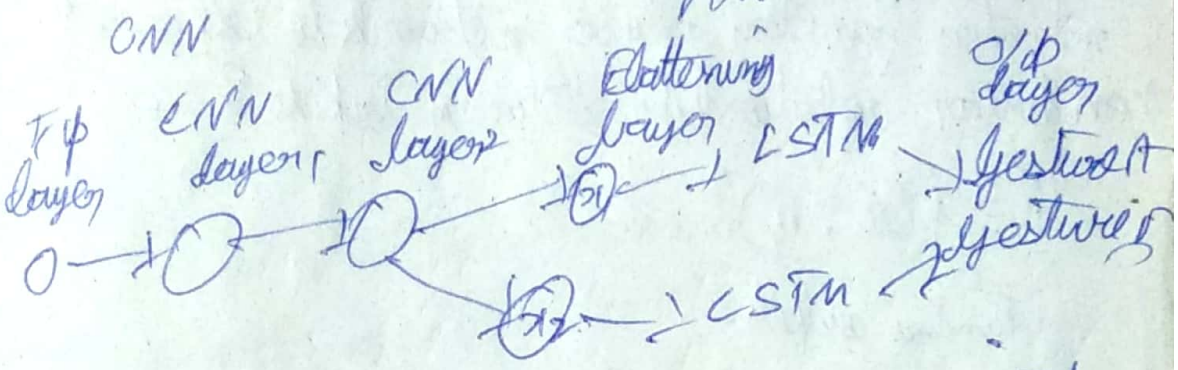
RNN capture temporal dependencies (dependencies over time). RNN - are called recurrent as we perform same task for each element in input sequence.

Feedforward neural networks were limited in that they were not able to model temporal endencies. So recurrence was added.

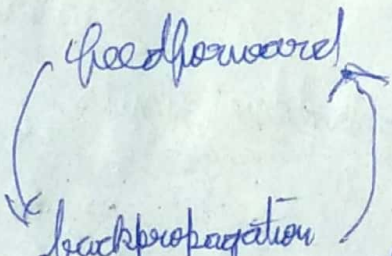
FFNN - quick reminder



CNN's and RNN's can be combined CNN's can be used to extract features & RNN for memory.



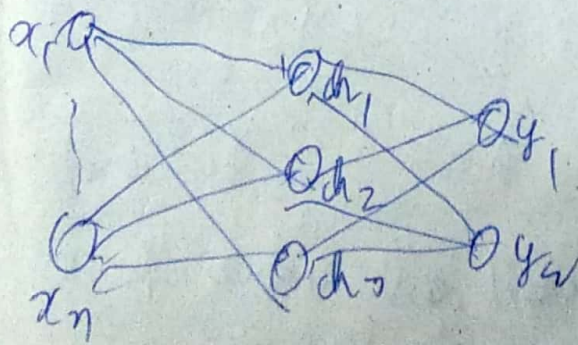
Neural networks are non linear function approximators. FFNN are static mapping as there is no memory & output just depends on inputs & weights, which means given same inputs & weights same output occurs.



This process is repeated as many times as needed.

Notation w_k - weight matrix, w^k_{ij} - ij^{th} element of weight matrix k.

Feedforward



- step 1 - finding h
- step 2 - finding y

w_{11} - connects x_1 & h_1,

$$[h'_1\ h'_2\ h'_3] = [x_1 \cdots x_n] \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & & \\ w_{n1} & w_{n2} & w_{n3} \end{bmatrix}$$

$h' = x \cdot w_1 \quad h_1' = x_1 \cdot w_{11} \quad \dots x_n \cdot w_n)$

The present h value from exploding we need to use an activation function. $h = \varphi(h')$.

tanh h — relu.

Activation function allows network to represent non linear relationships between inputs & o/p.

$h = \varphi(x \cdot w_1)$

Finding outputs :-

$[y_1 \; y_2] = [h_1 \; h_2 \; h_3] \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$

2 columns as only 2 o/p's so 2 rows as 3 hidden units. $y = h w_{ij}$

$h_1 = \varphi\left(\sum_i^d x_i w_{i1}\right)$

$h_2 = \varphi\left(\sum_i^n x_i w_{i2}\right)$

$h_n = \varphi\left(\sum_i^n x_i w_{in}\right)$

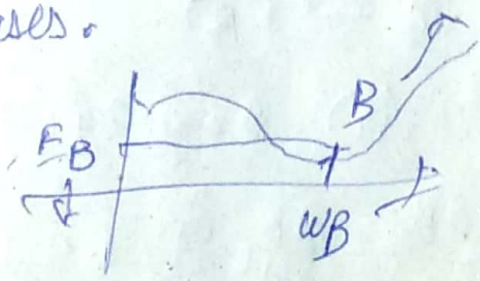
softmax is used when we want all values of o/p to be between 0 & 1 & their sum to be 1.

Backpropagation - SGD with chain rule.

Find set of weights minimizing network error.

We need to increase value of w_A to decrease

If gradient is negative, we need to change weight in that direction so that it increases.



To decrease error we need to decrease weight w_B.

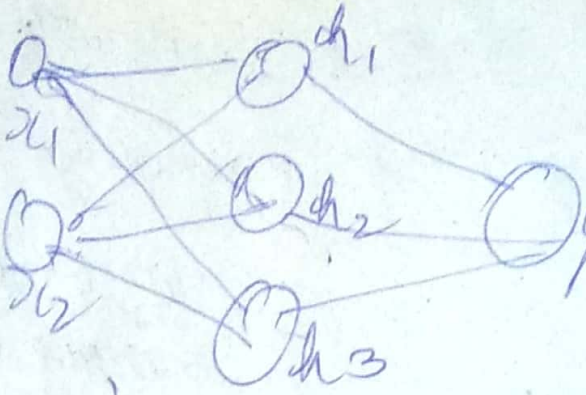
The gradient at B is increasing (positive). If we take a step in negative direction of gradient, then weight would be correctly decreased.

$$w = w_{prev} + \alpha\left(\frac{-\partial E}{\partial w}\right)$$

$$w_{new} = w_{prev} + \nabla w(-E)$$

(gradient - vector of partial derivatives of error w.r.t each weight.

$$\Delta w_{ij}^{k} = -\alpha \frac{\partial E}{\partial w_{ij}^{k}}$$



$h_1 = \varphi\left(\sum_i^2 x_i w_{i1}^1\right)$

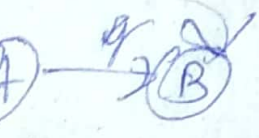
$h_2 = \varphi\left(\sum_i^2 x_i w_{i2}^1\right)$

$h_3 = \varphi\left(\sum_i^2 x_i w_{i3}^1\right)$

w_{ij}^1 $\quad$ w_{i}^2

$h = [h_1, h_2, h_3]$ $\quad$ $y = \sum_i h_i \cdot w_i^2$

$E = \frac{(d-y)^2}{2}$



$A = f(x)$ $\quad$ $B = g \circ f(x)$

composing functions is same as multiplying partial derivatives.

$\frac{\partial B}{\partial x} = \frac{\partial B}{\partial A} \frac{\partial A}{\partial x}$

$\frac{\partial y}{\partial w_i^2}$ ← output to hidden

$y = \sum_i^3 h_i w_i^2$

$\frac{\partial y}{\partial w_i^2} = \frac{\partial \left(\sum_i^3 h_i w_i^2\right)}{\partial w_i^2} = h_i$

$\frac{\partial y}{\partial w_{ij}^1} = \frac{\partial y}{\partial h_\phi} \frac{\partial h_\phi}{\partial w_{ij}^1}$ $\quad$ $\frac{\partial h_j}{\partial w_{ij}^1} = \partial \varphi_j\left(\sum_{i=1}^2 x_i w_{ij}^1\right)$

$\frac{\partial^2 \sum_i (x_i w_{ij}^1)}{\partial w_{ij}^1}$

$+ \frac{\partial \sum_{i=1}^2 (x_i w_{ij}^1)}{\partial w_{ij}^1}$

$= \varphi_0' x_i$

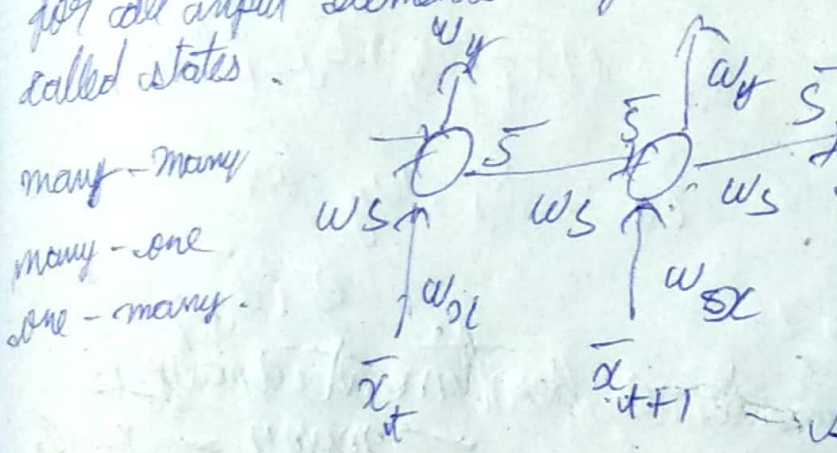
$$\frac{\partial y}{\partial w_{ij}} = w_{jk}^{2} \varphi_j' \cdot x_i$$

mini batch gradient descent - updating weights every N steps.

$$\delta = \frac{1}{N} \sum^{N} \varepsilon_{ijk}$$

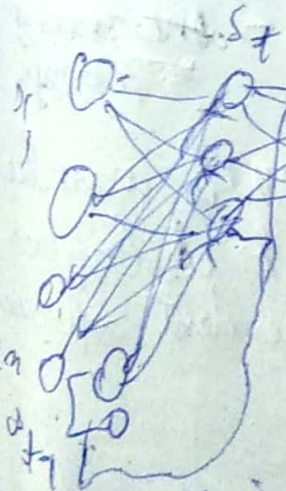
RNN - Recurrent neural networks.

They are called RNN as we perform same task for all input elements. They include memory elements called states.



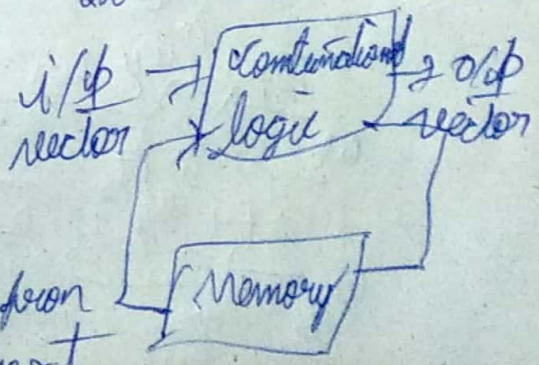
many applications have temporal dependencies & hence require past input, sequences.

This is similar to state machine



Elmann networks

- we can see this as hidden layer o/p from previous time step is given as i/p in next time step.

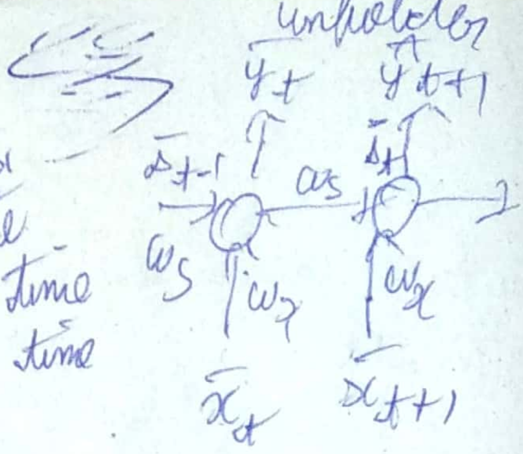


In FFNN inputs are considered to be independent.

