14/3/18

SRN

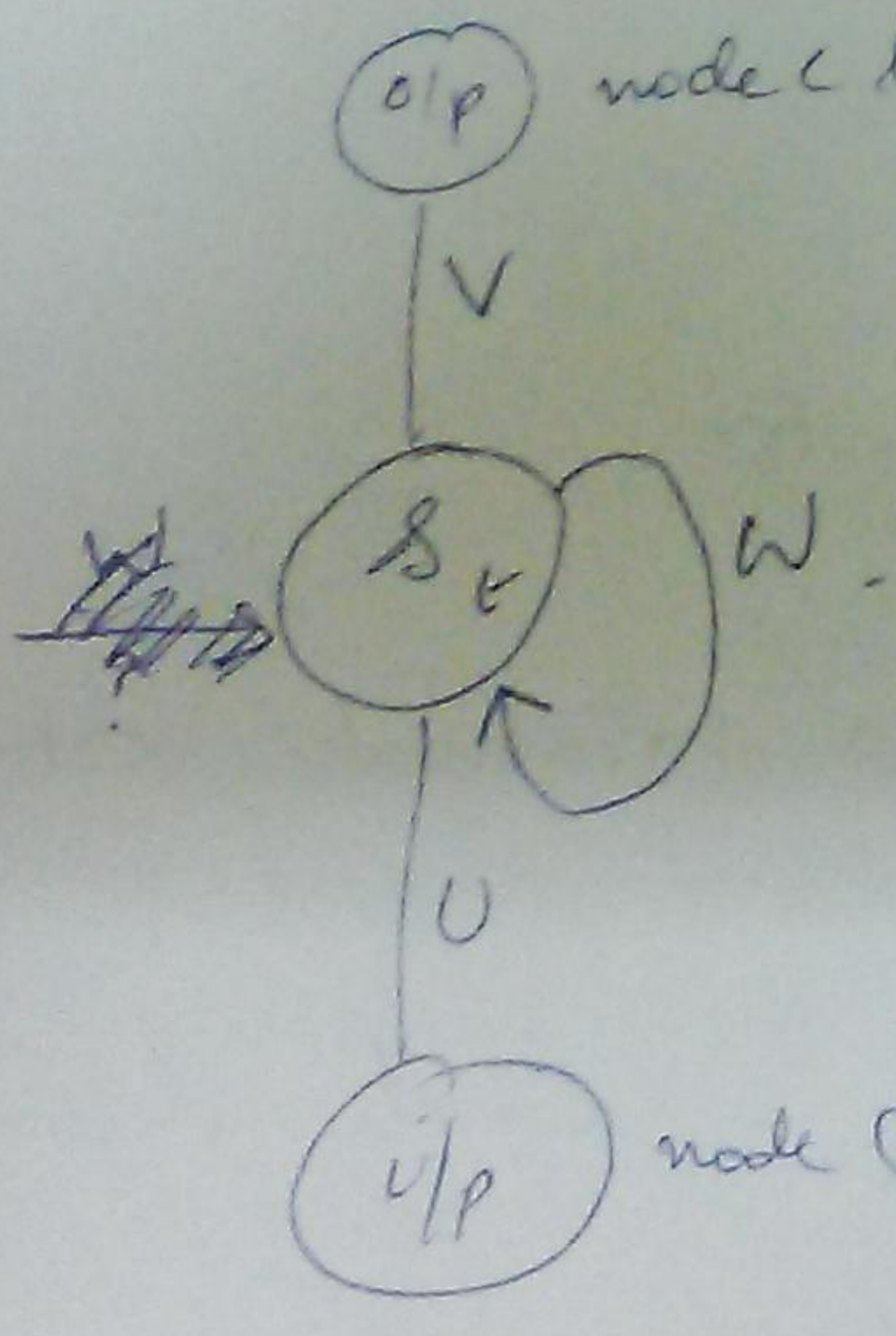
Recurrent Neural Networks

ANN: cant deal with temporal data
ANNs lack memory
ANNs have fixed Architecture
RNNs are biologically realistic (recurrent connectivity in the visual cortex of brain)

Typical applications

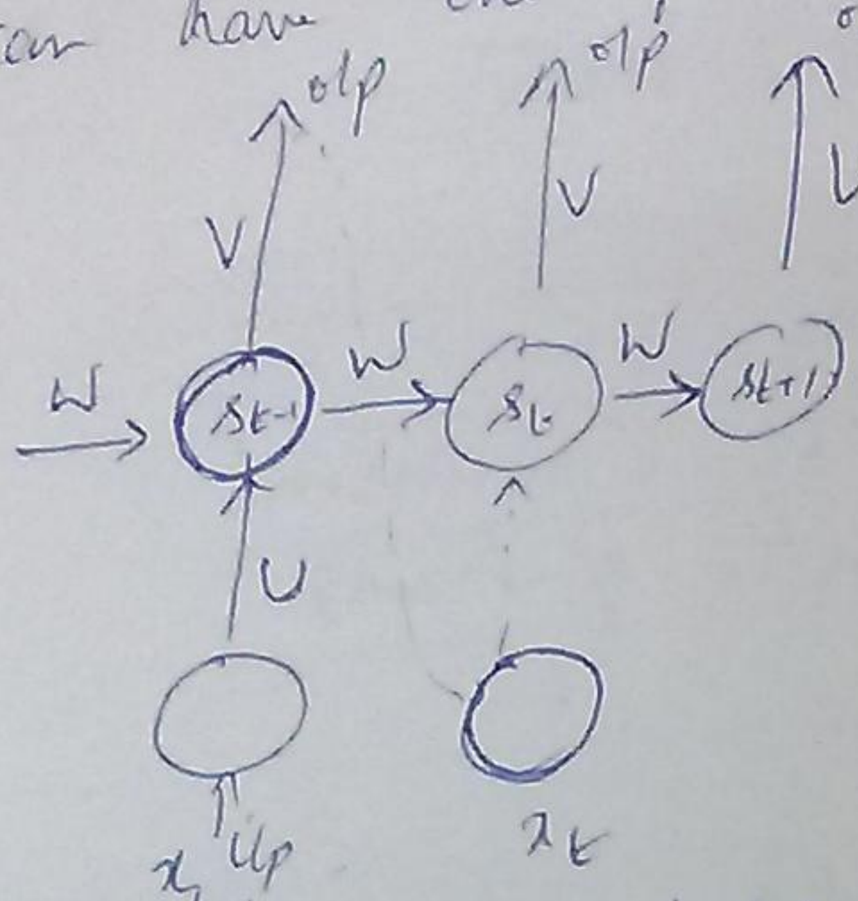
- Image captioning (one i/p - multiple o/ps)
- Sentiment analysis (multiple i/ps - one o/p)
- Machine translation (multiple i/ps - multiple o/ps)
- Video captioning (multiple i/ps - multiple o/ps)
- Speech analysis.

Typical RNN



S_t = state (hidden) of the Recurrent ne
(actually a layer). The state S_t at
any point of time 't' is S_t. Ca

If we unfold or unroll the RNN one time we can have the following



(one to many.

Ex- Image captioning or even word / sentence prediction)

there are 3 hidden states s_{t-1}, s_t, s_{t+1} (This can go on and on)

$$s_t = f(Ws_{t-1} + Ux_t) \quad -(1)$$

$$y_t = Vs_t$$

$$y_{t-1} = Vs_{t-1}$$

$$y_{t+1} = Vs_{t+1}$$

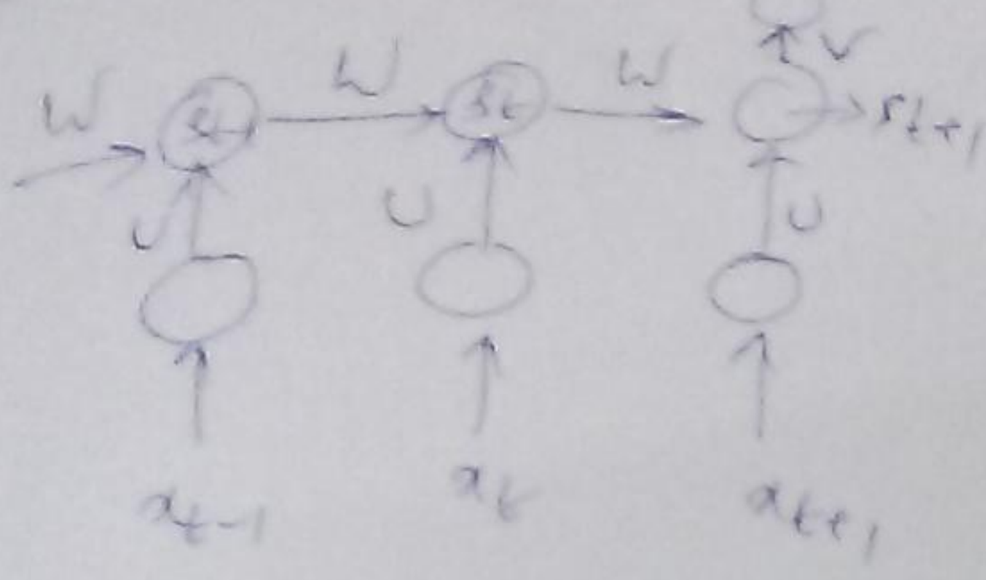
Tanh is a common function.

$$s_t = \tanh(Ws_{t-1} + Ux_t).$$

the o/p can have a softmax layer too

so $y_t = \text{softmax}(Vs_t)$ (a probabilistic vector)

[o/p at $t+1$: y_{t+1} depends on s_{t+1} which depends on s_t which in turn depends on s_{t-1} and so on. check eqn (1)]



