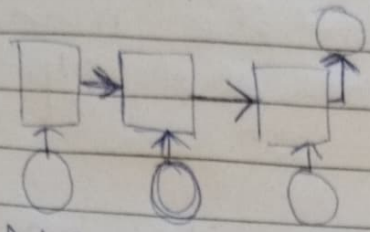
Recurrent N N, Transformers and Attention

Sequence model applications

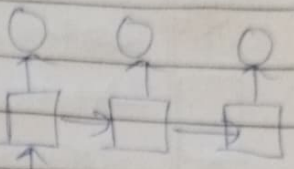
1) Many to one



Many to one

Eg sentiment classification

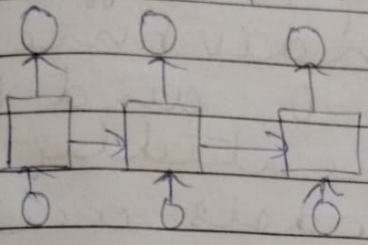
2)



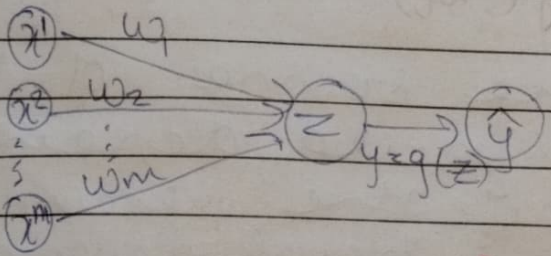
one to many

Eg - Img captioning

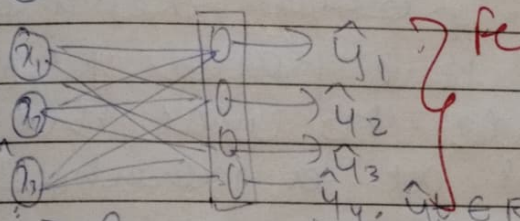
3) Many to Many



Neurons with recurrence



The perceptron



Feed forward N.N

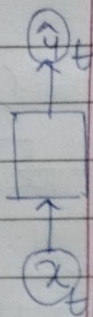
$x^t \in R^m$

$y^t \in R^n$

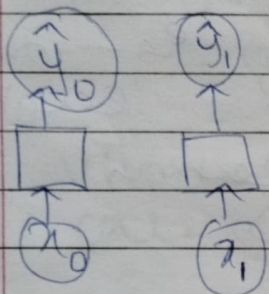
→ Even if we stack perceptrons to form feed forward NN we still dont have notion of temporal info or sequential processing

Handling Indiv. time steps

If we collapse the middle layers of stacked perceptrons into a single unit



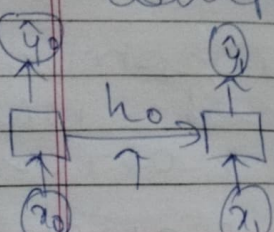
→ t denotes a single time step
→ input at single time step generates single O/p using NN



→ What if we do the operation multiple times to handle inputs fed in ~~the~~ corresponding to diff times
→ We hv an indiv time step starting with 0 and keep doing this repeatedly
→ All these models were copies of each other

$$\hat{y}_t = f(x_t)$$

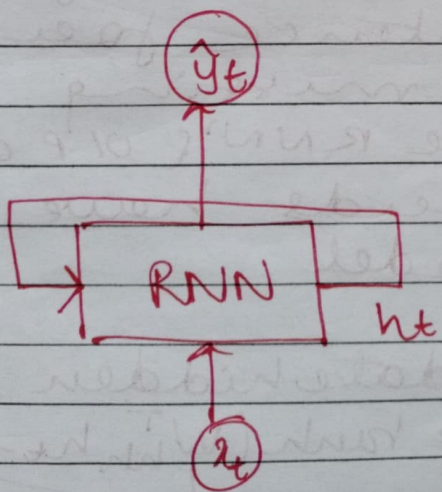
→ but here some of the later o/p would depend on some of the prev inputs if it is temporal dependance
→ If we link the prev N.N with next one also to link the networks computations at a particular time step to previous history