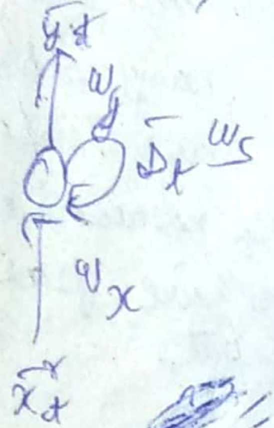
But in RNN $\bar{q}_t = F(\bar{x}_t, \bar{x}_{t-1}$ —

$- x(t-1, w)$

Bolded – represents weight matrix connecting state from previous time step to next time step.



$\bar{x}_t$ – i/p vector at time t, $\bar{y}_t$ – output vector at time t, $\bar{s}_t$ – hidden state vector at time t

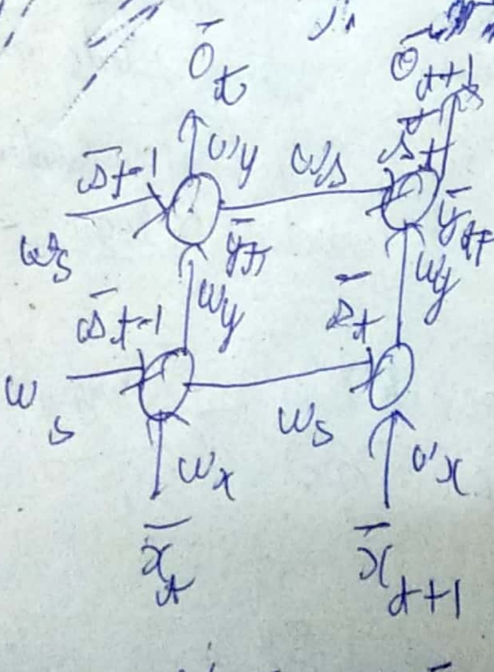


$$\bar{s}_t = \phi(\bar{x}_t \cdot W_x + \bar{s}_{t-1} \cdot W_s)$$

$$\bar{y}_t = \bar{s}_t \cdot W_y \text{ or } \sigma(\bar{s}_t \cdot W_y)$$

~~Sentime~~ Sentiment analysis – many – one

~~time~~ time series – ~~one~~ many – many.



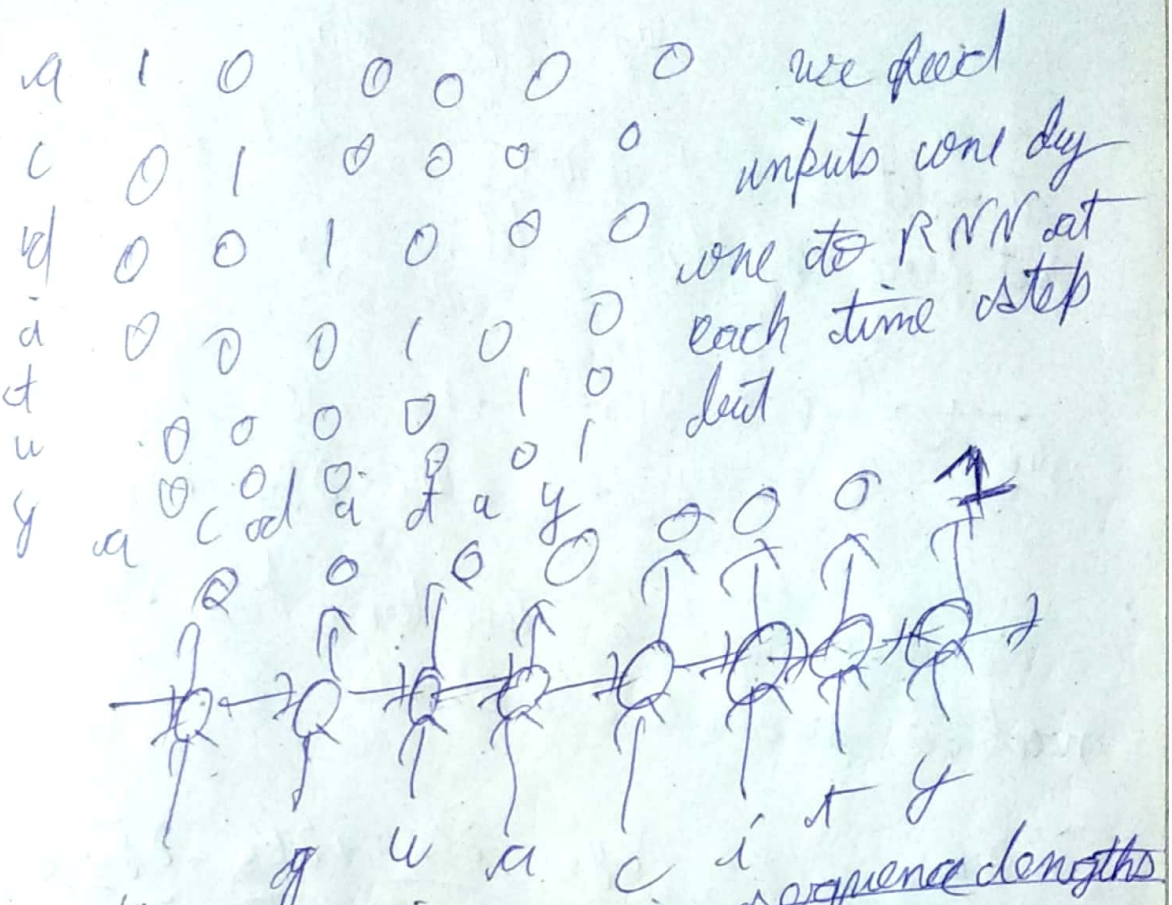
So RNN's can be stacked & layers are independent of each other.

~~RNN~~ FFNN $\bar{y}_t = F(\bar{x}_t, W)$

RNN, o/p at time t $\bar{y}_t = F(\bar{x}_t, \bar{x}_{t-1}, \bar{x}_{t-N})$

RNN - example - Sequence detection - word detection

Here we want to detect word audacity.



we feed inputs one by one to RNN at each time step and

RNN can deal with varying sequence lengths

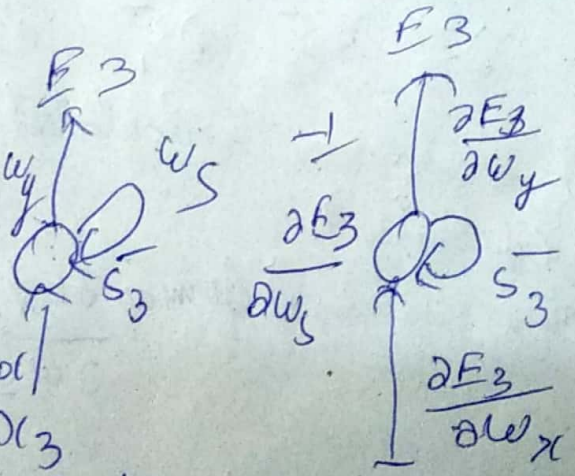
Backprop through time

Timestep 3

$E_3 = (d_3 - y_3)^2$

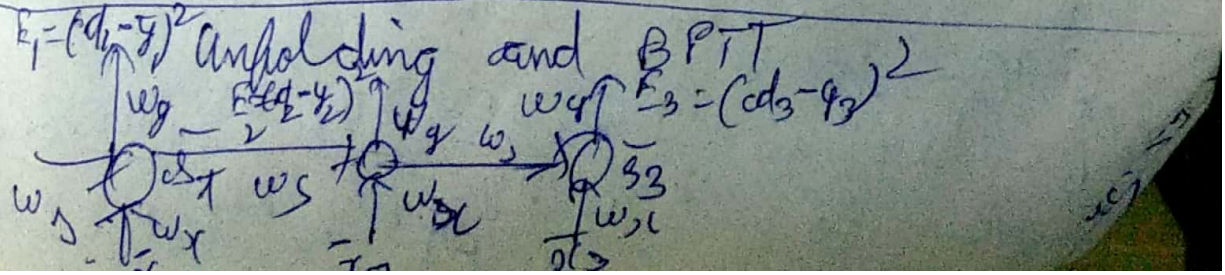
d - desired output

y - calculated o/p



We also need to consider previous timestep - s_2.

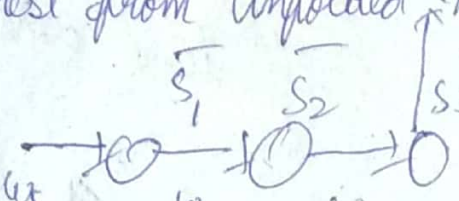
$E_1 = (d_1 - y_1)^2$ Unfolding and BPTT $E_3 = (d_3 - y_3)^2$



3 weight matrices to modify w_x, w_y, w_s.

At time $t=3$, $\frac{\partial E_3}{\partial w_y} = \frac{\partial E}{\partial y_3} \cdot \frac{\partial y_3}{\partial w_y}$

Now let us find updates needed for w_s, same rest from unfolded model.



$$\frac{\partial E_3}{\partial w_s} = \frac{\partial E_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial s_3} \cdot \frac{\partial s_3}{\partial w_s}$$

We also need to consider s_2 & s_1. So we need to accumulate gradients.

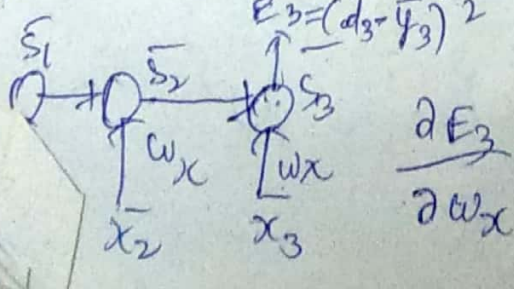
$$\therefore \frac{\partial E_3}{\partial w_s} = \frac{\partial E}{\partial y_3} \cdot \frac{\partial y_3}{\partial s_3} \cdot \frac{\partial s_3}{\partial w_s} + \frac{\partial E_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial s_3} \cdot \frac{\partial s_3}{\partial s_2} \cdot \frac{\partial s_2}{\partial w_s} +$$

$$\frac{\partial E_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial s_3} \cdot \frac{\partial s_3}{\partial s_2} \cdot \frac{\partial s_2}{\partial s_1} \cdot \frac{\partial s_1}{\partial w_s}$$

Accumulative gradients calculated considering all time steps (previous)

A more generalized form as $\frac{\partial E_N}{\partial w_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial y_N} \frac{\partial y_N}{\partial s_i} \frac{\partial s_i}{\partial w_s}$