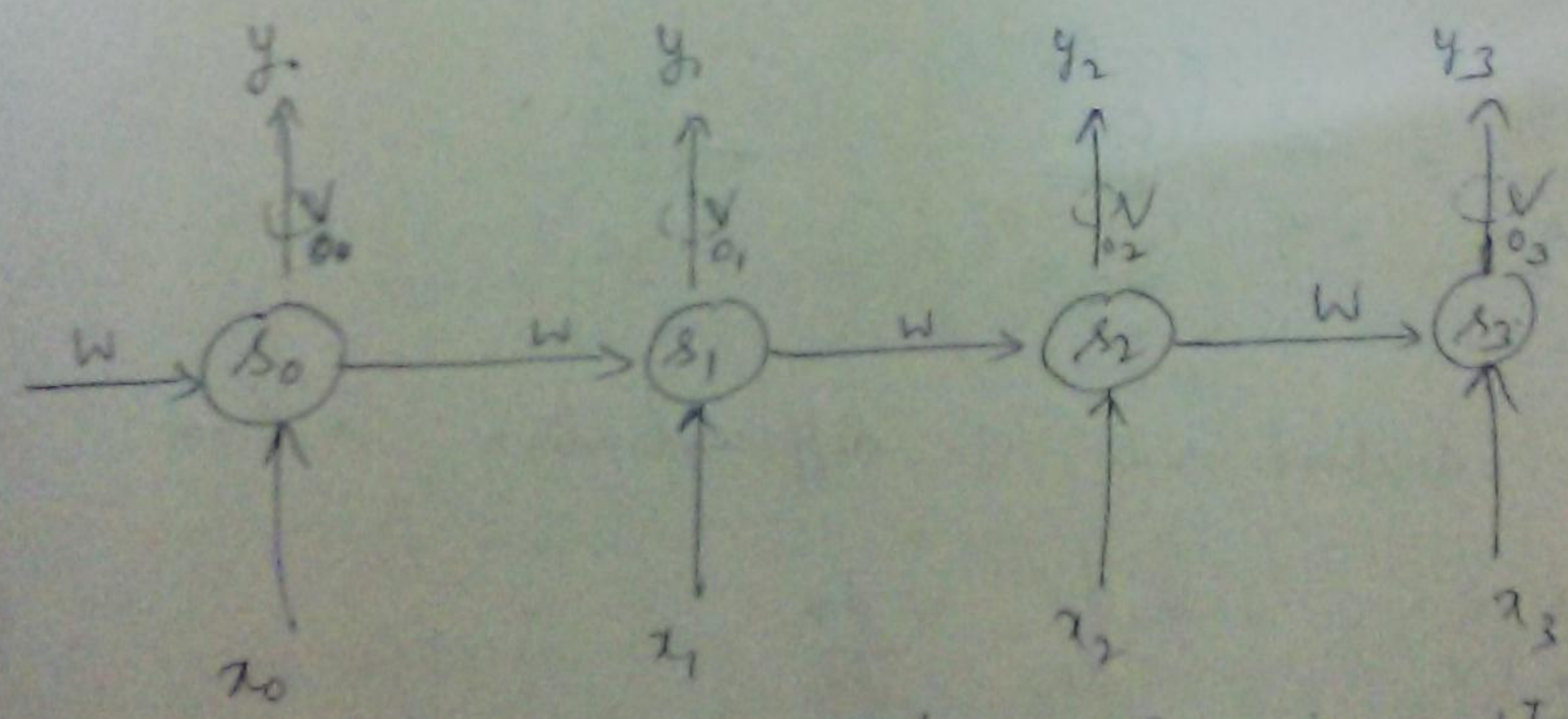
(many to one.

Ex: Sentiment (classifier)

After ALL inputs are fed in, the classification is o/p of $\text{softmax}(Vs_{t+1})$. Here s_{t+1} has been computed taking all previous hidden states and previous i/ps.

Makes sense also, because for a given sentence, the sentiment can be guessed only after ALL words of the sentence have been processed.

A typical RNN unfolded on time

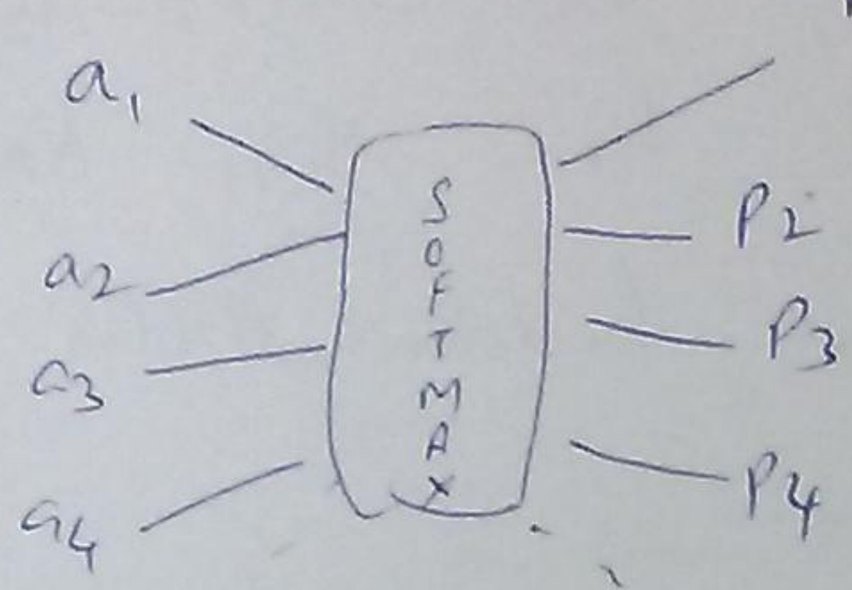


0, 1, 2, 3 are instances of time.. So in reality there is only ONE RNN neuron.

$O_1 = V s_1$ $\quad O_2 = V s_2$

$y_1 = \text{softmax}(O_1)$

Softmax detour



$$P_i = \frac{e^{a_i}}{\sum_{j=1}^{n} e^{a_j}}$$

If we have to find the differential of softmax

we have to resort to vector calculus)

$$\frac{\partial P_i}{\partial a_j} = \frac{\partial}{\partial a_j}\left\{\frac{e^{a_i}}{\sum_{j=1}^{n} e^{a_j}}\right\} \quad -(2)$$

From quotient rule of differentiation, we have

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v\cdot\frac{du}{dx} - u\cdot\frac{dv}{dx}}{v^2}$$

Applying on (2), we have

$$\frac{\partial P_i}{\partial a_j} = \frac{\sum_{j=1}^{n} e^{a_j} \cdot \frac{\partial}{\partial a_j}(e^{a_i}) - e^{a_i} \frac{\partial}{\partial a_j}\left(\sum_{i=1}^{n} e^{a_j}\right)}{\sum_{j=1}^{n} e^{a_j} \sum_{j=1}^{n} e^{a_j}}$$

2 cases. First, $i \neq j$.

$$\frac{\partial P_i}{\partial a_j} = \frac{0 - e^{a_i} \cdot e^{a_j}}{\sum_{j=1}^{n} e^{a_j} \cdot \sum_{j=1}^{n} e^{a_j}} = \frac{-e^{a_i}}{\sum e^{a_j}} \times \frac{e^{a_j}}{\sum e^{a_j}}$$

$$\frac{\partial P_i}{\partial a_j} \text{ when } (i \neq j) = -P_i \cdot P_j \quad -(3)$$

2nd case, $i = j$.

$$\frac{\partial P_i}{\partial a_j} = \frac{\sum_{j=1}^{n} e^{a_j} \cdot e^{a_j} - e^{a_j} \cdot e^{a_j}}{\sum_{j=1}^{n} e^{a_j} \cdot \sum_{j=1}^{n} e^{a_j}}$$

$$= \frac{e^{a_j} \sum_{j=1}^{n} e^{a_j}}{\sum e^{a_j} \cdot \sum e^{a_j}} - \frac{e^{a_j} \cdot e^{a_j}}{\sum e^{a_j} \cdot \sum e^{a_j}}$$

$$\frac{\partial P_i}{\partial a_j} (i = j) = P_j - P_i^2$$

$$= P_j(1 - P_j) \quad -(4)$$

Back on our RNN, now

$E_t(y_t, \hat{y}_t)$ → Cross entropy. (at time t)

$$= -y_t \log(\hat{y}_t)$$

$E(y, \hat{y})$ (Sum of all errors across various time

$$E(y, \hat{y}) = -\sum_t y_t \log(\hat{y}_t)$$

No BPTT (w.r.t V)

$$\frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial o_3} \frac{\partial o_3}{\partial V}$$

Now $o_3 = s_3 \cdot V$, so $\frac{\partial o_3}{\partial V} = s_3$.

$\frac{\partial E_3}{\partial V} = (\hat{y}_3 - y_3) \cdot s_3$ (long Proof → not important)

Now coming to the impact of W (and also)

W impacts E_3 via s_3. But the same W impacts E_3 thru s_2 (which impacts s_3) and W also impacts E_3 through s_1, which impacts s_2 which impacts s_3

So $\frac{\partial E_3}{\partial W} = \sum_{k=0} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial o_3} \frac{\partial o_3}{\partial s_k} \frac{\partial s_k}{\partial W}$.

$$\frac{\partial s_k}{\partial W} = \frac{\partial s_k}{\partial s_{k-1}} \cdot \frac{\partial s_{k-1}}{\partial s_{k-2}} \cdots \frac{\partial s_0}{\partial W}$$

note that $s_t = \tanh(W s_{t-1} + U x_t)$

for a tanh function, $\tanh(x)' = \frac{\partial}{\partial x}\tanh(x) = 1 - \tanh(x)^2$

so if $z_t = W s_{t-1} + U x_t$, then $\frac{\partial s_t}{\partial z_t} = 1 - s_t^2$

$(s_t = \tanh(z_t))$

and we have $\frac{\partial z_t}{\partial W} = s_{t-1}$ (provided s_{t-1} is not a function of s_{t-1})

Vanishing Gradient / Exploding gradient

Due to small derivatives and numerous multiplica-
the derivatives will be near 0 and updates are impossible.

LSTMs to the Rescue:

Long-short-term Memory:

Recall that in the Vanilla RNN, each hidden neuron is actually a hidden layer. (when unfolded)

LSTM:

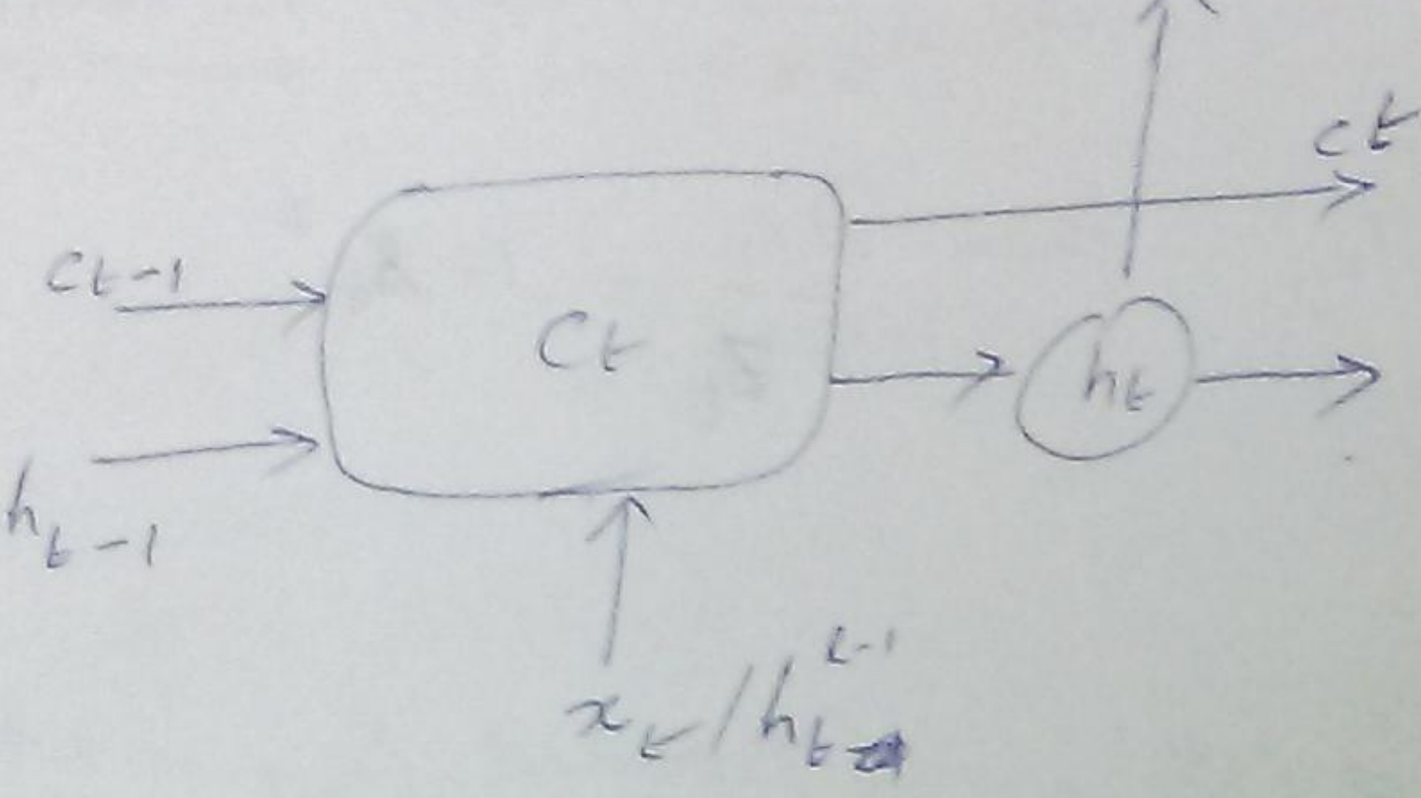
Hidden States still exist.