$h_0 \rightarrow h_1 \ldots$ via a recurrance relation
→ What the network computes at a particular time is passed to the later time steps

PAGE No.
DATE 16/4

→ we define this with variable h (internal state) which we can think of as a memory term that's maintained by network & neurons & this state that's passed on from network to network as we process the seq- into
→ Networks O/P is not only a function of input x but also h which captions notion of memory that's being computed by the network & passed on over time $y_t = f(x_t, h_{t-1})$
→ We can also depict this relationship according to a loop
→ This computation to this internal state variable h of T is being iteratively updated over time and fed back into neurons computation in the recurrence relation



RNN is maintaining the state & updating the state at each time step

* Apply recurrence relation at every time step to process a sequence

old state $h_t = f_W(x_t, h_{t-1})$ function with weights W

PAGE No.
DATE / /

Note - Same function and set of parameters are used at every time step

RNN Intuition:-

```
my_rnn = RNN()                          define RNN model
hidden_state = [0,0,0,0]                initialize hidden state
sentence = ["I", "love", "recurrent",
"neural"]
for word in sentence:                   } loop through each word
    prediction, hidden_state =          }
    my_rnn(word, hidden_state)          }
next_word_prediction = prediction
```

Suppose we want to predict the next word of the sentence which is our temporal sequence

we feed the word to the RNN along with prev hidden state

→ This generates pred. for next word & updates the RNN's hidden state in turn

→ Finally our pred. for final word in sentence for word that we are missing is simply the RNN's O/P after all prior words have been fed in the model

Input vector x_t

Update hidden state

$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$

we have 2 separate weight matrices that the network is learning

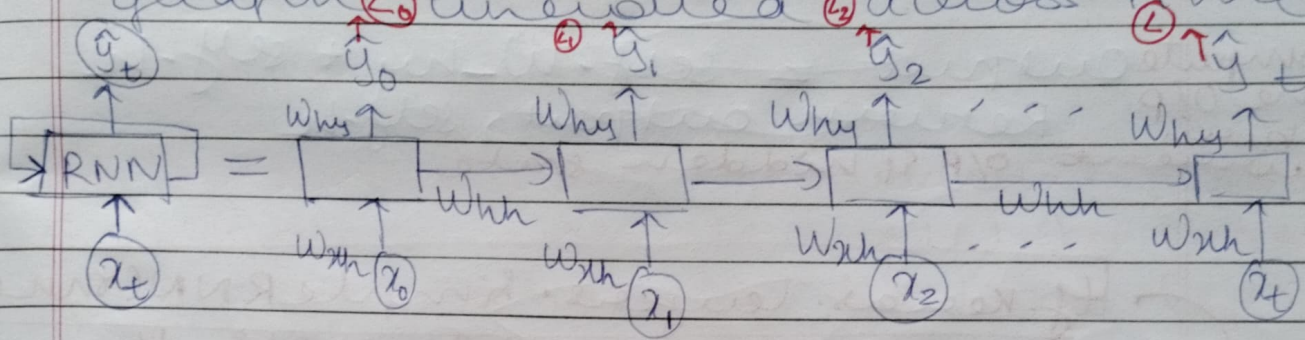
PAGE No.
DATE / /

Output vector = $W_{hy} h_t$

Seperate wt matrix

Computational Graph Across time

→ Represent as computational graph unrolled across time



→ prediction at an indiv time step will simply amount to a computed loss at that particular time step

→ Now we can compare those predictions time step by time step to the true label and generate a loss value for those timestamps and finally get total loss by taking indiv. losses & summing & defining total loss for particular I/P to RNN