Now we need to adjust w_x,



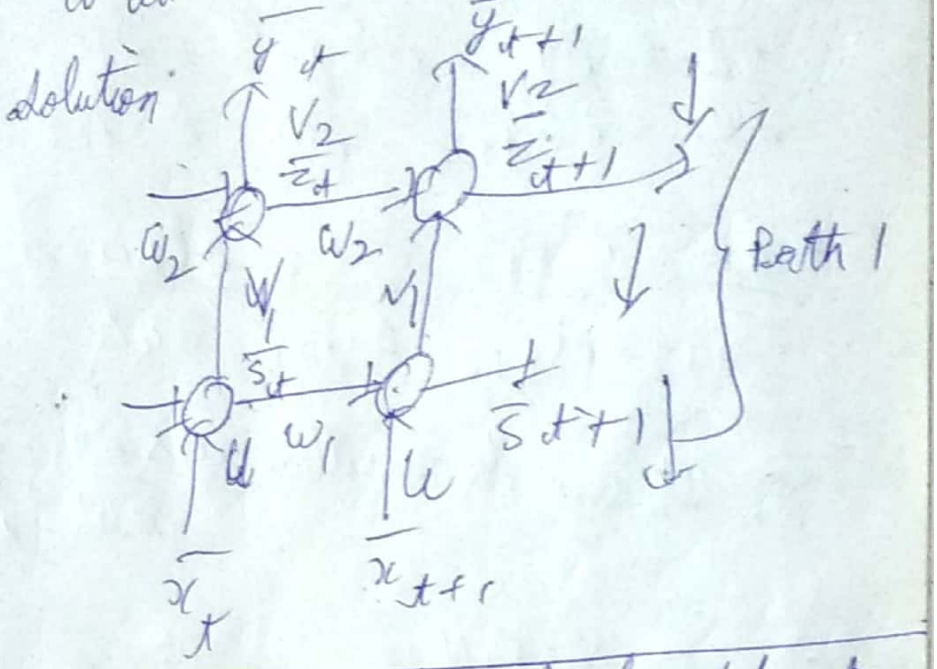
at time $t=3$

$$\frac{\partial E_3}{\partial w_x} = \frac{\partial E_3}{\partial y_3} \frac{\partial y_3}{\partial s_3} \frac{\partial s_3}{\partial w_x} + \frac{\partial E_3}{\partial y_3} \frac{\partial y_3}{\partial s_3} \frac{\partial s_3}{\partial}$$

$$\frac{\partial E_N}{\partial w_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial w_x}$$

what is update rule of matrix w at time $t+1$ over 2 time steps

solution:

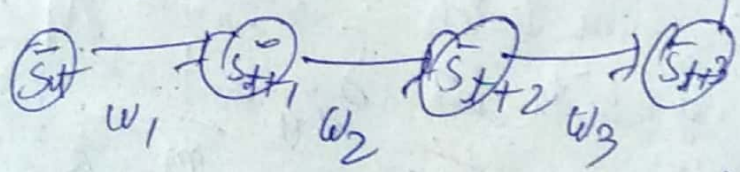


More math: generalization of deadpool three time. Generalized form

$$\frac{\partial E_N}{\partial w_s} = \sum_{i=1}^{r} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial s_i} \frac{\partial \bar{s}_i}{\partial w_s}$$

$$\frac{\partial E_N}{\partial w_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \hat{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial w_x}$$

To generalize case let us consider



$$\frac{\partial y}{\partial w_3} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial w_3} \qquad \frac{\partial y}{\partial w_2} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial w_2}$$

$$\frac{\partial s_2}{\partial w_x} + \frac{\partial E_3}{\partial y_3} \frac{\partial y_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial \bar{s}_1}{\partial w_x}$$

$$\frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_1}$$

In BPTT we accumulate gradients - i.e.

$$\frac{\partial y}{\partial w} = \frac{\partial y}{\partial w_1} + \frac{\partial y}{\partial w_2} + \frac{\partial y}{\partial w_3}$$ weight matrices are actually identical, $w_1 = w_2 = w_3$ etc.

$$\therefore \frac{\partial y}{\partial w} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial w} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial w}$$

$$+ \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial w}.$$

∴ It can be written as $\frac{\partial y}{\partial w} = \sum_{i=t+1}^{t+3} \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial w}$

$$\therefore \frac{\partial y}{\partial w} = \sum_{i=t+1}^{t+N} \frac{\partial y}{\partial \bar{s}_{t+N}} \frac{\partial \bar{s}_{t+N}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial w}.$$

(Example for $i = t+1$, we derive $\frac{\partial y}{\partial w} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial w}$.

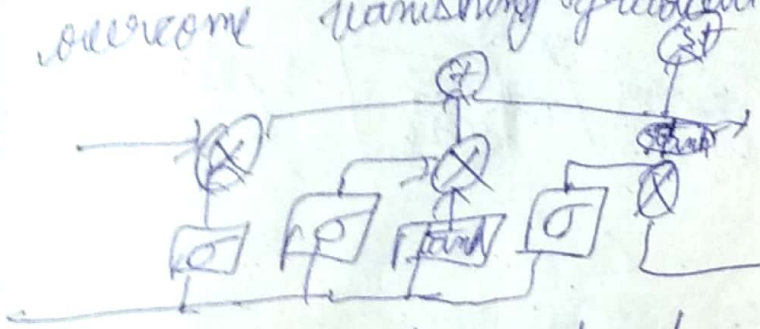
We can use mini batch gradient calculated once every single sample

we calculate gradients that update only after a certain number of steps. $\delta_{ij} = \frac{1}{m} \sum_k \delta_{ijk}$ (m = number of gradients accumulated)

For longer number of timesteps when we accumulate gradients it will lead to vanishing gradient issue where gradients become very

small - Another issue is exploding gradients where gradients become too large. We can perform gradient clipping where at each timestep we check if gradient is > threshold if yes we normalize the gradient.

- LSTM were proposed to overcome vanishing gradient problem.



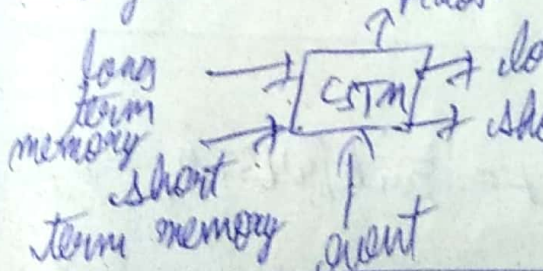
It solves vanishing gradient problem.

- Inputs can be stored without being forgotten. The LSTM cell is fully differentiable

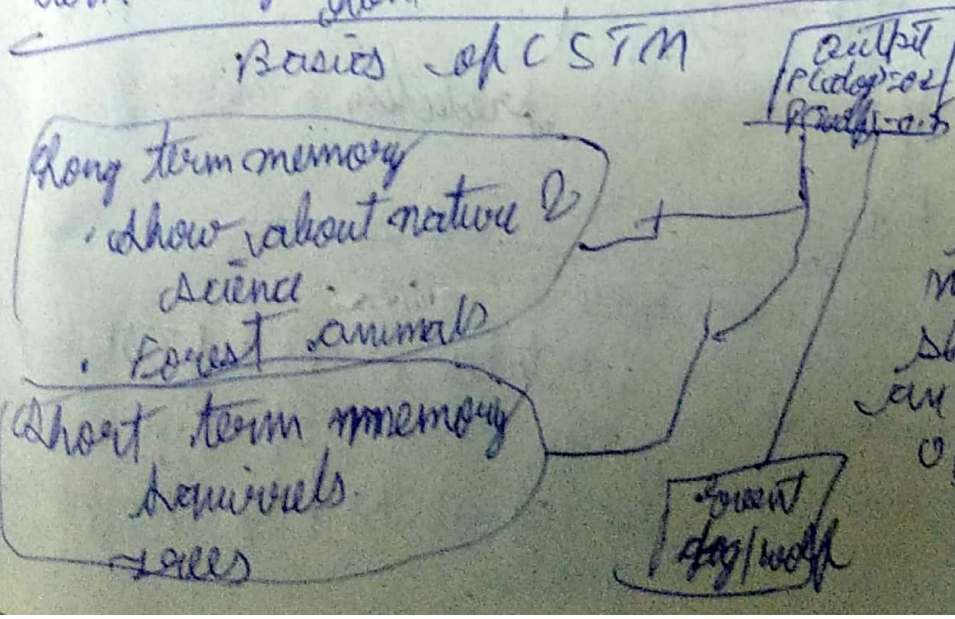
---

LSTM

RNN suffers from vanishing gradients problem. This is because RNN's only have short term memory. LSTM have long term memory.



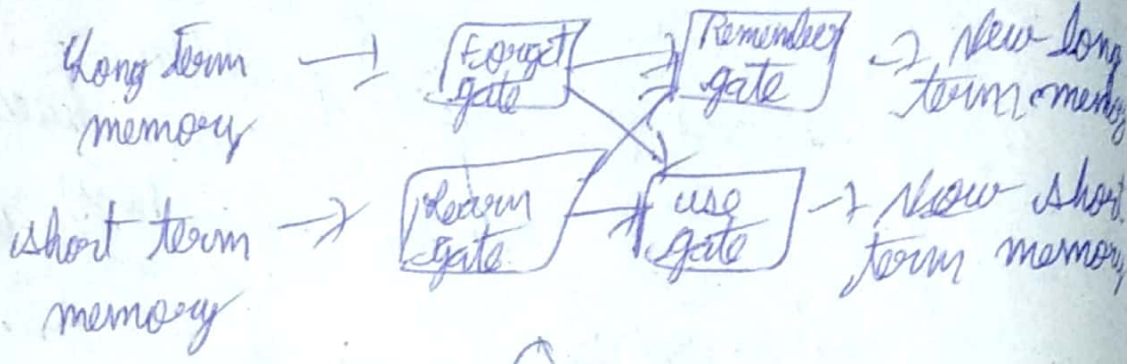
Basics of LSTM

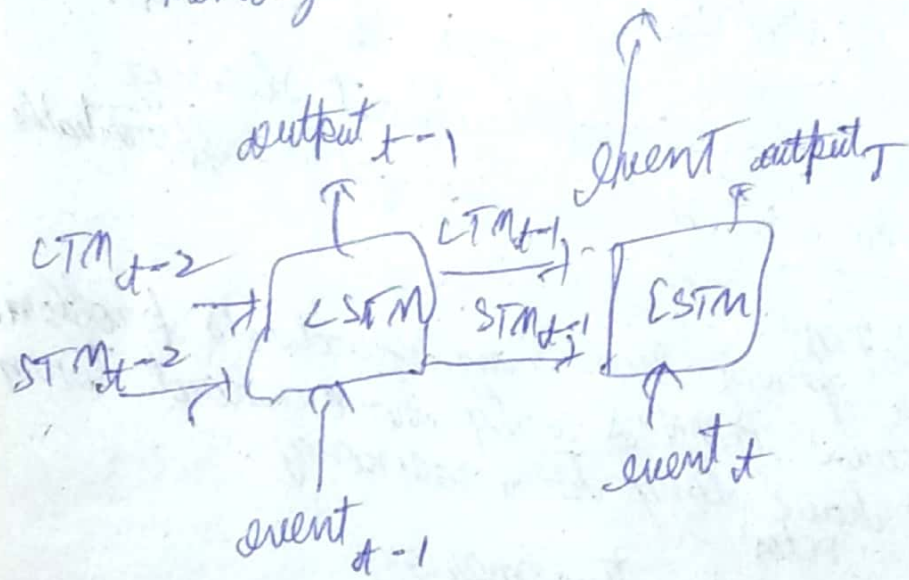


The long term memory should give an idea that o/p is wolf & not dog

Long term memory can be used to update short memory.

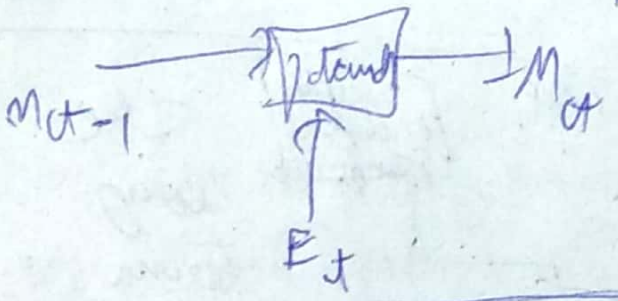
LSTM has 4 gates forget gate, learn gate remember gate & use gate.