h_t or s_t is now a function of something called as the "cell state".

so $h_t = f(c_t)$

$c_t = ff(c_{t-1}, x_t)$

$c_t = ff($ previous all states & previous hidden states in depth + $x_t)$

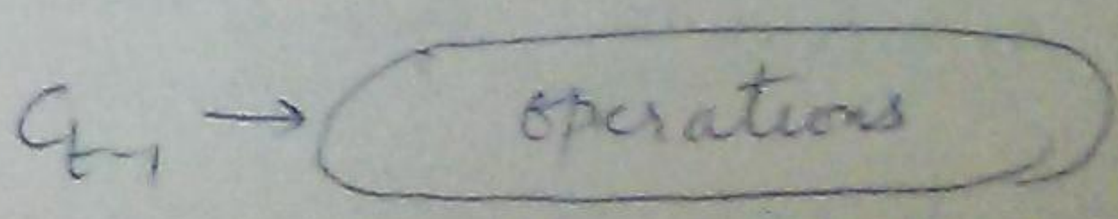


(l-1) previous depth
ignore for now

Consider only one hidden neuron.

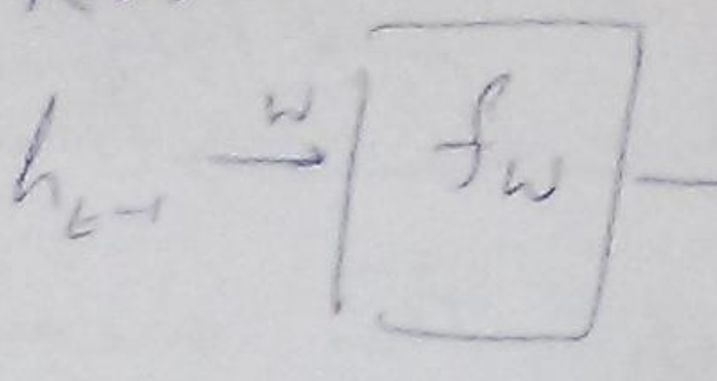
An LSTM cell can be compared with a conveyor belt making a burger (say).
There are workers at strategic points on the belt as the burger moves over the belt, the workers enhance or enrich the burger.

cell state => conveyor belt



So as information passes through a cell, it may be completely modified or partly modified or may be passed as is.

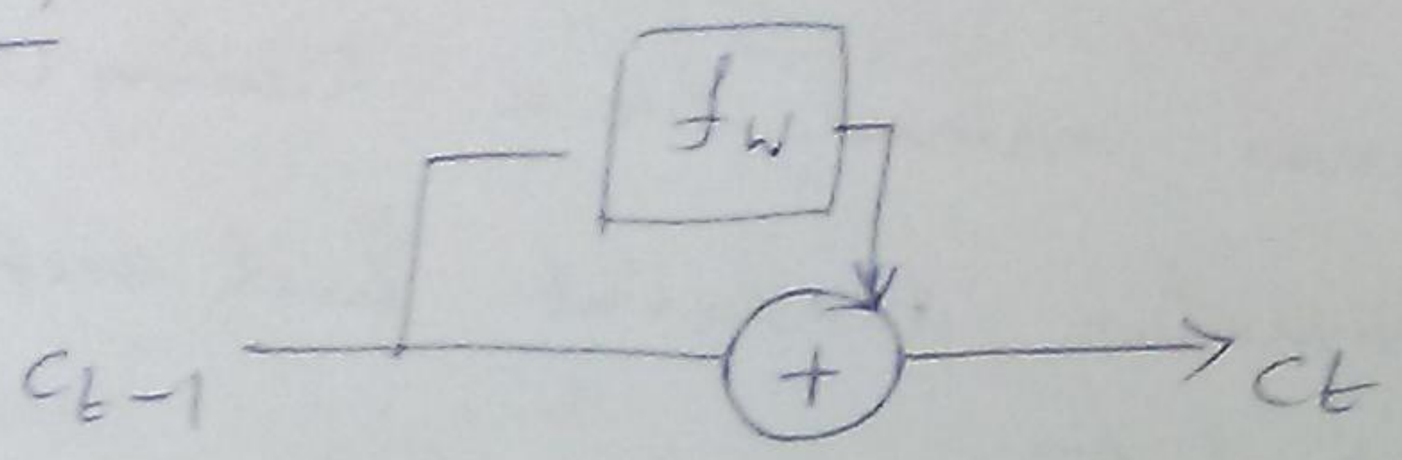
Traditional RNN



$h_t = f(h_{t-1}, W)$

(derivatives multiply and hence can vanish)

LSTM



(Extreme simplification)

c_{t-1} (+) some Vector = c_t.

f_W = some function (not the same as vanilla RNN)

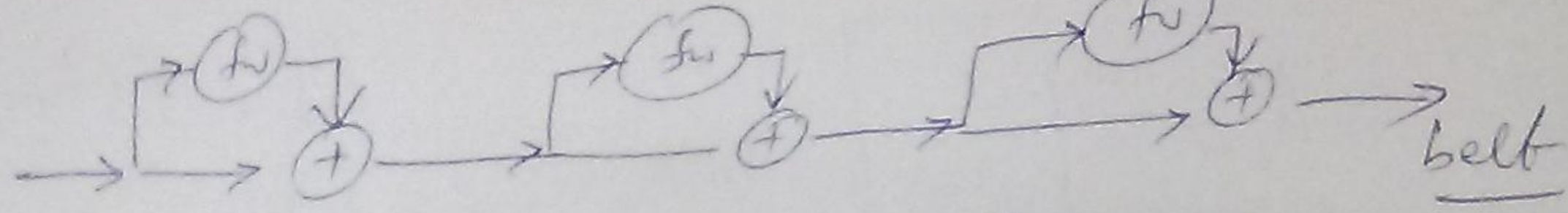
$$c_t = c_{t-1} + f_W(c_{t-1})$$

This can be expanded as

$$c_t = c_1 + f_W(c_1) + f(c_2) \cdots f_W(c_{t-1}) \quad - (I)$$

Why is this better?

Because when we differentiate, we differentiate a sum of terms. So the differentials will now add up (not multiply). So we have a (gradient super-highways) Even if some gradient go zero, it may not result in a sum total of zero



So, for we have added info Thus means Ct will Keep growing and at some point of time we can get an explosion'

So, the cell behaves instead like a Memory Unit:

1) It 'forgets' information it doesn't find useful from the previous cell-state

2) It writes new information it does find useful from the current input

3) It "reads out" a part of its information to help the computation of the hidden state

(Reset - write - read)

The Summation part is "writing" info. → Add info to the cell

Resetting part is Multiplicative and occurs before writing to memory. The multiplying vector is full of 1s and 0s and we perform an index-wise multiplication on the previous cell-state to eliminate the useless information.

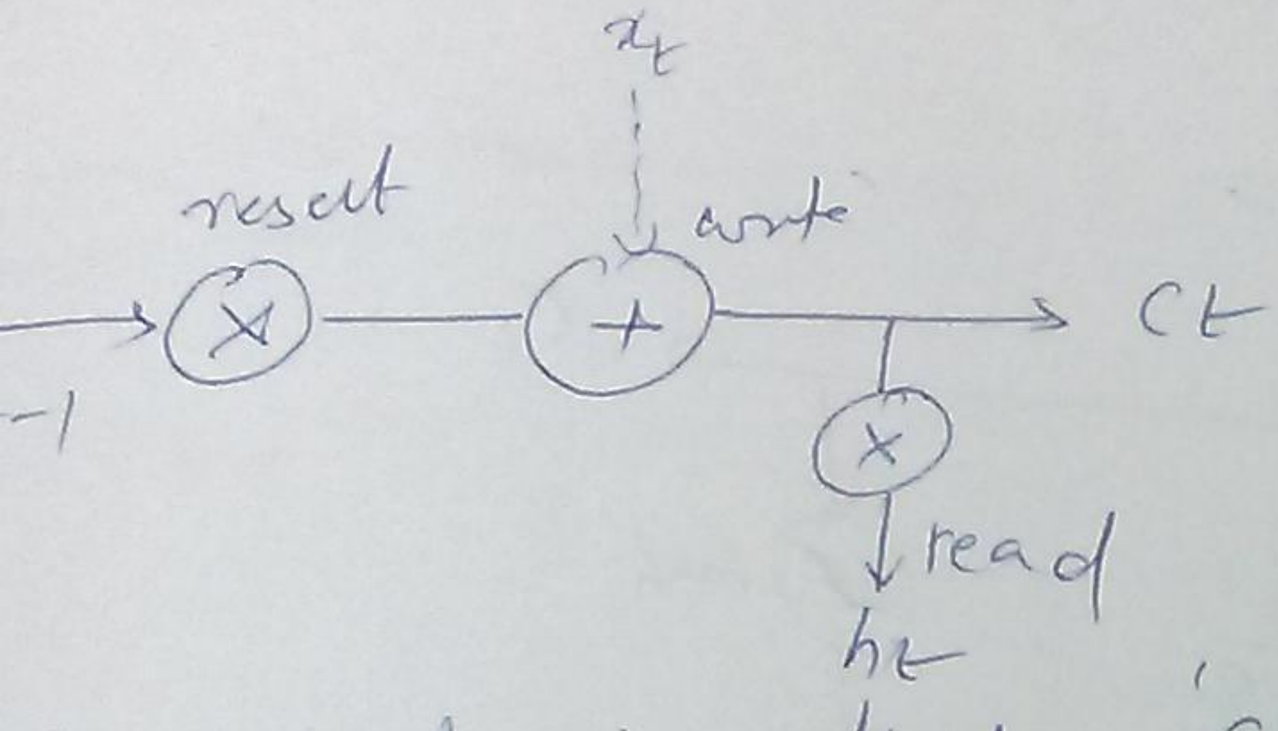
Reading Part is also multiplicative but doesn't modify the cellstate - Instead it modifies info flowing into the hidden state

nd thereby influences the hidden state. We do element-wise multiplication

$$\begin{bmatrix} a \\ b \end{bmatrix} \odot \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac \\ bd \end{bmatrix}$$

if $a = 0$, info in c is lost.

o a slightly enhanced version of our cell will be



his concept is called Gating

Gates

1. Forget Gate (actually 'remember Gate')

A sigmoid function is used to compute the 'Forget Gate'. Its values are between 0 & 1. If we get '0' we forget. If we get '1', we remember fully.