RNNs from Scratch

```
class myRNNCell(tf.keras.layers.Layer)
    def __init__(self, rnn_units, input_dim,
    output_dim):
        super(MyRNNCell, self).__init__()
        self.W_xh = self.add_weight([rnn_units,
        input_dim])
        self.W_hh = self.add_weight([rnn_units, rnn_units])
        self.W_hy = self.add_weight([output_dim,
                                     rnn_units])
```

Initialize wt matrices

PAGE No.
DATE

Initialize hidden states to zeros

```
self.h = tf.zeros([rnn_units, 1])
```

Update the hidden state

```
def call(self, x):
    self.h = tf.math.tanh( self.W_hh *
    self.h + self.W_xh * x )
```

Compute the O/P

```
    output = self.W_hy * self.h
```

return current O/P & hidden state

```
    return output, self.h
```

`tf.keras.layers.SimpleRNN(rnn_units)` tensorflow abstracts the operation of defining RNN so we can define simple RNN by this call

Sequence modelling: Design Criteria

- To model sequences, we need to
- Handle variable length sequences
- Track long-term dependencies
- Maintain info about order
- Share parameters across sequence ie same wts used in diff time steps shld still give a meaningful prediction

Eg diff length sentences

Long dependencies far away from each other

diff words in diff order

no diff meaning

Representing Language to NN

→ NN cannot interpret words and requires numerical inputs

Encoding language for NN

Embedding: transforms indexes into vectors of fixed size

transforms indices or something that can be rep. as an index into a numerical vector of a given size

Embedding can be used for words as:-

1) Vocabulary (all words) 2) Indexing (word to index) 3) One hot embedding "cat" = [0, 1, 0, 0, ...] 1 → 1th index

Instance of the word corresponds to a 1 at its index

Embedding / index to fixed size vector

This is a very sparse method simply based on count index

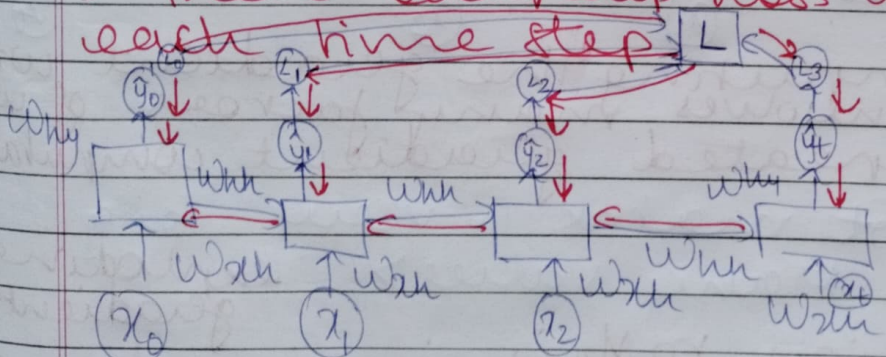
⇒ Alternatively, commonly what is done is to use a NN to learn in embedding. We can learn a NN that captures some meaning in the I/P data & maps related words/inputs closer in this embedding

~~# Handle Variable sequence length~~

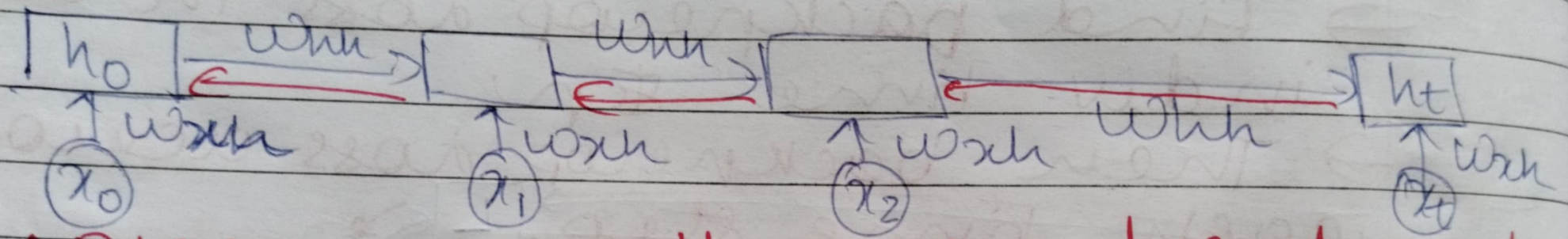
Backpropagation through time (BPTT)