Recap of RNN

$$M_t = \tanh[W[STM_{t-1}, E_t] + b]$$



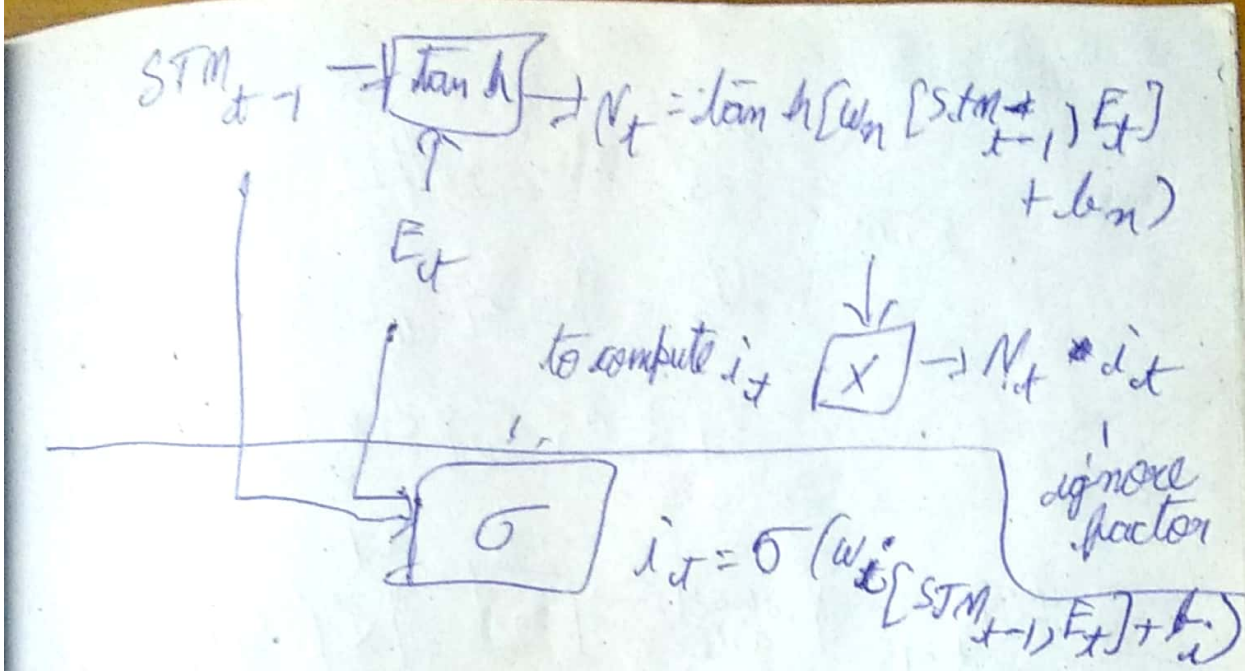
prediction + memory.

Learn gate

STM → Learn gate

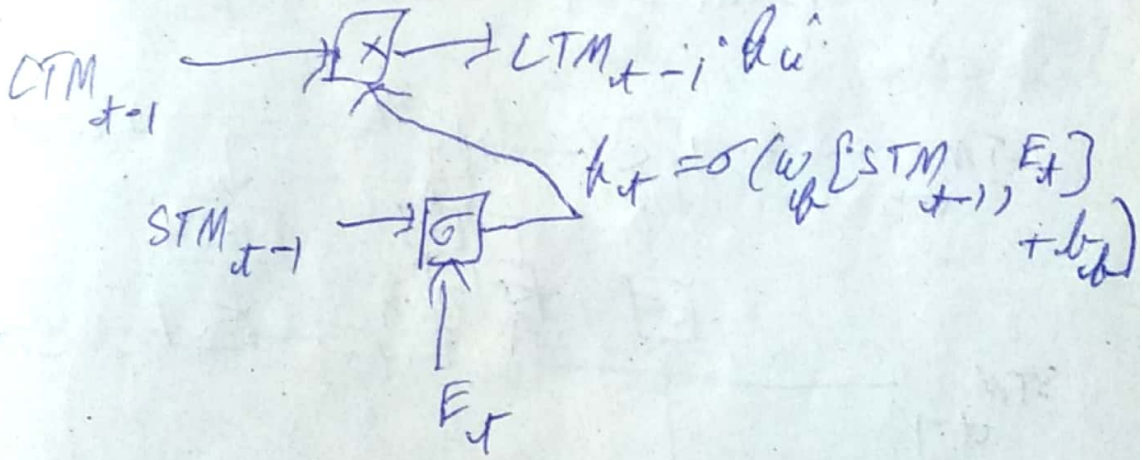
event →

It takes event & STM and joins them. It ignores a bit of it keeping important parts.

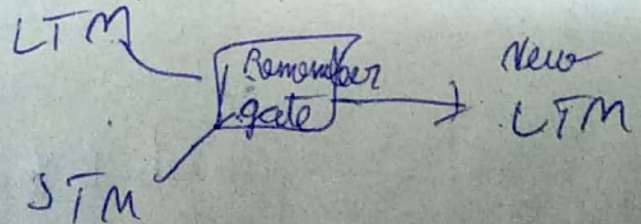


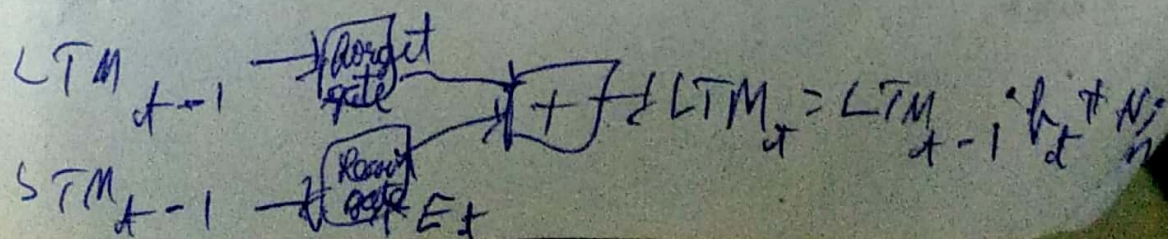
Forget gate

Takes long term memory as input & decides what to keep & forget.

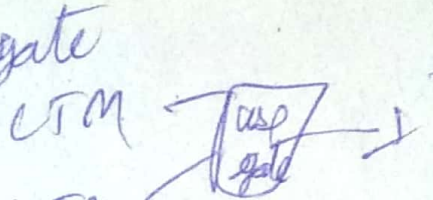


Remember gate combines LTM from forget gate & STM from learn gate & computes new long term memory



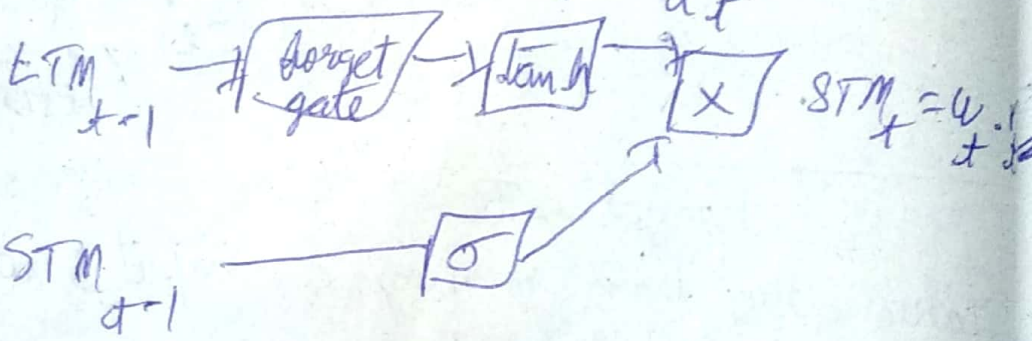


use gate

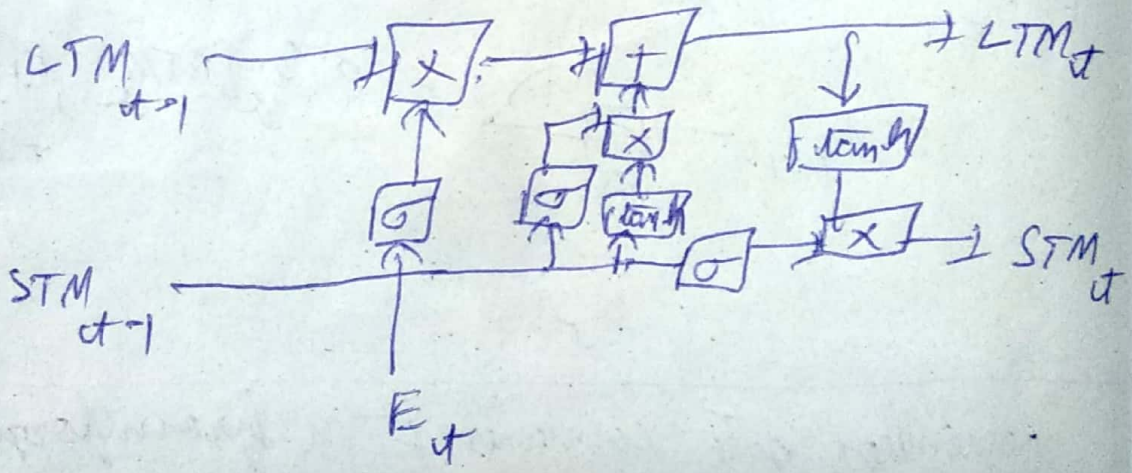


$$ud_t = \tanh(w_{ud}[LTM_{t-1}, h_t] + b_{ud})$$

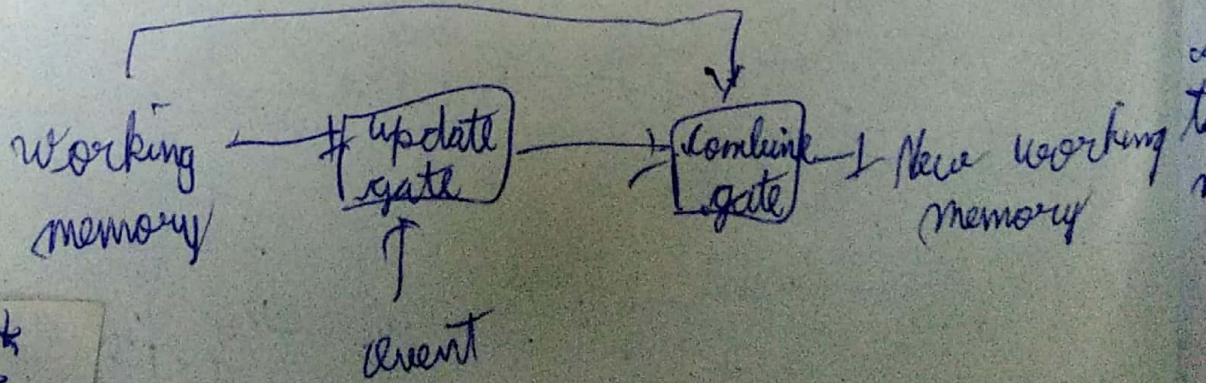
$$V_t = \sigma(w_V[STM_{t-1}, E_t] + b_V)$$