Write Gate (g) → Partly Responsible for write process. Stores a value between -1 and 1 and represents how much we want to add (input to the cell state) ½

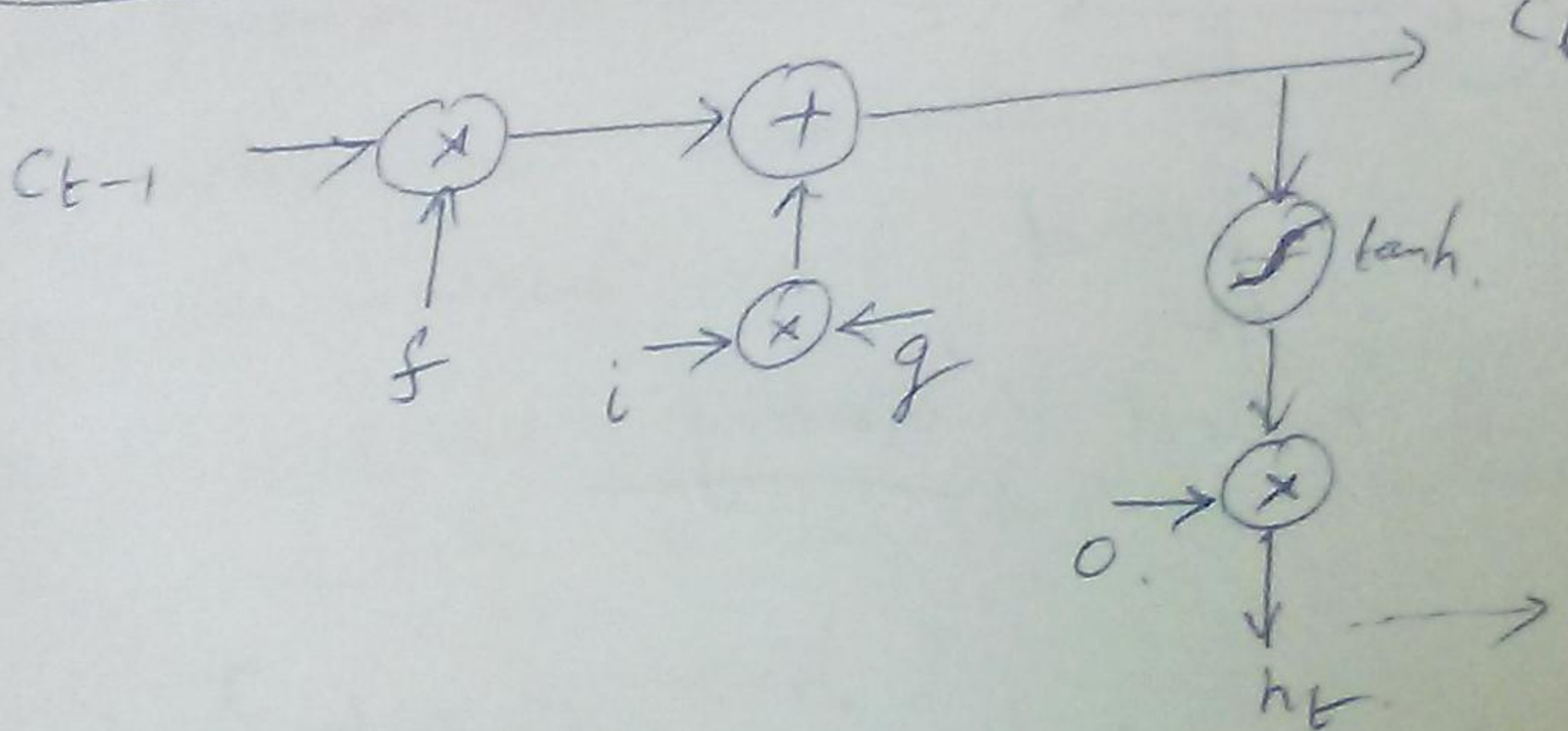
Computed with a 'tanh' function.

Input gate (i) — The other gate responsible for the 'write' process. How much of 'g' is allowed in?

Incrementing or decrementing counter. Computed with a sigmoid function.

4) Output Gate: Passed through sigmoid, and is a number between 0 and 1 → Moderates or modulates which aspects of the cell-state can be drawn by the hidden state for "Read" operation. (tanh) also

slightly enhanced version.



$$C_t = f \odot C_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(C_t)$$

(No activation function. C_t stays stable by "forgetting" and "writing".)

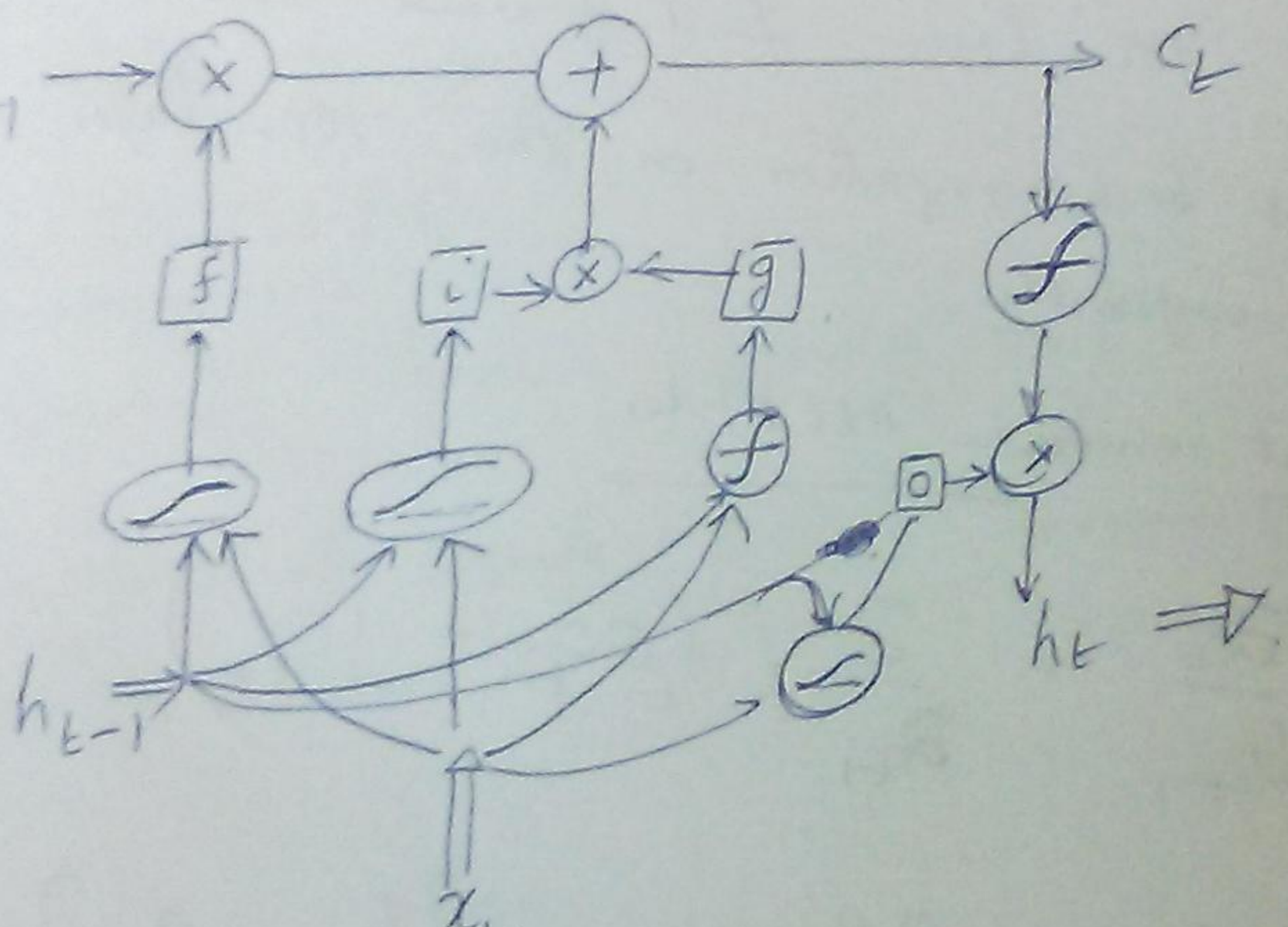
How are Gates calculated?

. They have their own weights and are functions of (h_{t-1}, x_t)

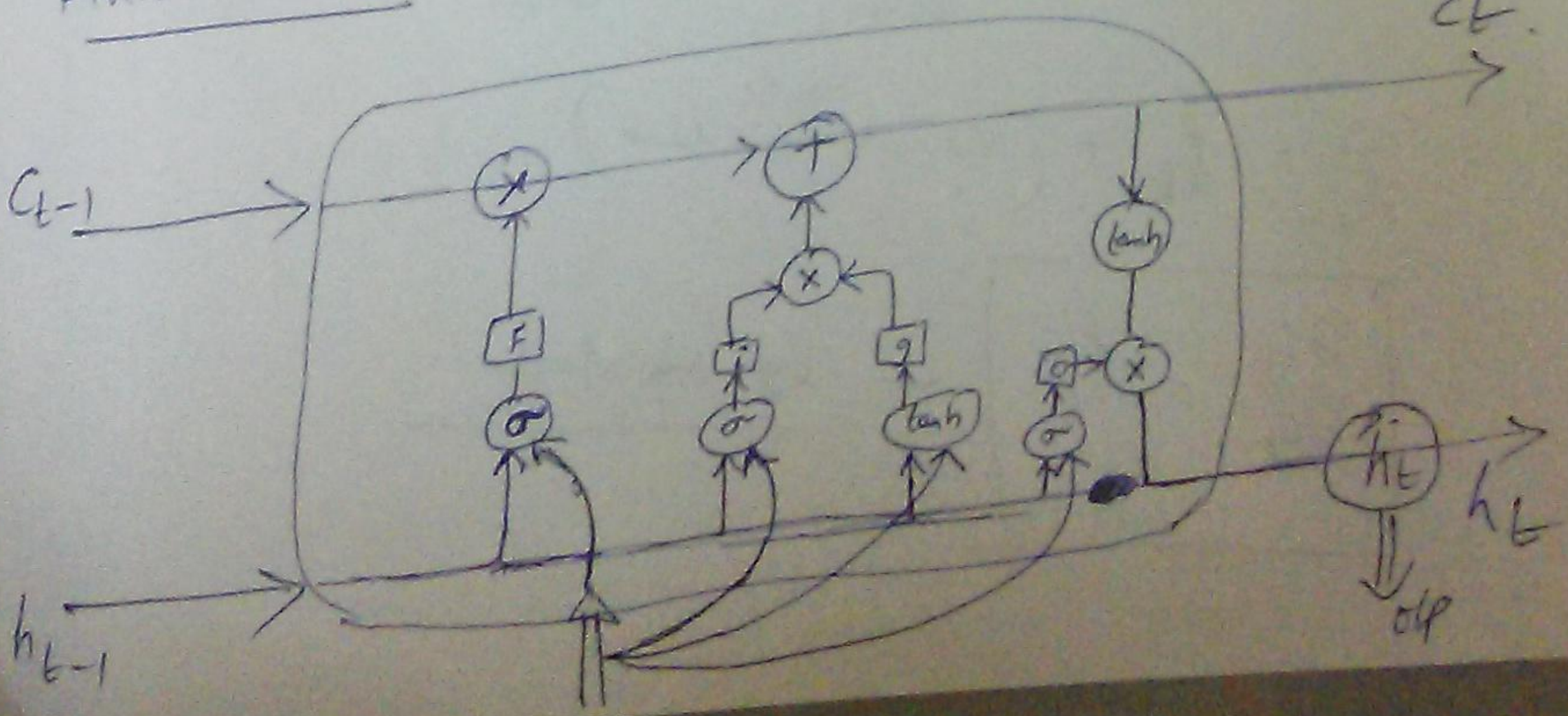
$$f_t = \text{sigmoid}(W_{xf} x_t + W_{hf} h_{t-1})$$
$$g_t = \tanh(W_{xg} x_t + W_{hg} h_{t-1})$$
$$i_t = \text{sigmoid}(W_{xi} x_t + W_{hi} h_{t-1})$$
$$o_t = \text{sigmoid}(W_{xo} x_t + W_{ho} h_{t-1})$$

The whole-thing is end-end differentiable.