⇒ Find backprop loss in each indiv. time step

⇒ Then backprop loss across each time step



⇒ This algo involves many repeated computations & multiplications of wt matrices



computing the gradient wrt h0 involves many factors of W_{hh} + repeated gradient computation

If wt matrix is very big
ie Many values > 1: exploding gradients

Solution to this is gradient clipping which scales back big gradients to construct them in a more restricted way

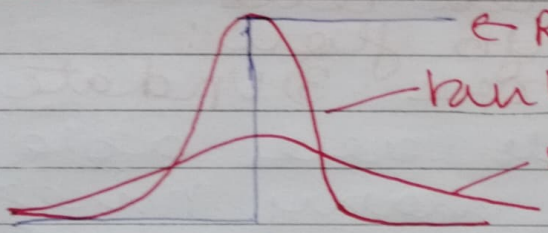
Conversely the wt matrices could be very very small and we end up with vanishing gradients

Many values < 1:
vanishing gradients

Tools to mitigate vanishing grad
1) Activation func 2) wt initialization 3) Choose architecture

Why are vanishing gradients a problem? (Problem of long term dependencies)
Multiply many small numbers together → Errors due to further back time steps have smaller & smaller gradients → Bias parameters to capture short-term dependencies → RNN becomes unable to connect the dots & establish long term dependencies

Trick 1 Activation functions



Using ReLU prevents f' from shrinking gradients when $x > 0$ as its derivative is 1

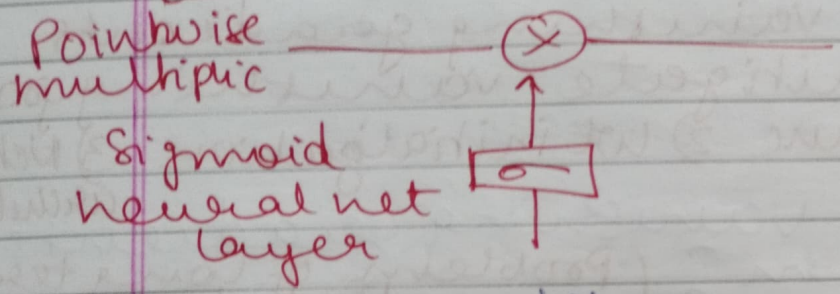
Trick 2 Parameter initialization
→ Initialize weights to identity matrix
Initialize biases to 0

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

This helps prevent wts from shrinking to 0

Trick 3 Gated cells — Most robust solution

idea: To use gates to selectively add or remove info within each recurrent unit with



→ We can filter out whats not important while maintaining whats imp.

→ The most popular kind of Recurrent unit that achieves this gated is LSTM (Long short term memory) — networks rely on gated cell to track info throughout many time steps

LSTM's These cells control info flow:

1) Forget 2) Store 3) Update 4) O/P

→ LSTM cells are able to track info thro many timesteps

`tf.keras.layers.LSTM(num_units)`

Key concepts 1) Maintaining a cell state — just like a std RNN & that cell state is independant from what is directly outputted

2) use gates to control flow of info (for updating cell states)

• Forget gate gets rid of irrelevant info

- Store relevant info from current I/P
- Selectively update cell state
- Output gate returns a filtered version of the cell

3) We can train the LSTM using backprop thro time algo. Mathematics of how LSTM is defined allows for uninterrupted flow of gradients which completely eliminates vanishing grad. problem