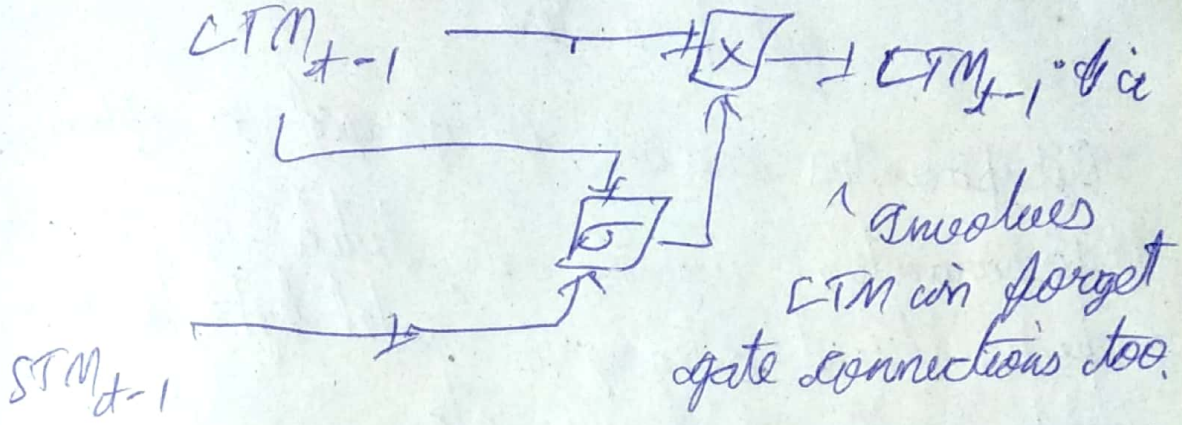
Final LSTM architecture



ORW



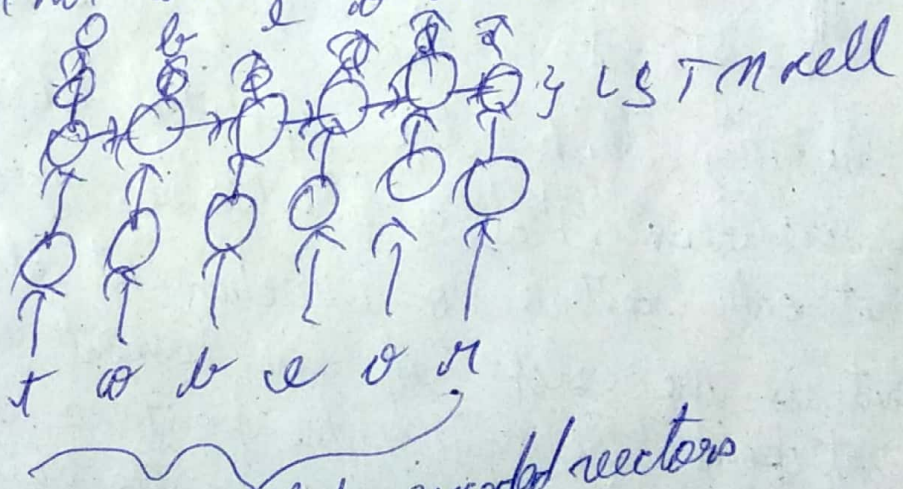
Peephole connections



Character wise RNN

- Predict next character in sequence

To be or not to be



Sequence batching :- RNN use train sequence data. Instead of whole sequence if we split & take smaller sequences, matrix operations would be more efficient eg :- [1 2 3 4 5 6]

We can split it into 2 sequences.

[1 2 3] [4 5 6]

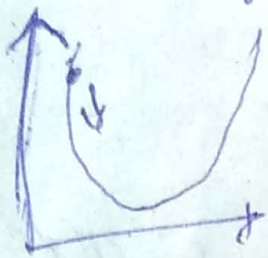
The batch size = no. of sequences.

Hyperparameters

Hyperparameters can be categorized as optimizer hyperparameters - learning rate, batch size, number of epochs. Second category is model hyperparams. It includes number of layers etc.

Learning rate

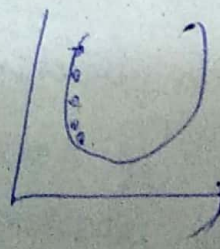
Good starting point - 0.01.



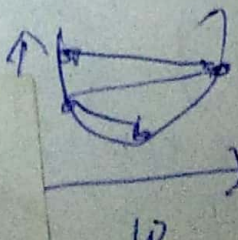
Learning rate is the multiplier used to push weight in right direction.

If learning rate is chosen to be very high the updated value will overshoot more than ideal weight value. Gradient does not only contribute to direction but also to value. that is the slope of line tangent to curve at that point. The higher the point is on curve more steep the slope is & larger the value of grad.

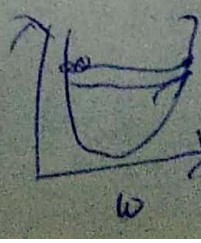
Validation err: Decreasing
learning rate: good



V error:
too slo
α - should
be increa



V error;
decrease
α - good

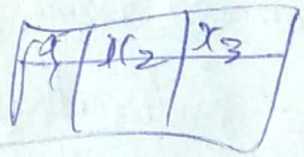


α - dec
error
- unnec

Learning rate decay - Reduce learn rate during training for convergence.

Minibatch size

It has effect on computing resources, training speed.

online (stochastic)  - one example is fed & with that forward pass, error calc & backprop is done.

Batch - entire training set, minibatch - 1 - (online), same size as input (batch). For minibatch often sizes are 1, 2, 4, 8 ---- 256

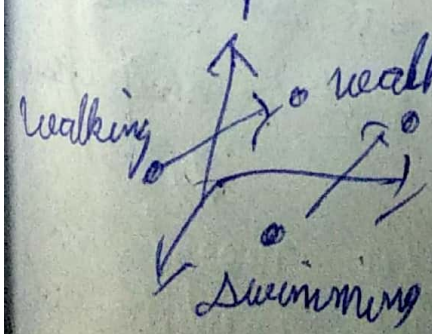
↓ more resources but fast

Small minibatch sizes have noises that help not settling on local minima

Word embeddings

Converting the words to numerical vectors. Done to reduce dimensionality.

Word2vec - Learns to map words to embeddings that contain semantic meaning, eg - word embeddings can learn relationships between present & past tense.



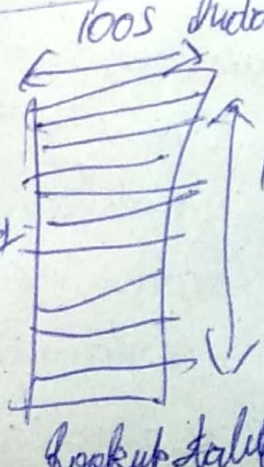
or

addn

Embedding weight matrix :- Imagine if our vocab is long (1000's of words), one hot encoding them would result in a large vector where only 1 value is equal to 1. This is computationally inefficient as a lot of operations occur on values that are 0.

Inputs - embedding layer ⇄ embedding weight matrix.

eg :- $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ Values of hidden layer can be directly looked up from weight matrix as ~~each row~~ multiplication of weight matrix with one hot vector gives row where value in one hot is 1. Each word can be encoded as an integer which is an index that can be used to lookup in embedding weight matrix.



word2vec finds efficient representation by finding vectors that represent words. These vectors have semantic info about words.

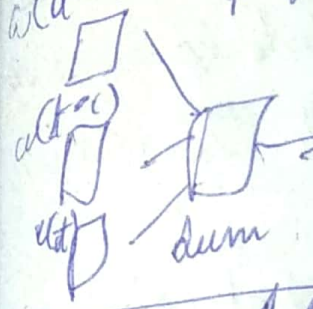
Word2vec is implemented in 2 ways :-

- (BOW) cont. bag of words ; - skip gram

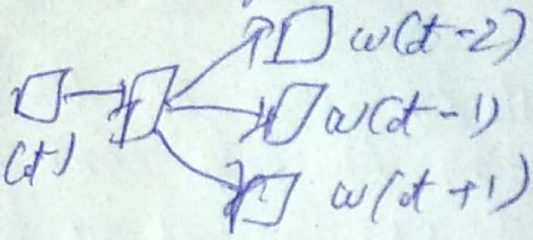
given context predict missing word.

- Give word of interest, predict context

CBOW



skip-gram



Subsampling :- $P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$

we need to remove words that do not provide much context. we can discard such words & this process is called subsampling. For each word w_i we can discard with prob calculated as above.

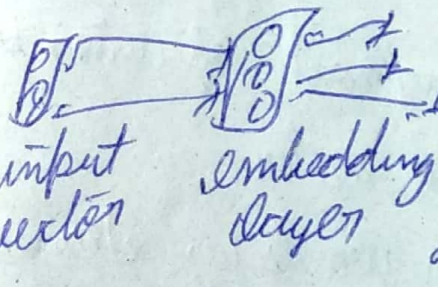
For example : A dataset with 1 million words (10^6)

learn - appears 700 times

$t = 1 \times 10^{-4}$. prob. that we will discard word "learn"

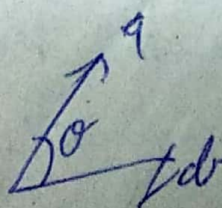
$1 - \sqrt{\frac{(1 \times 10^{-4})}{700/1 \times 10^6}} = 0.622$

$\approx 62\%$



prob that a word chosen from within context window is the etc.

Closest words can be found using cosine similarity



$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \, |\vec{b}|}$

validation words can be encoded as $\vec{a}$ using embedding table & then similarity with each word vector $\vec{b}$ is calculated.

Negative sampling

We have only one or few true outputs in softmax layer. So only some weights in embedding layer will be updated. Instead we can approximate the loss in softmax layer.

Like the embedding layer to map inputs to embedding weights we can use another embedding layer to map outputs to embedding layer. We need a loss func. that considers only the target & few noisy target words.

$$-\log_\sigma(w_{w_o}^T v_{w_I}) - \sum_i^N E_{w_i \sim P_n(w)} \log_\sigma(-w_{w_i}^T v_{w_I})$$

$w_{w_o}^T$ - embedding vector for o/p target word.

v_{w_I} - embedding vector for i/p word.

$\log_\sigma(w_{w_o}^T v_{w_I})$ - log sigmoid of inner product o/p word vector & i/p word vector.

$\sum_i^N E_{w_i \sim P_n(w)}$ sum over words w_i drawn from noise distribution $w_i \sim P_n(w)$.

The first term in loss pushes prob that our network will predict correct word to 1. The second term $\log_\sigma(-w_{w_i}^T v_{w_I})$ we negate & push prob of noise words to 0.

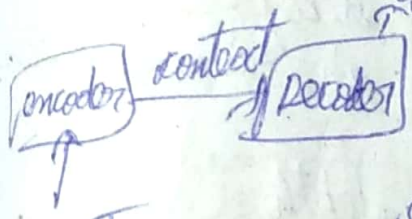
Sentiment analysis with RNN

Attention

Humans don't process an image at once but focus on selected parts, combining the fixations to form an internal representation.

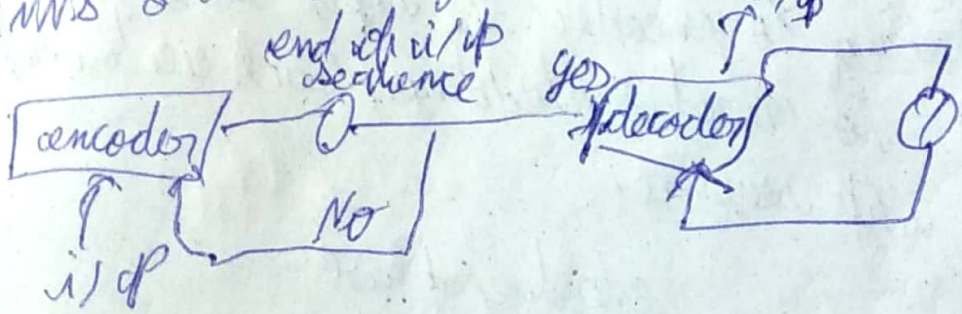
Attention enables sequence-seq models to look at important parts of i/p & output instead of looking at whole input once & generating o/p

Encoders & decoders -



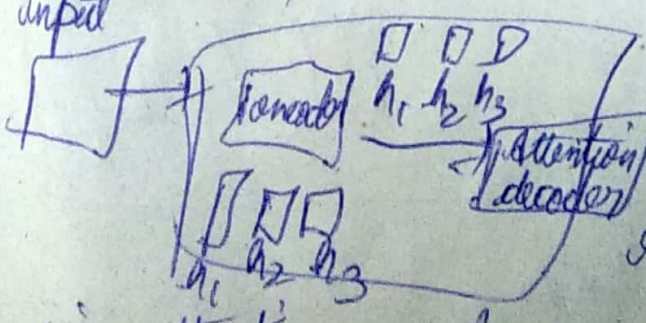
No matter how long the input context size remains same.

encoder & decoder are both RNNs & hence they have loops.



A major limitation is that the context vector produced by encoder no matter how short or long the i/p sequence is. Choosing a reasonable size for this vector is a challenge.