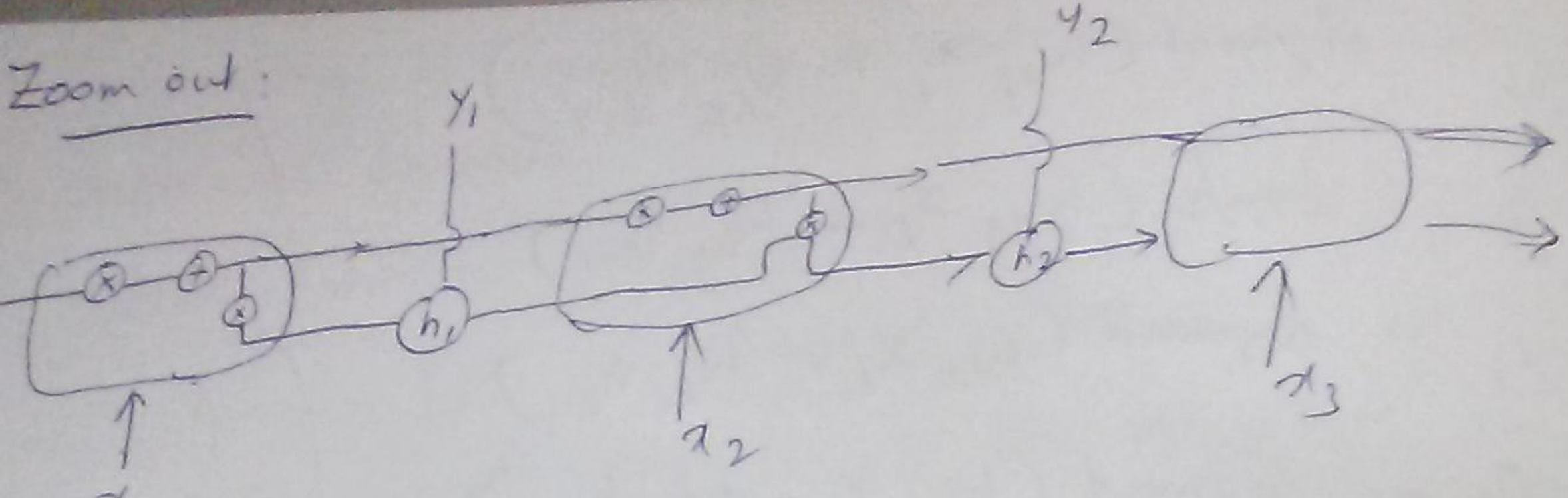
A-near-full-version:



Final Version:



Zoom out:



Training:

Truncated BPTT is used here. Ex: if a 1000 word data is sent in, we effectively have a 100-layer feed-forward n/w.

forward & backpropagation on the 100-layer n/w

n be impossible.

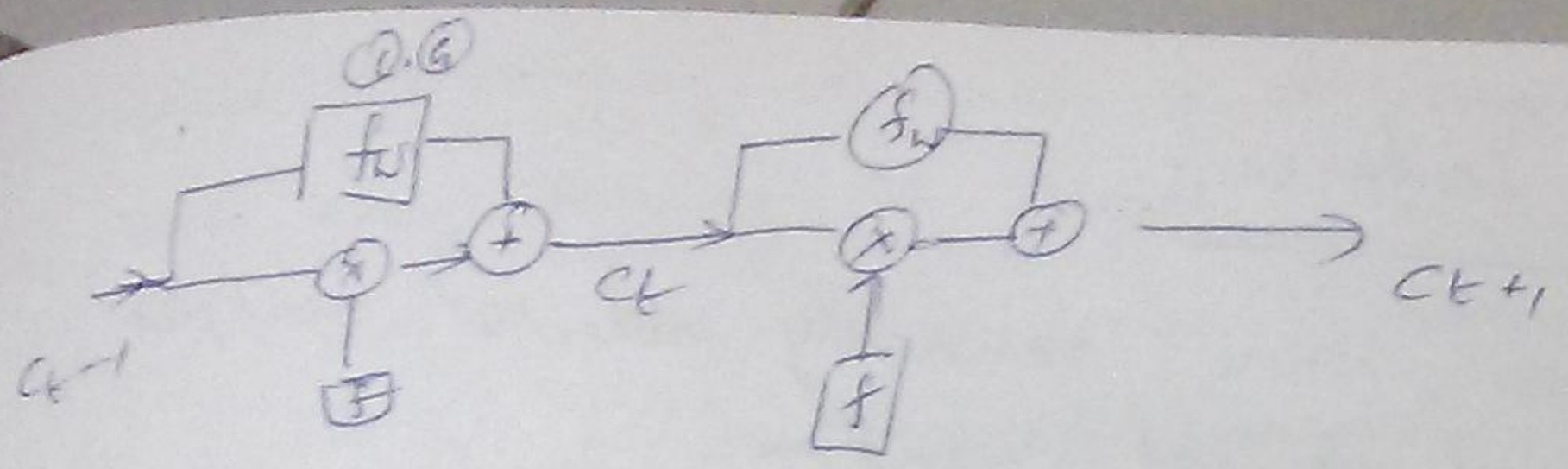
We don't remember ALL states

$$\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial}{\partial c_{t-1}} (f_t \odot c_{t-1} + i_t \odot g_t)$$

$$= f_t \cdot 1 * \frac{\partial f_t \odot}{\partial c_{t-1}} + i_t \frac{\partial g_t}{\partial c_{t-1}} + g_t \frac{\partial i_t}{\partial c_{t-1}}$$

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t + (\text{vanishy gradient}^{\text{may be}})$$

$$\boxed{\frac{\partial c_t}{\partial c_{t-1}} = f_t} \quad \text{— Vvg important}$$



Usually,
the gates bias is set very high, so that we can start with all 1's in "f".

If a gradient vanished, cell-states before that state are not contributing (we need to forget them) which might be good.

Exploding Gradient Pb:

Use Gradient clipping grad = min(grad, threshold)

Exploding gradients quite possible because we have summation of terms.

Dropout technique.

A technique where randomly selected neurons are 'dropped out' at random.

Their contribution to activation of downstream neurons is temporarily removed on the forward pass and any weight updates are not applied to the neurons on the backward pass.

This means that if neurons are randomly dropped during training then other neurons must step in and handle representations required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the nw

This results in the nw being less sensitive to the specific weights of the neurons.

p → dropout rate. Typically set to 0.5

After training, neurons dont get dropped anymore

[neurons are dropped at random at every training instance. This means a neuron dropped for one instance may come back for the next instance.

would a company do well if its employees tossed a coin every morning to decide whether to come to work or not? So the company cannot rely on ONE individual to fill in the coffee machine or manage supplies. The expertise is spread across several people. The company could be more resilient. If one person quit, it couldn't make too much difference.

A different way of understanding this is to realize that with dropout, a different neural n/w is generated at each training step. Since each neuron can either be present or absent, there are 2^N networks possible (N = total no of droppable neurons). So if we went through 10000 training steps, we have effectively 10000 n/ws, each with one training instance. The resulting neural n/w can be seen as an averaging 'ensemble' of these smaller n/ws

One small technical detail:

Since $p = 0.5$, each neuron is connected to twice as many input neurons (on avg) as it was during training. To compensate, we need to multiply each neuron's weights by $\underset{[\text{keep rate}]}{0.5}$ AFTER training. [keep rate $= 1-p$]

[Remember: After training, all neurons are back. The n/w is up to full-strength]

Binary_cross entropy v/s categorical_cross entropy

$$E(y, \hat{y}) = -\sum_t y_t \log(\hat{y}_t) \rightarrow$$ ~~Binary cross entropy~~ log Entropy

For categorical_cross entropy, the targets need to be one-hot vectors, the size of the target vector being equal to no. of classes possible

Bendy_cross_entropy

$$bce(t, o) = -(t \log o + (1-t) \log(1-o))$$

Categorical C.E

$$cce(t, o) = -\sum_x t(x) \log(o(x))$$

Naive Bayes' v/s RNNs

Both are classifiers. They can accomplish multi-class classifications.

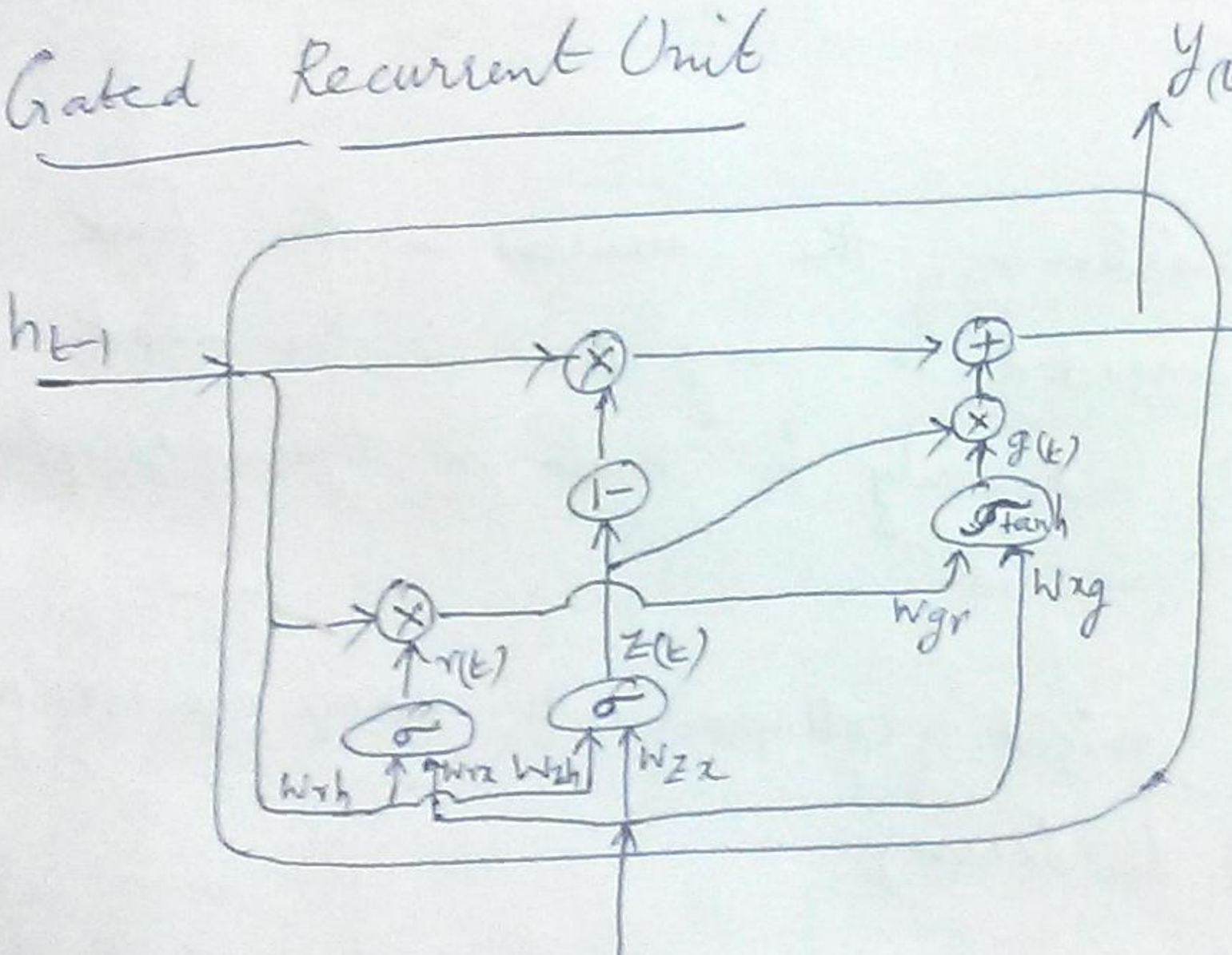
Naive Bayes' needs some preprocessing (RNNs too) in terms of (bag of words + tf-idf → n-grams)) and assumes i.i.d data. If the assumption holds, it's a very powerful classifier.