RNN applications & limitations

Eg 1 – Sentiment classification
I/P → Seq. of words O/P – Prob of having the sentiment

loss = tf.nn.softmax_cross_entropy_with_logits(y, predicted)

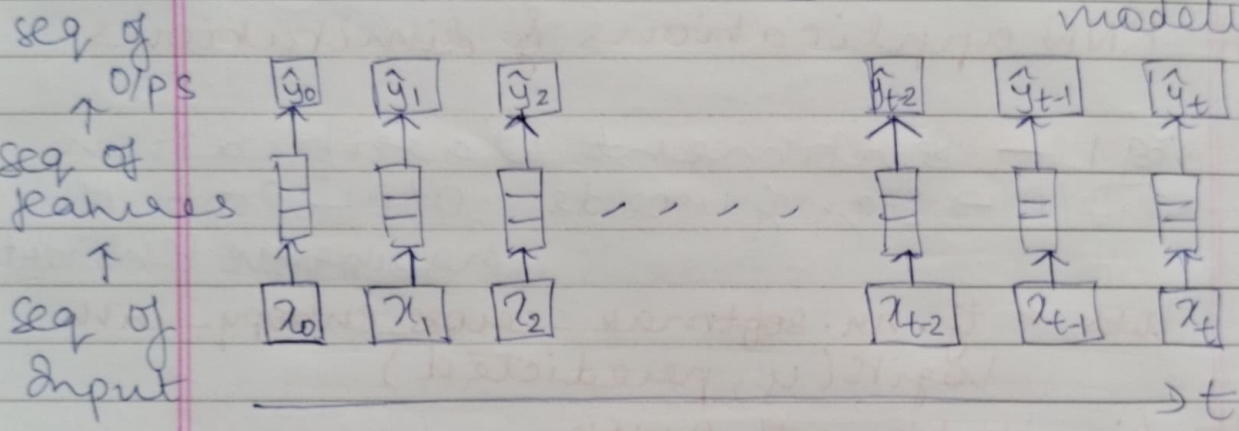
Limitations of RNNs

1) Encoding bottleneck – RNN require that seq. info is taken in & processed time step by time step which imposes an encoding bottleneck – we r trying to encode a lot of content into a single O/P that may be just at the very last time step. We cant ensure that all that info leading up to that time step was properly memorized & encoded

2) RNN's can be quite slow, there is no easy way to parallelize that computation
3) Capacity of RNN & LSTM is not that long so we can't really handle data of tons of thousands or even beyond that effectively to learn the complete info & patterns present within a rich data src

Goal of sequence modelling

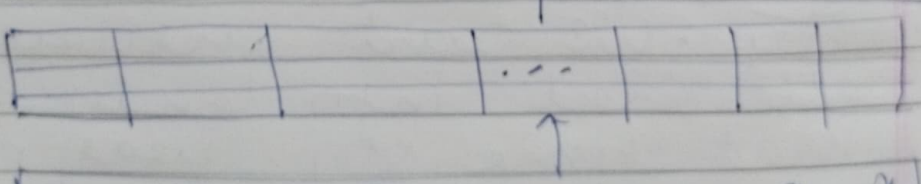
RNN - recurrence to model sequence modelling



Desired capabilities - 1) continuous stream 2) Parallelization 3) Long memory

Can we eliminate need for recurrence?

1) O/P $\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$ ... $\hat{y}_{t-2}$ $\hat{y}_{t-1}$ $\hat{y}_t$