Attention - overview encoding



unlike passing just a single context vector (last hidden state) like in regular encoder-decoder in attention mechanism we pass all hidden

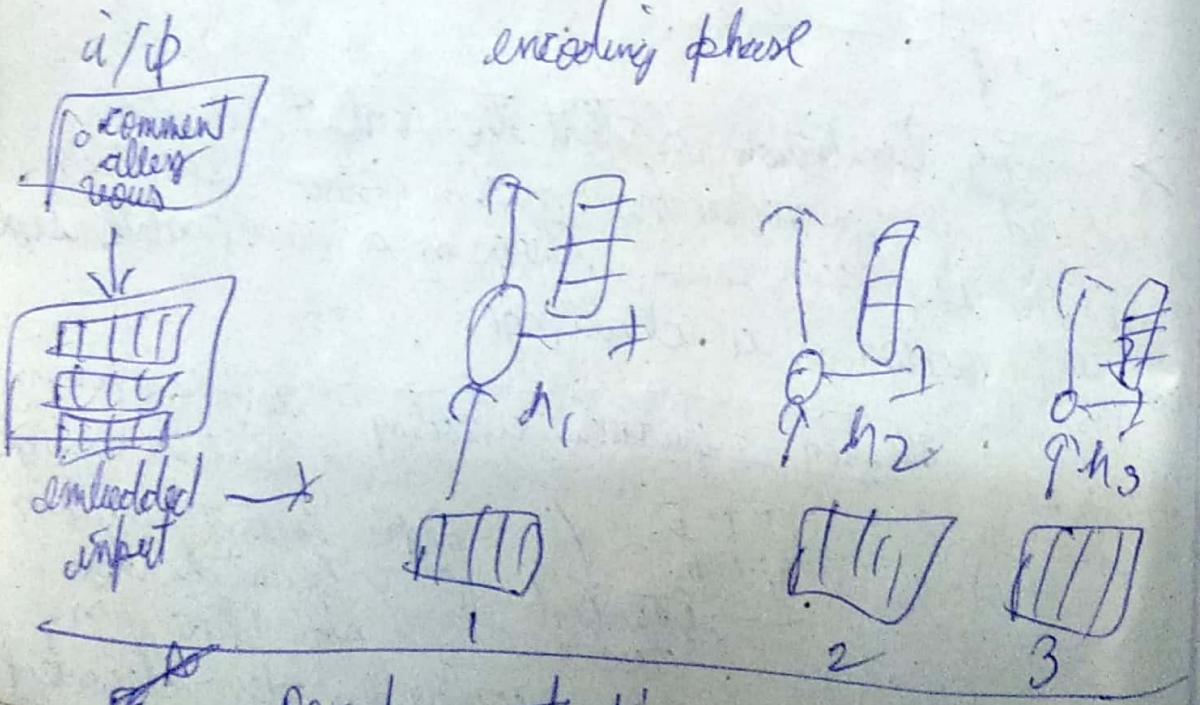
states. This is advantageous as longer sequences have longer context vectors & hence enable capture of information. The main advantage is that attention mechanism enables each hidden state to focus on a word. Eg example first h_1 represents first word & so on. Although each hidden state represents other info preceding it.

. Attention decoder :- At every time step it pays attention to appropriate part of i/p sequence & decodes it (using context vector). The decoder knows which part to pay attention to in training process

---

Attention encoder - in depth

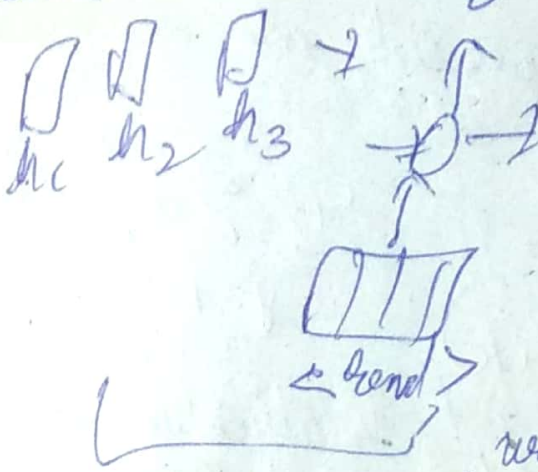
Before feeding to encoder the input sequence is translated to word embeddings. Each vector represents words in i/p sequence.



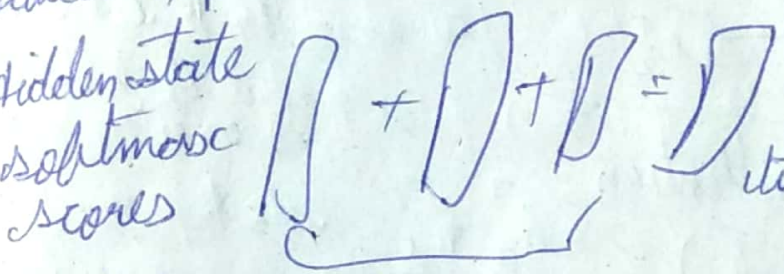
Decoder - attention

In seq2seq without attention only the

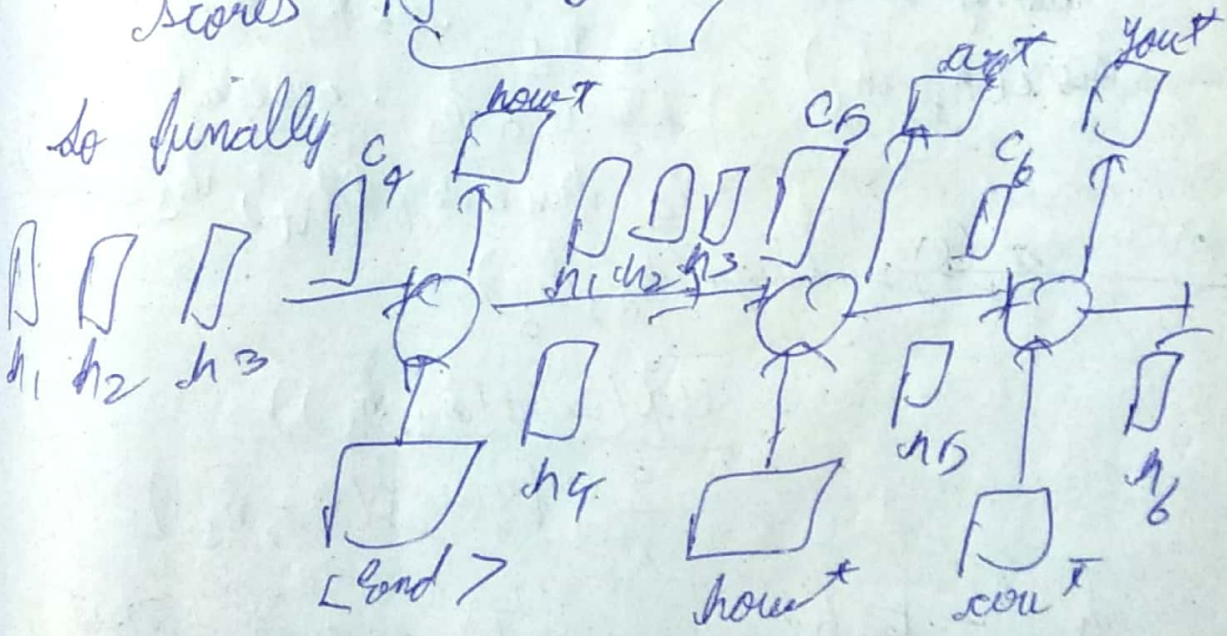
Last context vector is fed to decoder in addition to embedding of <end> token.



It uses a scoring function to score the hidden states. The we pass it through softmax whose output determines how much these context vectors will be expressed in attention vectors before decoder produces o/p.



So finally



2 types of attention - addition & multiplicative

Bahdanau Luong

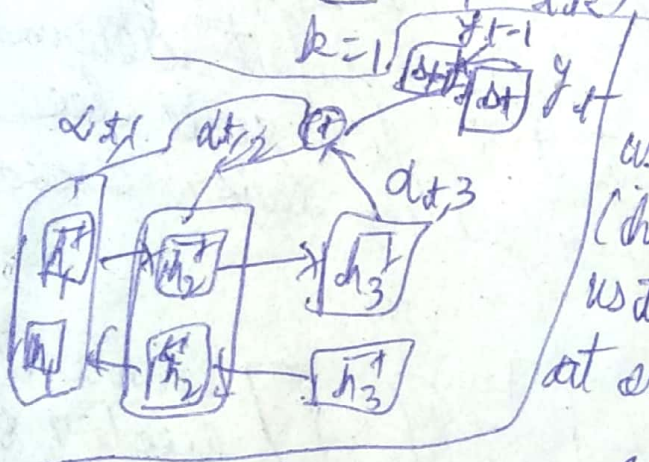
$e_{ij} = V_a^T \tanh(W_a s_{i-1} + U_a h_j)$ h_j - hidden state of encoder.

s_{i-1} - hidden state of decoder at previous

v_a, W_a, U_a - weight matrices learnt during training process.

then they are passed into softmax.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \qquad c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$



Multiplicative attention uses only RNN at top (hidden state). This helped us to use stack of RNN at encoder & decoder.

Luong attention - product of hidden states of encoder & decoder

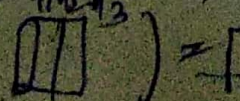
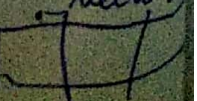
$$score(h_t, \bar{h}_s) = \begin{cases} h_t^T \bar{h}_s & \text{dot} \\ h_t^T W_a \bar{h}_s & \text{general} \\ v_a^T \tanh(W_a [h_t; \bar{h}_s]) & \text{concat} \end{cases}$$

~~$a_t(s) = \text{align}(h_t, \bar{h}_s)$~~

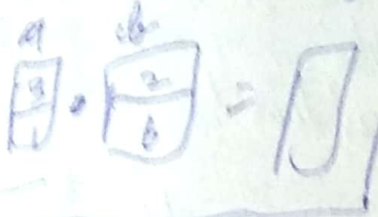
$$= \frac{\exp(score(h_t, \bar{h}_s))}{\sum_{s'} \exp(score(h_t, \bar{h}_{s'}))}$$

$\tilde{h}_t = \tanh(W_c [c_t; h_t])$ - final output

Attention scoring function: It takes hidden state of decoder at decoding step t & hidden state of encoders at each encoding step $\bar{h}_s$

$score(h_t, \bar{h}_s) = score($  $) =$ 

It calculates context vector for all encoder hidden states at once but rather for example taking one:



So multiplicative attention scoring method is

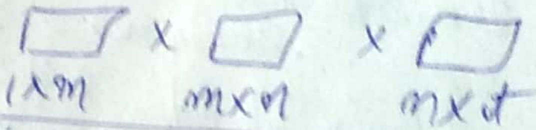
$score(h_t, \bar{h}_s) = h_t^T \bar{h}_s$

However this method assumes both enc & dec use same embedding space

= (t × s) - Attention score for each encoder state with w.r.t current decoder hidden state

But for machine translation encoder & decoder will not have same embedding space. So an alternative method is $score(h_t, \bar{h}_s) = h_t^T W_a \bar{h}_s$

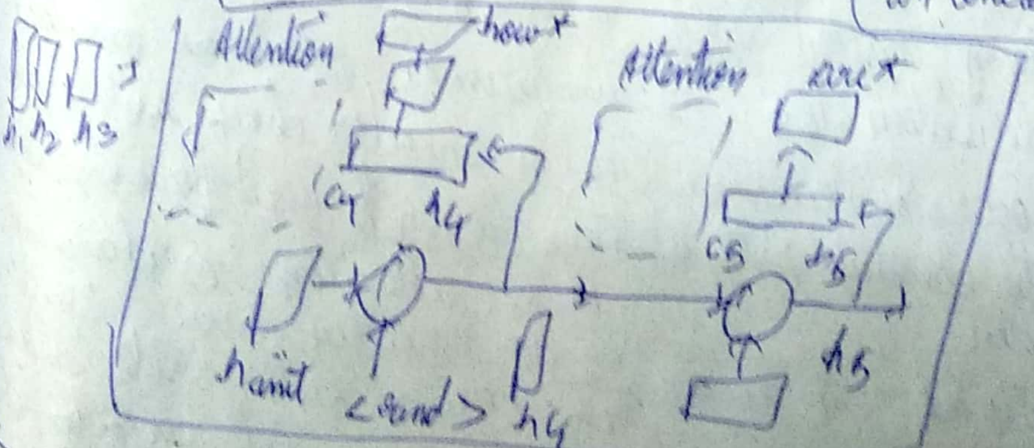
Here a weight matrix is introduced. It is a linear transformation that enables i/p & o/p to have diff embeddings.



t - no of elements in sequence

n - no of hidden units in encoder RNN

Multiplicative attention decoding phase



Attention scoring functions - additive (concat)

Here we concat the decoder hidden state & encoder state & proceed

Formally concat = $v_a^T \tanh(W_a [h_t; \bar{h}_s])$

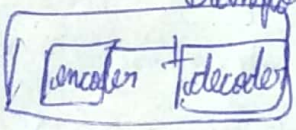
h_t - hidden state at time t

$\bar{h}_s$ - collection of encoder hidden states.