RNNs take data sequentially and have memory. correlation between words is maintained.

NB → Generative Model - That is, during training, the model tries to see how the data generated. Try to figure out the underlying distribution that produced the training examples. (HMM, GANs)

RNN - Discriminative: It tries to figure out the diff between the positive and negative samples to learn how to classify. (SVM, decision trees too)

Gated Recurrent Unit



(Performance is similar to LSTM)

$$z_t = \sigma(W_{zh} h_{t-1} + W_{zx} \cdot x_{(t)})$$

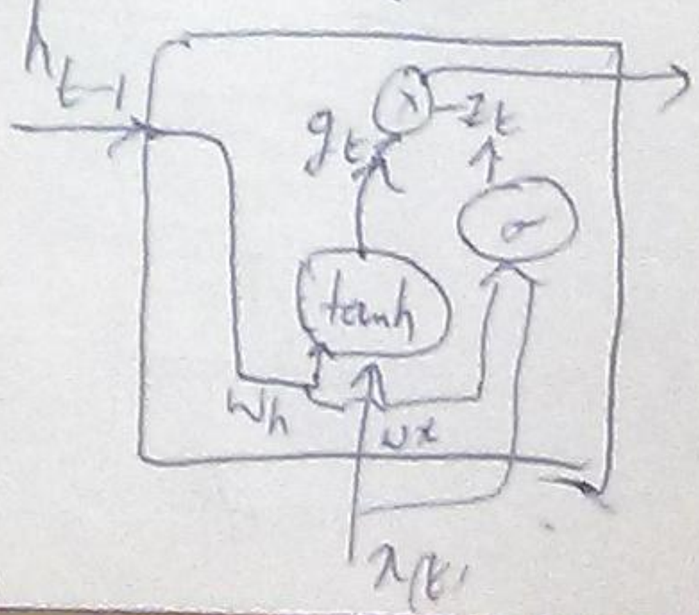
$$r_t = W_{rh} \cdot h_{t-1} + W_{rx} \cdot x_{(t)}$$

$$h_{(t)} = h_{t-1} \otimes (1 - z_t) + z_t \otimes g_{(t)}$$

$$g_t = \tanh(W_{xg} x_t + W_{gr} \cdot (r_t \otimes h_{t-1}))$$

$z_{(t)}$ = Gate that controls memory

or simply



Convolutional Neural networks:

- Deep Feed-forward class of Neural nws.
- Visual Imagery - Computer Vision.
- Based on the operation of the visual cortex.

The neurons in the visual cortex have a small receptive field. These fields might overlap and together they tile the whole visual field.

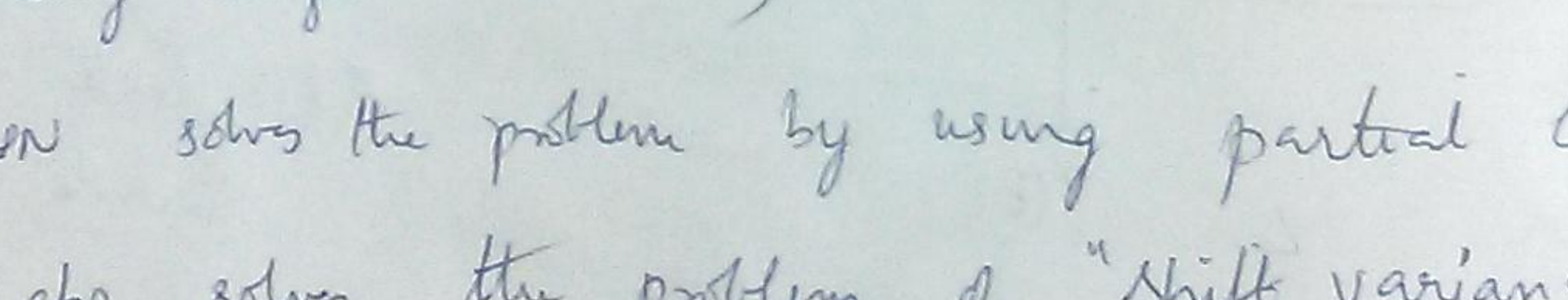
So, the main deviation is the neurons in the first layer are not connected to every pixel in the input image. (like ANNs), but only to pixels in their receptive areas.

Traditional nlws might collapse with image size 100x100. (Too many weights to learn)

CNN solves the problem by using partial connections

CNN also solves the problem of "Shift variance"

Shared-weight architecture.



We use 'filters' → basically the weights of a neuron that need to be learnt. So this effectively means the 'features' will be learnt.

Padding:

- Preserve spatial size. With very deep networks, feature-map size can reduce very quickly. Hence padding helps with 'dimension' & math
- Preserves 'information at the borders'

(Compare to how 'we see')

$$O = (i - K) + 2p + 1$$

O = output dimension
K = filter dimension
p = padding
i = input dimension]

Filters

Parameters - Window size? Stride = ?

filter → also called kernel or feature detector

Output of filter operation on an image part is called 'Activation Map' or 'feature map'

The more the number of filters, the better the classification or learning.

Depth: No of filters used

Zero-padding: if applied → wide convolution
if not applied → narrow convolution.

ReLU: Non-linearity and getting rid of negative values

Spatial pooling: sub sampling or downsampling.

- Reduce dimensionality.

a) Max pooling b) Mean pooling c) Sum pooling.

↳ window-size

↳ Stride. Usually equal to window dimension

- Reduces input Representations (manageable)
- Reduces no of parameters and computations, reducing overfitting.
- Makes the n/w invariant to minor transformations (we take max or mean which reduces the effects)

Fully connected Layer:

Traditional Multi-layer Perceptron (MLP) that uses a softmax activation unit at the output. This is a fully connected n/w.

The o/p from the pooling layer is high-dimensional features.

Using Existing Models

Training a CNN from scratch can be extremely time-consuming,

we can leverage existing trained Models.

Popular Models are

1) VGG -16/19 - Oxford
2) ResNet 50 - Microsoft
3) InceptionV3 - Google
4) LeNet
5) AlexNet etc

We can either use the architecture and train the model on our data from scratch

or we can freeze many layers (initial layers, which learn low-level features like curves, edges etc) and train on subsequent layers only.

(Refer the document (softcopy) for more detail)

GANs:

Please refer last Semester's notes for now !!

A Quick Note on Vanishing Gradients problem

The deeper a network is, the greater the chance of Vanishing gradients due to numerous multiplications.

Some common Solutions

~~Early stop~~.

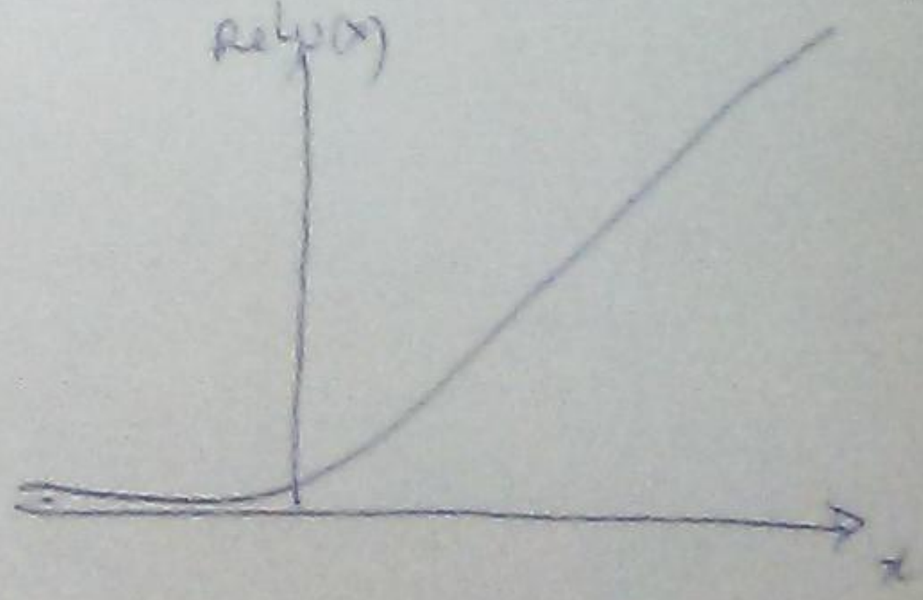
1. Use non-saturating activation functions.
(With Sigmoids, we have the max derivative as $\sigma(1-\sigma)$ which is $1/4$. Hence the max derivative we can get for multiplication is 0.25
With tanh, it is slightly better. we can set a max value of '1'.)

ReLUs are better.