feature vector / concatenate feature vec of all time steps

I/p x_0 x_1 x_2 ... x_{t-2} x_{t-1} x_t

disadv - not scalable, no order, no long memory

2) Attention - foundational mechanism of transformer architecture

*Intuition behind self-attention - attention to the most imp. parts of an input

1) Identify which parts to attend to
2) Extract features with high attention

STEP 1 compute attention mask : how similar is each key to desired query

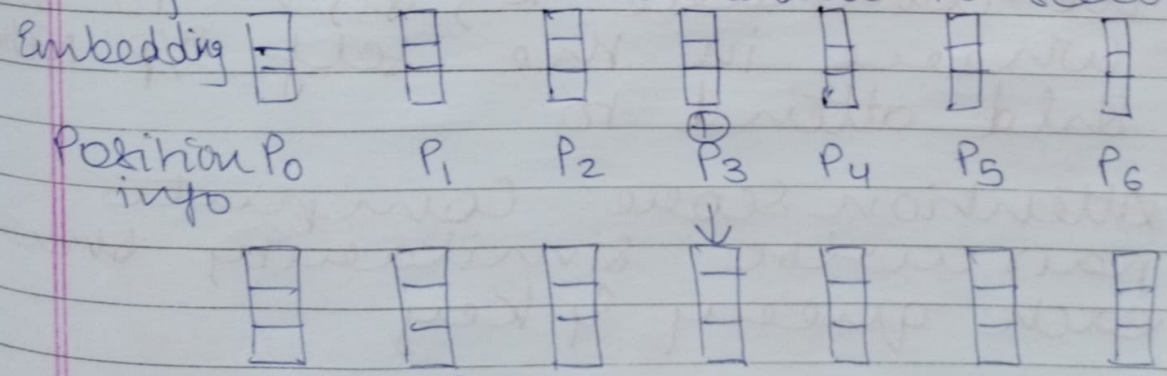
STEP 2 : Extracting values based on attention : Return the values of highest attention