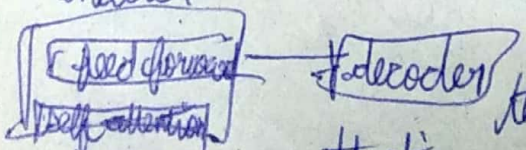
Attention - computer vision applications - caption generation.

Other attention methods

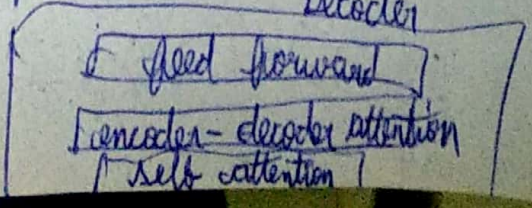
Attention is all you need - uses a single component called transformer & improvise instead of overhead in encoder decoder architecture. A transformer does not look at the i/p seq one by one like encoder but rather takes all of them at once & produces o/p sequence. Transformer does not use RNN.



Transformer - Transformer also uses encoder, decoder architecture but uses feedforward networks instead of RNNs - it uses something called self attention. This allows parallelisation too. Transformer contains a stack of encoders & decoders. Each encoder has sublayers A multi headed self attention layer & feed forward layer



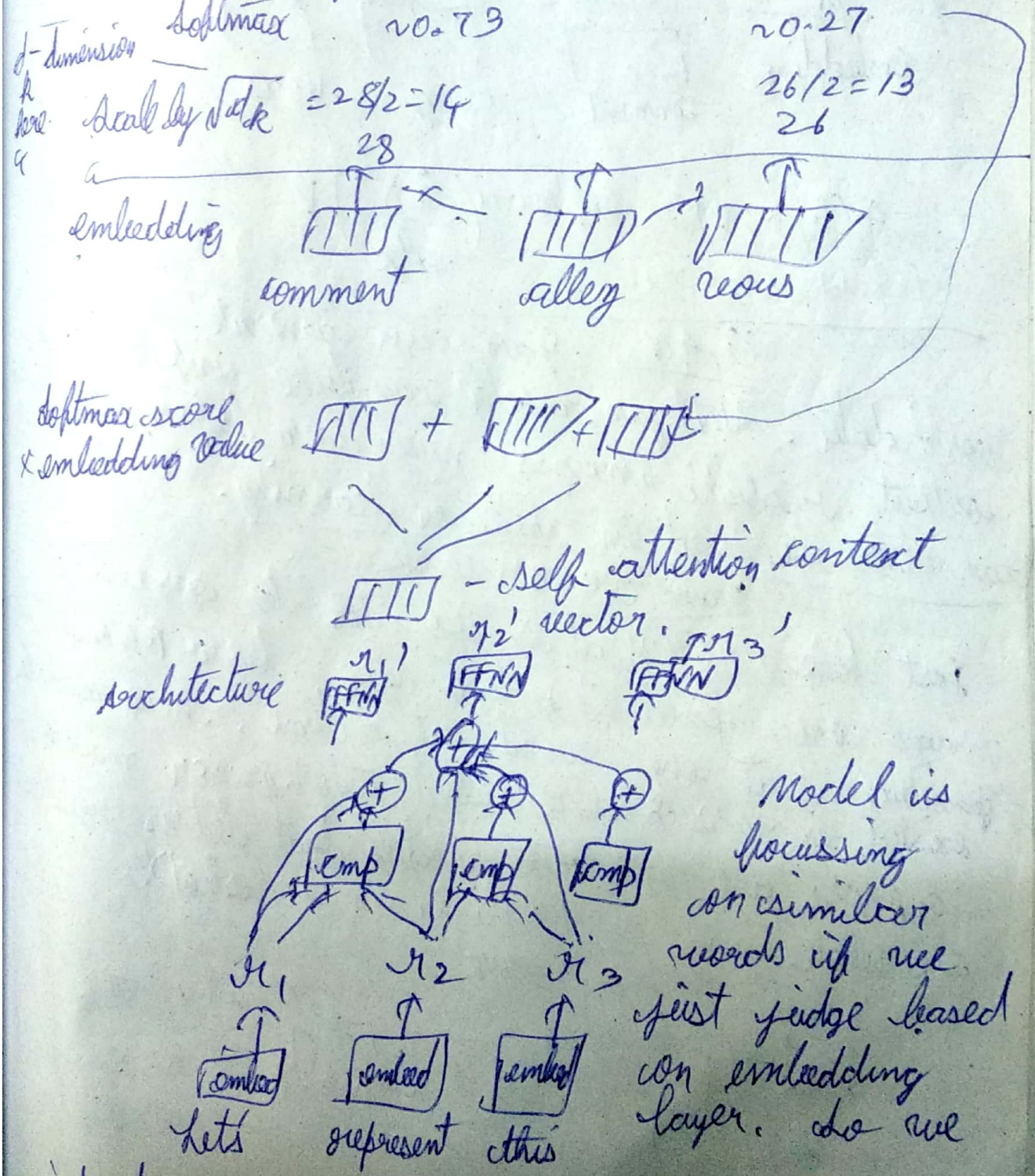
Here attention mechanism is on encoder side rather than on decoder side like previous attention mechanisms. At encoder it plays attention to relevant parts when reading the input. Decoder too has attention components.

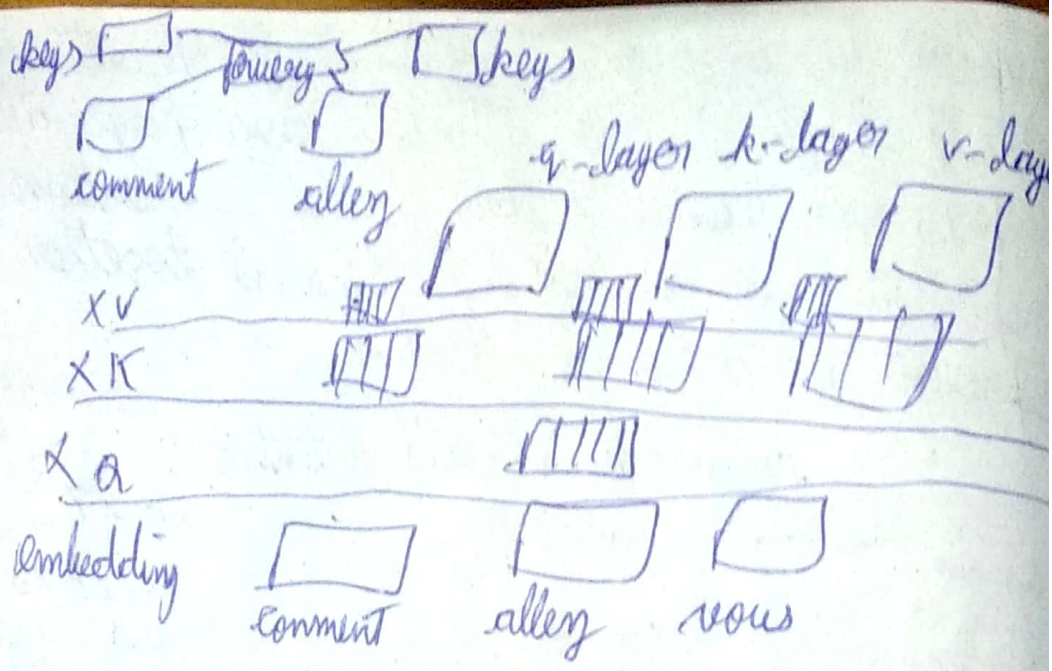


It has 2 attention components.

encoder-decoder attention helps to look at relevant part of inputs & self attention only pays attention to previous decoder outputs. The 3 attention components (1 in encoder & 2 in decoder) together represent multiplicative attention.

Transformers & self attention





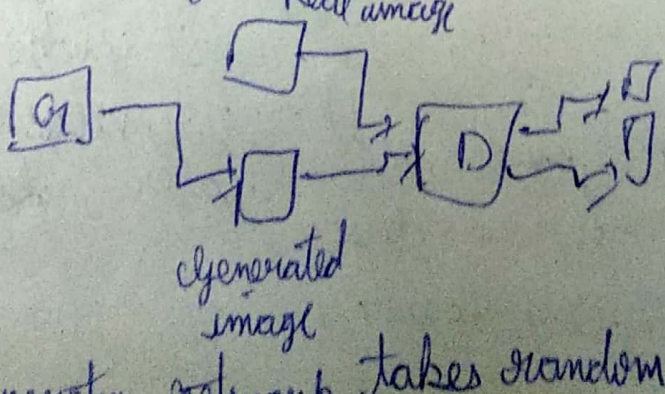
$$A(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

GAN Gan can generate real world data. A trackgan model can take input text & output realistic images never seen before. GAN can also be used in imitation learning.

How GAN's work

Fast like RNN's it is possible to generate images one pixel at a time like how RNN's produce text one word at a time. These models existed in 90's & were called autoregressor models.

GAN's are generative models that let us render the whole image in 1 go (i.e. in one shot).



Generator network takes random noise as i/p.

& runs it through a differentiable function that transforms noise to realistic structure.

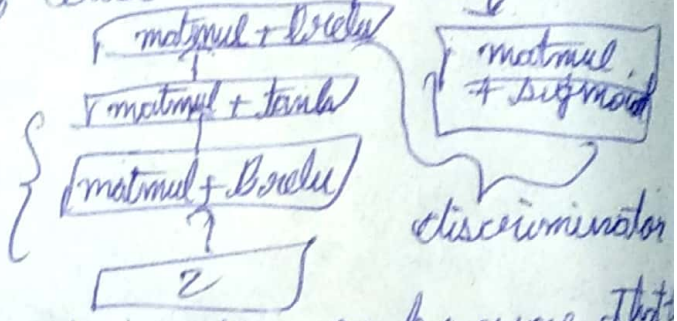
Discriminator guides the generator. During training process, discriminator is shown real images & fake images. D outputs prob that the input is a real image. It outputs a prob close to 1 to real images & near 0 to fake images. D has to output prob of 0.5 everywhere for convergence.

Games & equilibria :- The adversarial in GAN stems from the fact that generators & discriminators compete with each other. This comes from game theory. Aim of game theory is for agents to reach equilibrium. example :- rock, paper, scissors. If both players choose their moves uniformly at random then even if they change their strategy it wouldn't affect the outcome & they would tie (equilibrium).