We can consider

$$Relu(x) = \log_e(1+e^x)$$

This ensures the following graph



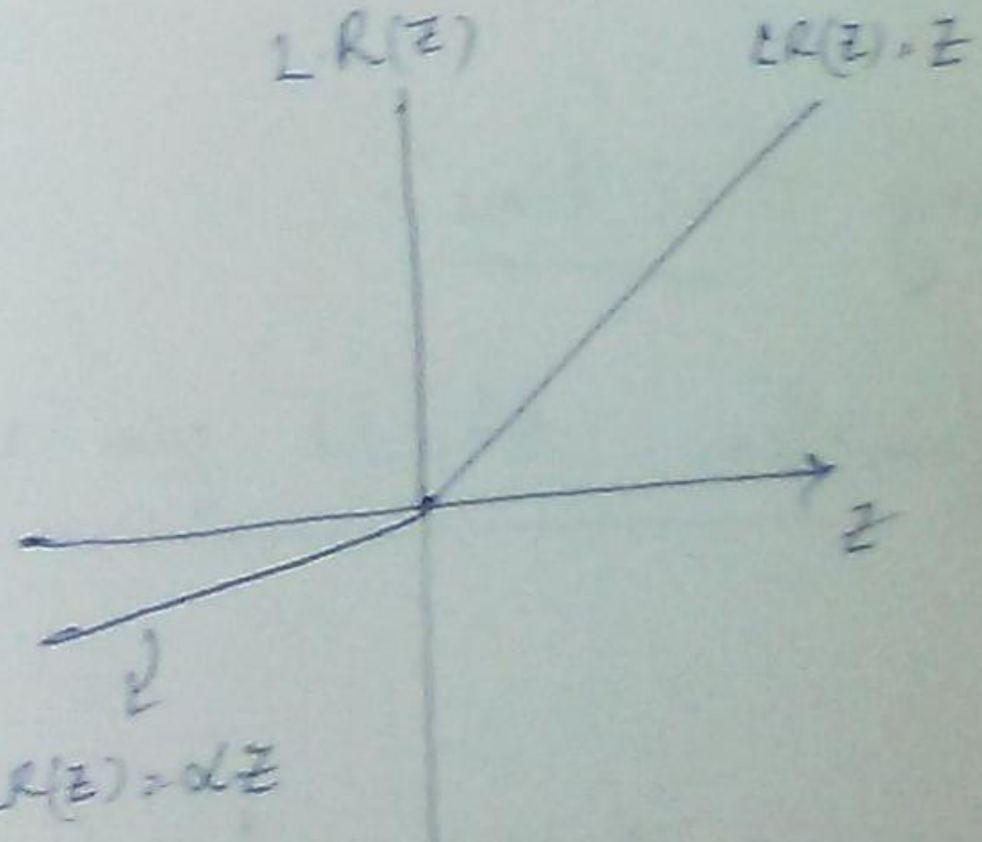
for positive values of 'x' derivative is significant

But for negative values of z, derivative might reach zero, and so we have no weight updates. This is called dying ReLU problem.

One of the concerned neurons. Learning ceases.

2) Therefore, we can use a modified ReLU, called Leaky ReLU.

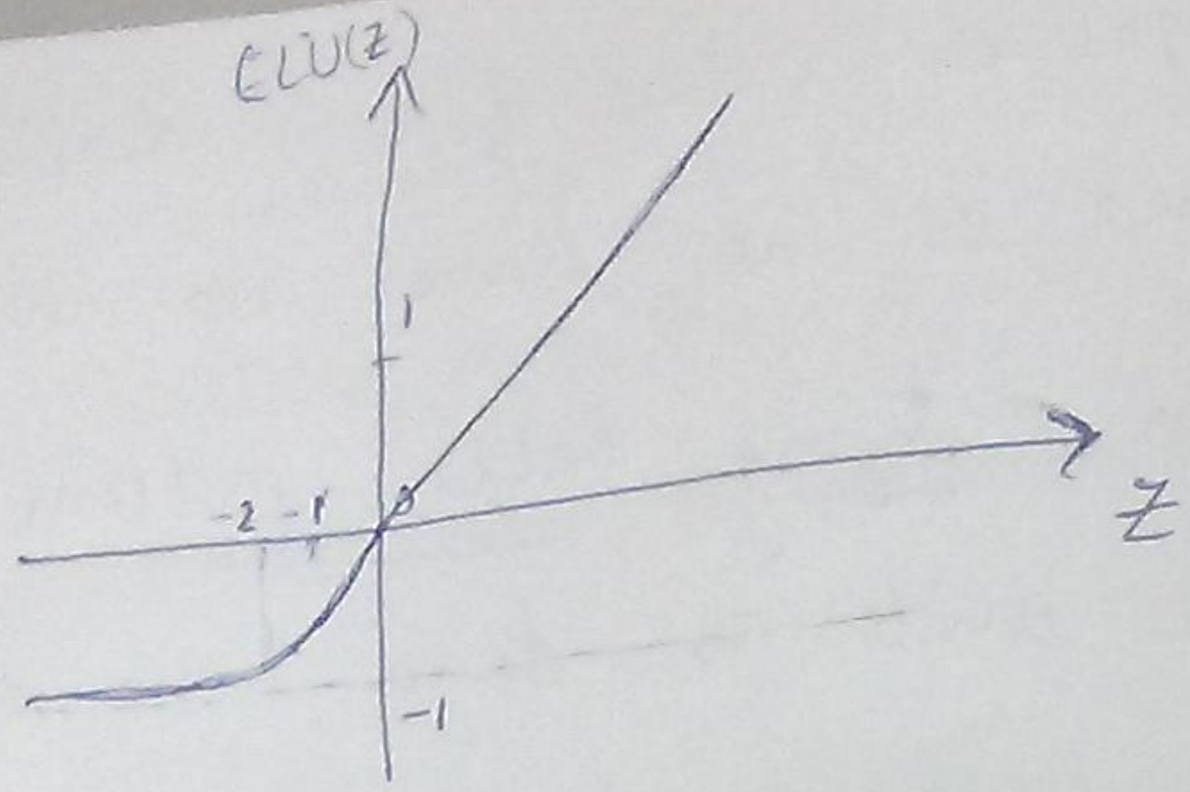
L-ReLU (z) = max (αz, z). The parameter α decides how much the function leaks. It is the slope of the function for $z < 0$. (Usually set to 0.01 or lesser)



The small slope ensures the ReLU never dies.

3) A novel variation of the ReLU is the ELU also called Exponential Linear Unit. This performs best. (training time reduces and performance increases)

ELU



this graph is for $\alpha = 1$

[not to scale]

$$ELU_\alpha(z) = \begin{cases} \alpha(e^z - 1), & z < 0 \\ z & z \geq 0 \end{cases}$$

This ensures that for negative values of z, we have a constant o/p from the ELU. This ensures we don't have the vanishing gradient issue and also don't have the 'dying ReLU' problem

Drawback:

can be slower during 'test time'.

Which activation function should you use?

Generally speaking,

ELU > Leaky ReLU > ReLU > tanh > σ

For Exploding Gradients, use Gradient clipping.

[I will cover the concept of Batch Normalization after the Test]

SKD

Convolutional Neural Networks - CNN

chandra's DC TE

convolution - winding together

widely popular with image classification Traditional image processing methods are still popular.

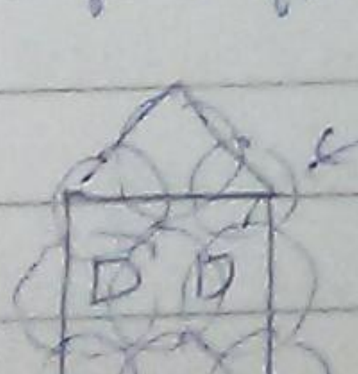
The problem:

Identifying a classifying an image we want the computer to differentiate between all images and learn the unique features that make a dog a dog and a cat a cat.

The computer should be able to perform image-classification by looking for low level features like curves or edges and then building upto more abstract concepts through a series of convolutional layers.

Needless to say, the network will be humungous.

The biological inspiration is the visual cortex where each neuron has a limited "local receptive field". This means they react to visual stimuli located only within a limited region of the visual field. Different visual fields of different neurons might overlap



← the circles represent receptive fields of each neuron. One circle per neuron.

Example - Task is to identify forward / or backward slash \ or X.

The Convolutional NN contains mainly of 3 main components

a) The convolutional layer - Consists of filters or neurons or kernels which react to a limited visual field.

b) The Max pool layer (To reduce computation)

c) The fully connected layer - final classification.

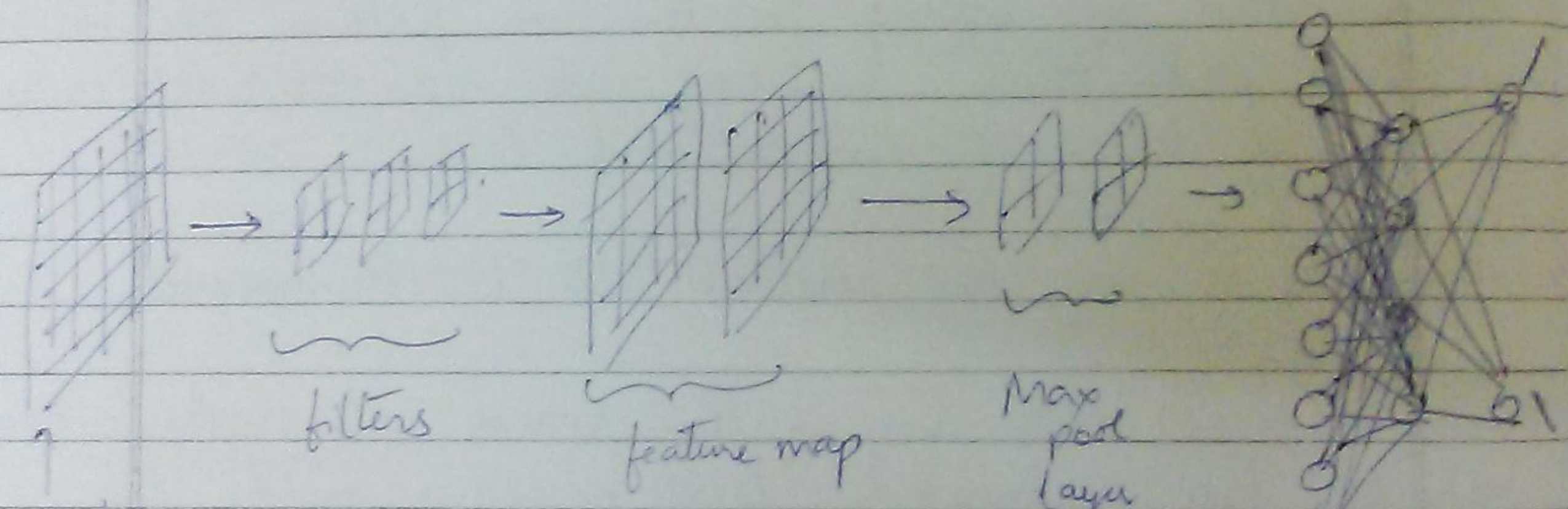


fig 1

CNNs are robust to mouse like translation, scaling, rotation or weighting

The idea is to match tiny parts of the image with a reference we have and then building up to full object detection.

chandra's

Filter: A neuron with weights. The no of inputs to the neuron = the size of the filter.

Ex: a 2×2 filter means a neuron with 4 weights and 4 inputs. The values of the filter will be the weights of the neuron. Each neuron will observe an area = the size of the filter. That is, it observes a 2×2 area in the image.

There is a 'stride' factor (stride) which is usually set to 1. This is the no of pixels by which a filter moves over the input image so that the next neuron can react.