Learning self attention with NN

Goal - identify & attend to imp features in input

1) Encode position info

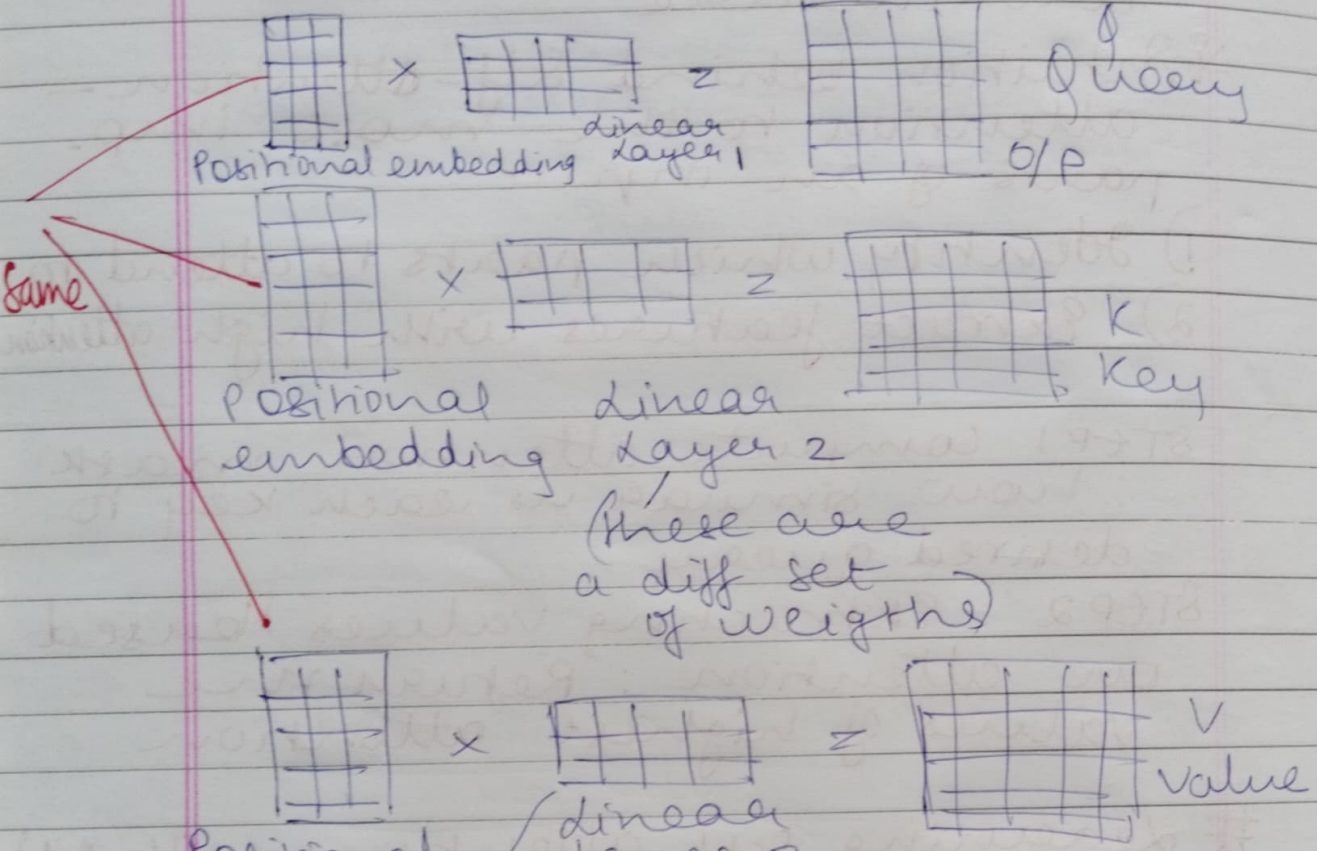
Eg He tossed the tennis ball to serve



Position - aware encoding

* Data is fed in all at once need to encode position info to understand order

2) Extract queries, keys, value for each

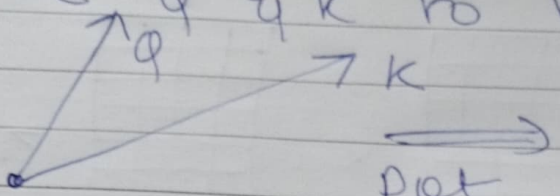


3) Compute attention weighting
We compare K, Q, V to see where in the self O/P Network shld attend to

Attention score: Compute pairwise similarity btw each query & key

To compute similarity btw 2 sets of features:

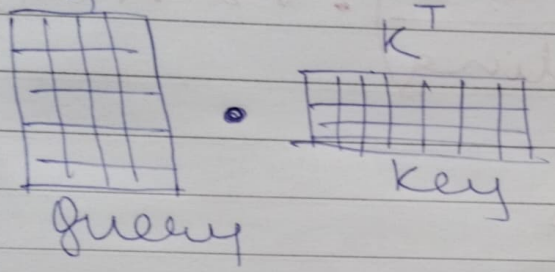
⇒ assume Q & K to be vectors



$\xrightarrow{\text{Dot Product \& Scale}}$ $\frac{Q \cdot K^T}{\text{scaling}}$

Similarity metric also known as "cosine similarity"

This also applies to matrices

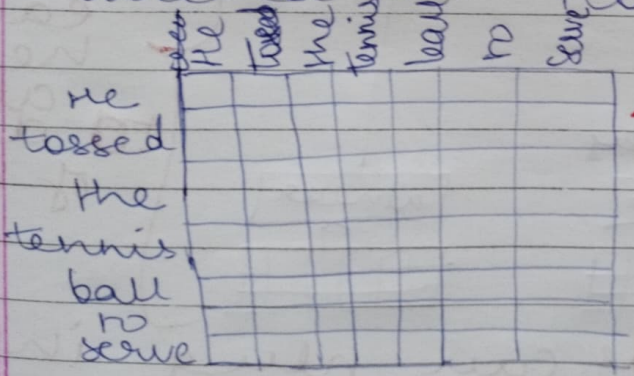


Dot Product ⇒ Scale

$\frac{Q \cdot K^T}{\text{scaling}}$

Captures how similar these vectors are and whether or not they point in the same direction