Generators & discriminators have their own loss functions. So for GAN there is a value function. Generator wants to minimize the value function & discriminator wants to maximize the value function. We attain equilibrium when value func is at minimum for generator & at max for discriminator. This is a saddle point.

If we analyse game of GANs' the local m
for discriminator occurs when it accurately as
that i/p is real rather than fake, This prob
given by ratio of data density at input &
model density induced by generator at input.
This ratio determines how much prob mass
comes from data.

Training GANs we must make sure
that both generator & discriminator have atleas
one hidden layer.



Leaky relu is used as they make sure that
gradient flows through entire architecture.

For GAN's 2 optimization algo must run
simultaneously. d-loss, g-loss.

Discriminator uses BCE loss as it's just
classification. As sigmoid o/p are unstable at time
we multiply labels by 0.9 values closer to 1 but
<1 (0.9) so that o label goes to 0 & 1 is labelled
accordingly (eg ÷ 0.9).

Our generator also uses BCE loss but with
labels flipped.

To scale GAN's to work for

large images we use convolutional layers.

discrim { matmul + sigmoid
conv + lrelu
conv + tanh

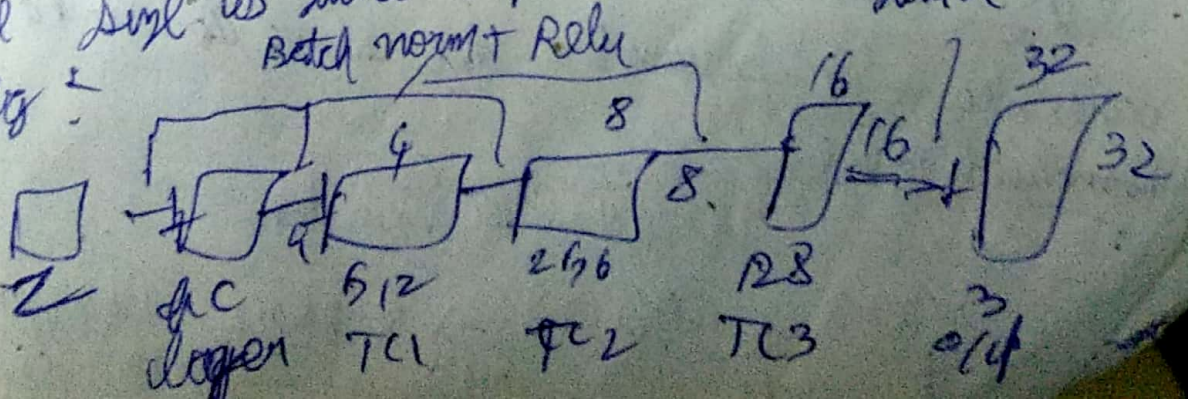
generator { conv + lrelu
reshape
z

DCGAN - discriminator :-

i/p → conv1 → conv2 → conv3 → flatten → fc layer → sig ← fc layer

Every hidden layer has leaky relu & batch normalization. Batch normalization scales the output of the layer to have mean of 0 & variance of 1.

DCGAN generator :- As usual input is a latent vector z which is fed to GAN to generate realistic images. We need to upsample the inputs to generate the realistic image. Instead of linear interpolation we can use transposed convolutions. For transposed convolutions for every 1 pixel in i/p we move 2 pixels in o/p image. Stride determines o/p image size. For stride of 2 o/p image size is twice i/p image size.



Batch normalization:

In training we normalize each layer's i/p by using mean & std dev. This is batch statistics. Batch norm normalizes o/p of previous layer by subtracting the batch mean & dividing by batch std dev.

Internal covariate shift: It refers to change in distribution of inputs to diff layers. Training is more efficient when dist of inputs to each layer is similar.

Mean: average value of o/p of any layer before passing them non-linear act func.

$$\mu_B$$

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

Normalize: $\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$

ϵ is a small constant that is added to avoid divide by 0. It also slightly increases variance for each batch.

$\hat{x}_i$ - normalized value, $y_i \leftarrow \gamma \hat{x}_i + \beta$. γ & β are learnable parameters that serve to scale & shift normalized value.

DCGAN - discriminator

32x32x3 i/p → conv1 → 16x16x128 → conv2 → 8x8x256 → conv3 → 4x4x512 → flatten →

Pix2pix & cycle gan :-

cycle gan is used for image to image translation.

Image of cats (training data)

Random noise

cycle gan can also be used for image colorization, super resolution. It is basically also used to transform (map) image in one domain to another domain.

Designing loss functions.

Here we need to compare true & generated image.

eg :- Image colorization, we can use euclidean distance comparing pixel by pixel. $L(g,\hat{g}) = \frac{1}{2hw}\sum \|g - \hat{g}\|_2^2$

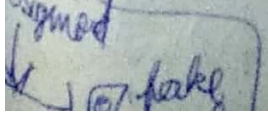
In practice this is too simple.

GAN recap

Generator tries to generate fake images to fool best D

$$\min_G \max_D E_{z,x}[\underbrace{\log(D(G(z)))}_{\text{fake}} + \underbrace{\log(1-D(x))}_{\text{real}}]$$

If discriminator outputs 0.5 it means generated & real image are indistinguishable. It means generator has learnt a mapping to map a latent vector z to a realistic image.



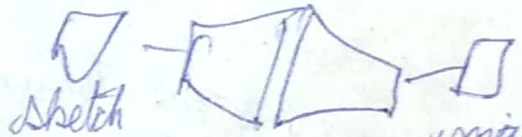
Latent means hidden or concealed

space, a compressed feature level representation of image. For image to image translation the training dataset is paired image data.

x_i y_i G_x; $x_i \rightarrow y_i$.

G_x should be indistinguishable from real image y.

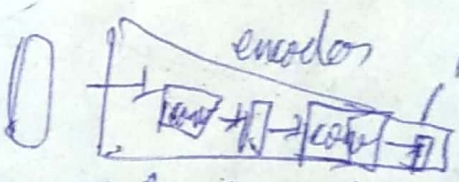
Pix 2 pix: Rather than like an regular GAN, pix 2 pix architecture involves some modifications in generator. For translating an input sketch to an image.



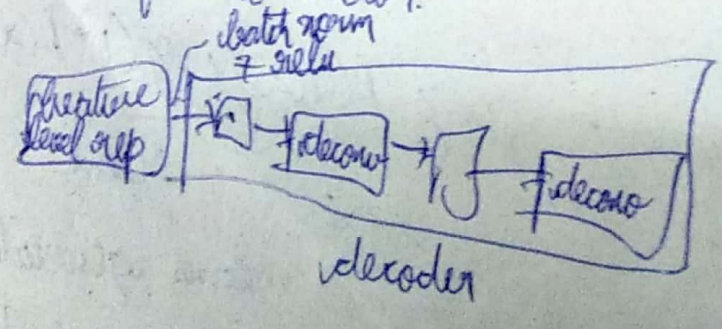
feature representation

The learned feature representation will help us in generating image from the sketch.

Encoder:



By compressing input the encoder captures the content of the image in small feature vector.



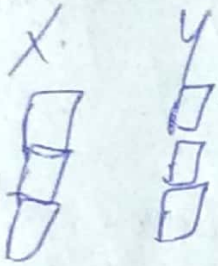
Discriminator pix 2 pix:

Instead of taking a single image as input & outputting a prob for real or fake here discrim takes a pair of images & o/p if real pair or fake pair.

$$\min_G \max_D E_{xy}[\log D(x,G(x)) + \log(1 - D(x,y))]$$

real fake

cycle gan's & Pix to pix works only with paired images & learns mapping $G_1 : x_i \to y_i$. But paired image collection is time consuming. So instead of pairs of images we have sets of input & o/p - cycle consistent networks help in image - image translation in this case.



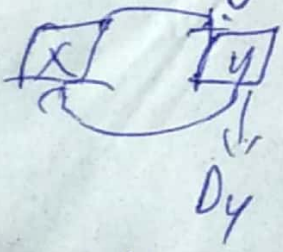
$G_1 : X \to Y$ (Mapping)

X - set of images, x denotes a single image from X.

$G_1(x)$ should look like it comes from set Y.

Unpaired data risk: mode collapse. For example neural net might map multiple horses to same zebra image.

Adversarial loss can be used as a constraint we need to add another constraint. Inverse mapping G_Y to X : $G_Y \to X$. do this ensures cycle consistency



$G_{X \to Y} : G_X \to Y$

$G_{Y \to X}$ $Y \to X$.

$G_{Y \to X}(G_{X \to Y}(x)) \approx x$

Consider style transfer. It can be considered as mapping from X domain to Y domain. Here cycle consistency is needed. For this we can use 2 discriminators : D_Y & D_X. D_Y checks.

in real image(y) and generated image $G_{x \to y}(x)$ as i/p
D_x takes in $G_{y \to x}$ & takes in generated image.
$G_{y \to x}(y)$ & x(real image) in x domain as i/p

$x \to G_{x \to y} \to$ generated $\to G_{y \to x} \to \hat{x}$
$G_{x \to y}(x)$

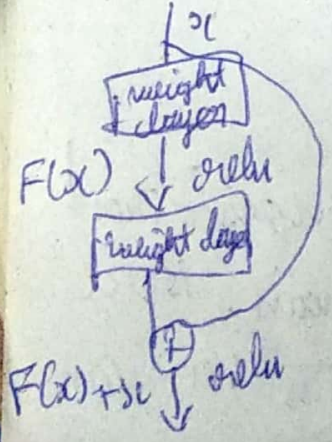
If $x \& \hat{x}$ are same then forward cycle consistency holds good. Diff between $x \& \hat{x}$ is called cycle consistency loss.

cycle consistency loss = transversial losses + forward and backward cycle losses.

$= L_x + L_y + \lambda L_{cycle}$.
weight reduce

Generator for cycleGan, Here we use residual blocks which rely on connecting o/p of one layer with input of an earlier layer.

residual function: $F(x) = H(x) - x$, diff bet mapping applied to x & original

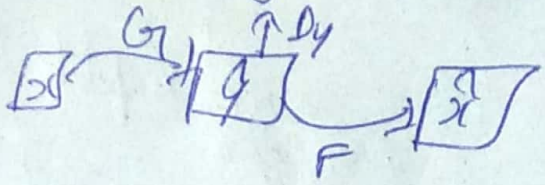


$F(x)$ - typically 2 conv layers + norm layer + relu inbetween.

$\therefore H(x) = F(x) + x$, $y = F(x) + x$

Idea is that it is easier to optimize dual function than optimizing $H(x)$.

D & G losses :- In addition to disc loss we need to calculate cycle consistency loss. This loss ensures reproducability



G - X to Y F - Y to X.

Here as opposed to cross entropy loss, least squares can be used. GANs treat disc as classifier with sigmoid cross entropy loss function. This can lead to vanishing gradient problem. To overcome this we can use least squares loss func.

Discriminator loss :- mean squared errors between o/p of discriminator given an image & target value 0 or 1, depending on whether it should classify image as fake or real.

Generator losses :- In generator we might generate fake images that look like they belong to set of X images but are based on real images in Y & vice versa. Real loss can be computed by looking at o/p of disc as it's applied to fake images. Generator aim is to classify fake images as real images.

cyclegan.

Discriminator training :-

1) Compute discriminator loss Dx on real images
2) Generate fake images that look like domain X based
3) Compute fake loss for Dx
4) Compute total loss & perform backprop & Dx opt.
5) Repeat 1-4 with Dy

Generator training

1) Generate fake images in X based on images in Y.
2) Compute generator loss based on how Dx responds to fake X.
3) Generate reconstructed Y images based on fake X images in step 1.
4) Compute cycle consistency loss by comparing reconstruction with real Y images.
5) Repeat 1-4 swapping domains
6) Add up generator & reconstruction & perform backprop