(Each filter is super-imposed on an area of the input image = 2×2. The o/p of the neuron observing this part of the image is $w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$ (perceptron)

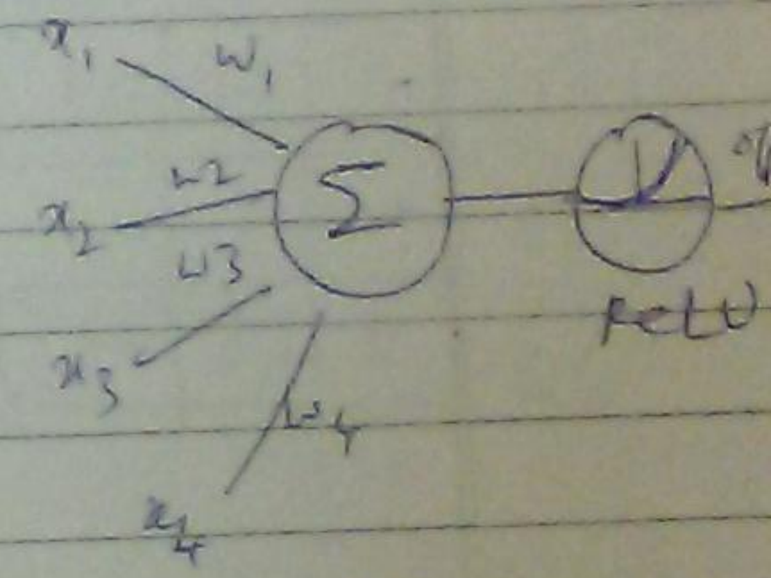
where

w_1	w_2
w_3	w_4

Filter

x_1	x_2
x_3	x_4

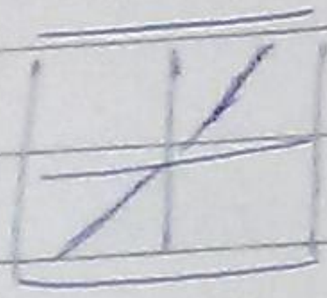
area of i/p image in visual field of perceptron



Each perceptron uses the 'ReLU' as activation unit.

The 'filters' are small parts of the actual (correct, classified) image which give high o/p when the i/p image part on which it is super-imposed, matches the filter.

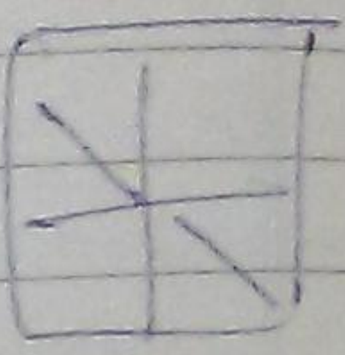
Example - A filter of size 2×2 representing a '/' can be



$\Rightarrow$

-1	1
1	-1

Similarly a filter representing a \ is



which is

1	-1
-1	1

Now if we present this to an i/p image which is actually a forward /, (i/p image is 3×3 say)

	a	b	c
1	-1	-1	1
2	-1	1	-1
3	1	-1	-1

The filter starts at top left corner and slides rightwards, one pixel at a time.

(Diff neurons - 4 to be precise are waiting to operate on the i/p area presented to them

1st neuron covers 1a 2b

2nd neuron covers 1b 2c

3rd neuron covers 2a 3b

4th neuron covers 2b 3c

The o/p of the neurons (assuming ReLU) are

(Applying the forward filter 1 $\Rightarrow$ $\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}$

1st neuron $\rightarrow$ $-1\times-1 + 1\times-1 + 1\times-1 + -1\times1 = -2$

2nd neuron $\rightarrow$ $-1\times-1 + 1\times1 + -1\times-1 + 1\times1 = 4$

3rd neuron $\rightarrow$ $-1\times-1 + 1\times1 + 1\times1 + -1\times-1 = 4$

4th neuron $\rightarrow$ $-1\times1 + 1\times-1 + 1\times-1 + -1\times-1 = -2$

So the feature map (for this filter) is

$\begin{bmatrix} -2 & 4 \\ 4 & -2 \end{bmatrix} = \begin{bmatrix} 0 & 4 \\ 4 & 0 \end{bmatrix}$ (See Fig 1 for reference)

(Likewise there are other feature maps for other filters,

Now to reduce computation, we will feed this to a max pool layer. The Max pool layer also has a window size and stride (usually equal to window size)

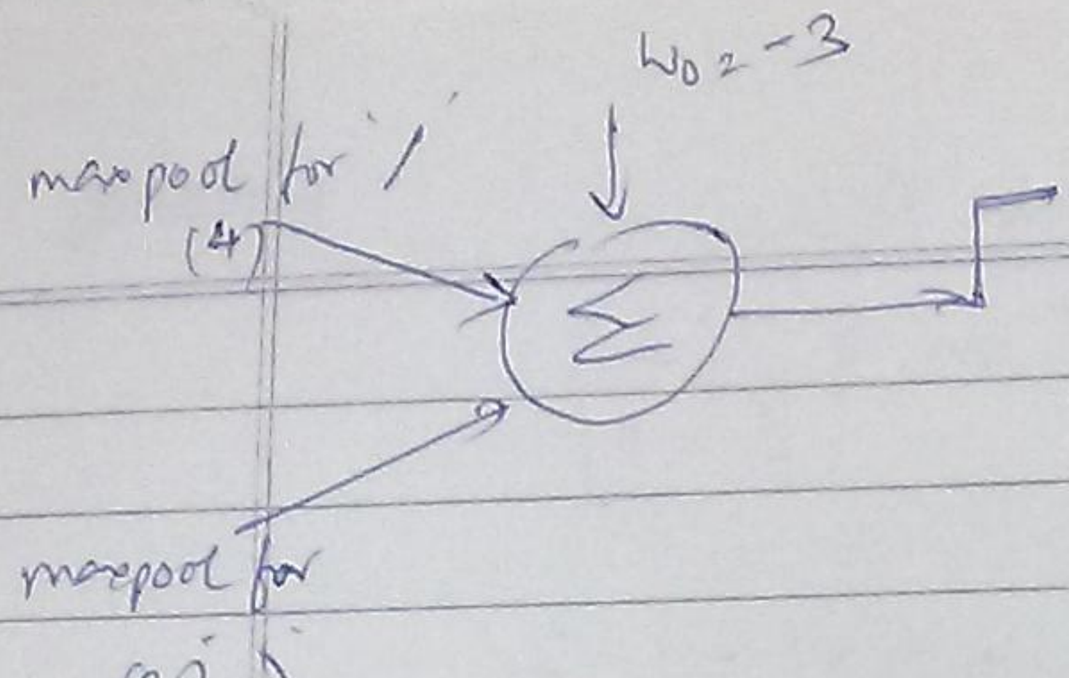
Here let's consider 2×2 :- The Max pool layer simply takes the max value in the window.

So for the above feature map, Max pool outputs 4

(The second filter for the backslash) would o/p the following feature map

$\rightarrow \begin{bmatrix} 2 & -4 \\ -4 & 2 \end{bmatrix} \xRightarrow{\text{ReLU}} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$. Max pool will o/p 2 here.

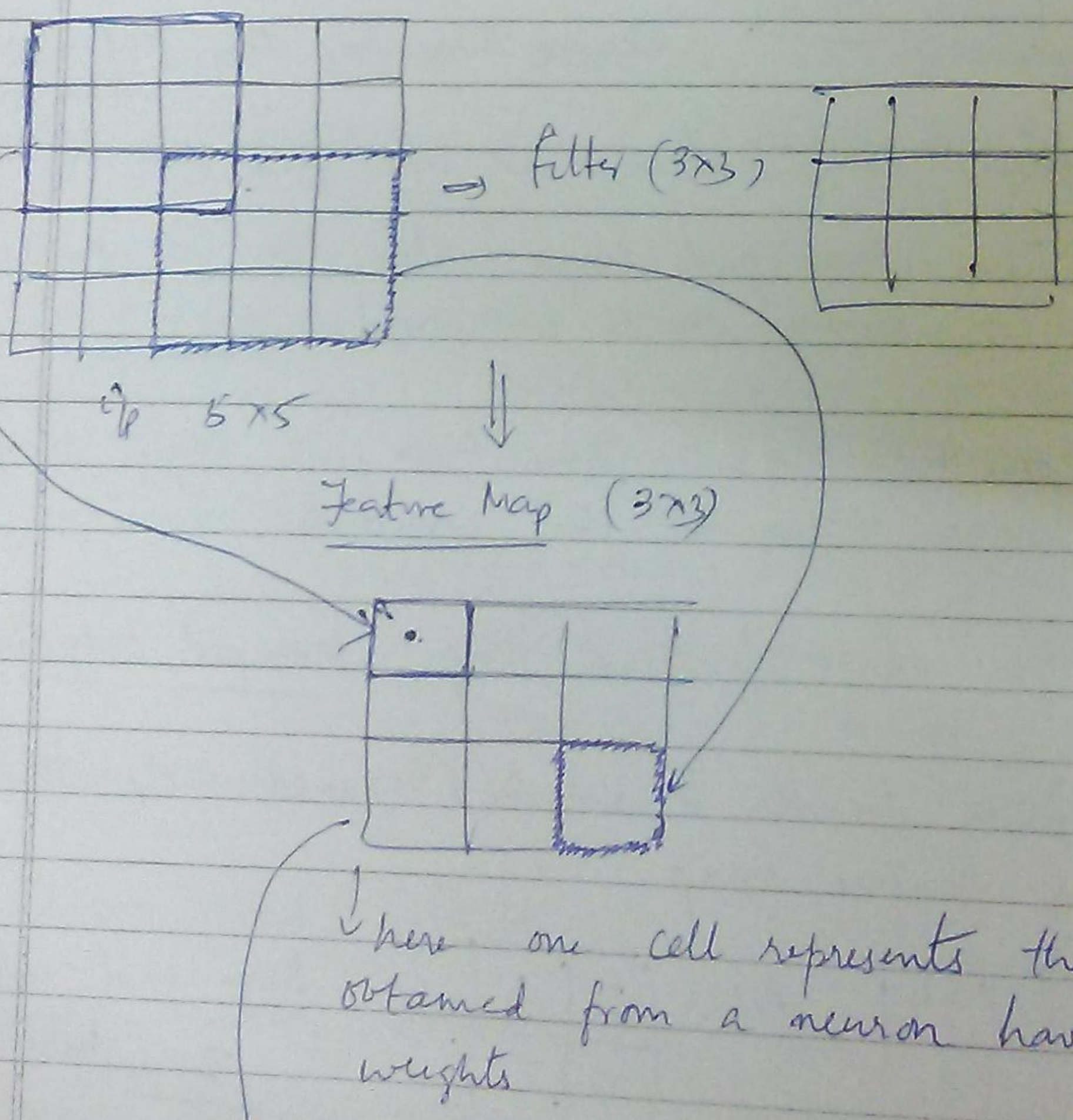
These 2 Max pool o/ps can now be inputs to a fully connected Neural n/w. In this case we only need a single perceptron with $W_0 = -3$.



If the neuron triggers it is a forward slash. Else a backward slash (1 or 0)

If the same thing is imagined on a bigger scale, we have a real CNN

Training happens by the well-known Backpropagation (The filters are also learnt)



here one cell represents the value obtained from a neuron having nine weights

Max pool (2×2), stride = 2 (placed on the feature map)

⇒ 2×2 o/p

Now - if there are 2 filters, then there will be chandra's
2 o/p from the max pool layer each 2x2 size - So 8 o/ps.

These 8 o/ps are fed to a fully connected neural n/w having 8 i/p nodes, one or more hidden layers and an o/p layer having as many nodes as classifications required