Attention weighting: where to attend to! How similar is K to Q



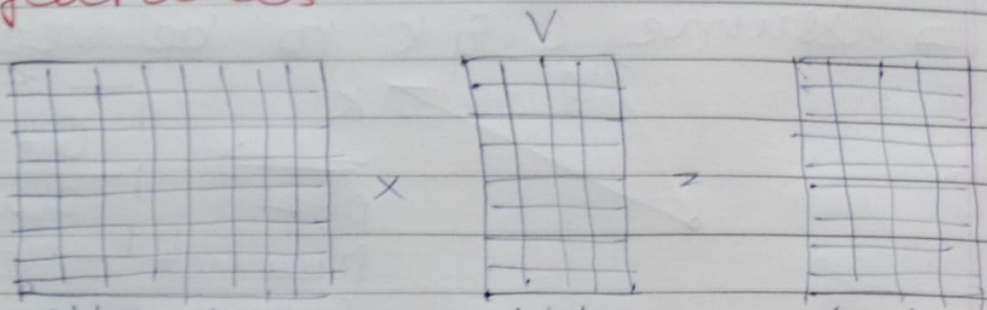
attention weighting

$\text{Softmax}\left(\frac{Q \cdot K^T}{\text{scaling}}\right)$

attention weighting

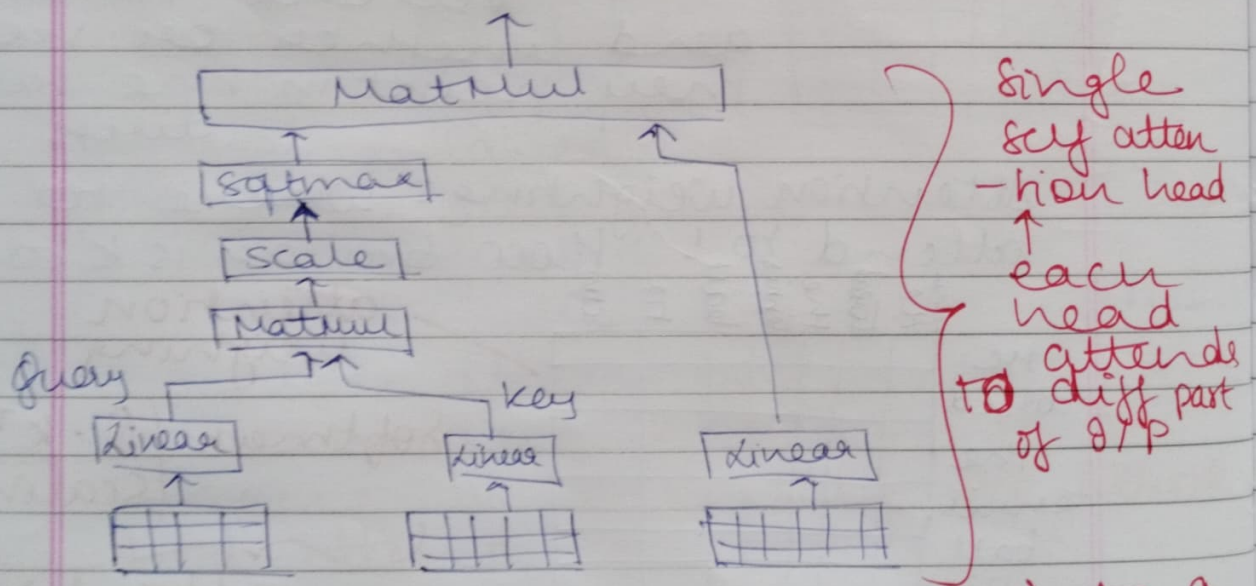
words related to each other in the sequence shld have high attention wts

4) Extract features with high attention: Self-attend to extract features



$$\text{Softmax}\left(\frac{Q \cdot K^T}{\text{Scaling}}\right) \cdot V = A(Q, K, V)$$

Summary



These heads can plug into a larger network