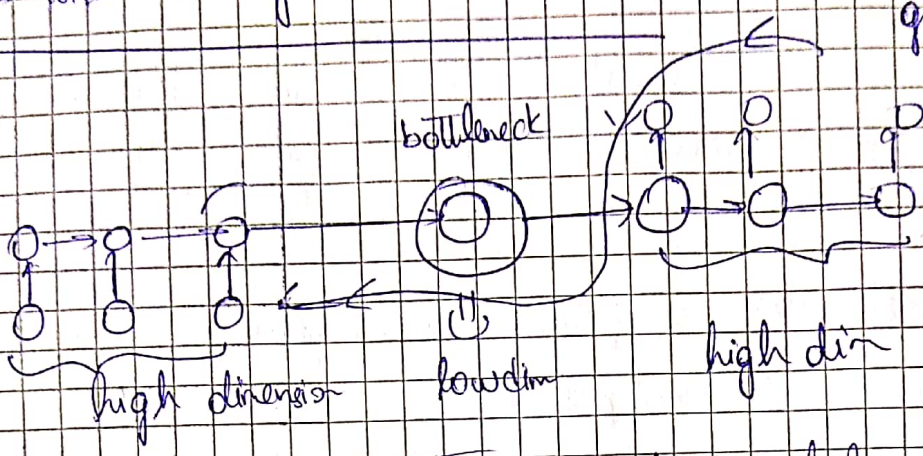
7/06/20

Attention-1

Problems with existing encoder-decoder (Need for attention)



* Difficult to capture long term dependencies of long sentences as everything is mapped to one 512/1024 low dim vector in bottleneck

→ Both long & short sentences have same no of nodes at bottleneck

→ Gradient should pass from o/p → bottleneck → i/p hence difficult to learn (decode, encode)

Soln: Why can decoder look at bottleneck for all the time

This attn

→ This allows for decoder to focus on relevant hidden state at that particular timestep (No bottleneck)

→ Last bottleneck is connected to all encoders with some weights (attention) Hence allows for creation of shortcuts dynamically making gradient update more localized to some encoders

For example: The quick brown fox jumped over a lazy do

P1 P2 P3 P4

(shorter dependency)

→ When decoding P_1 part of sentence P_1 must have higher gradient update & maybe some for P_2, but P_3/P_4 should recieve less prominence

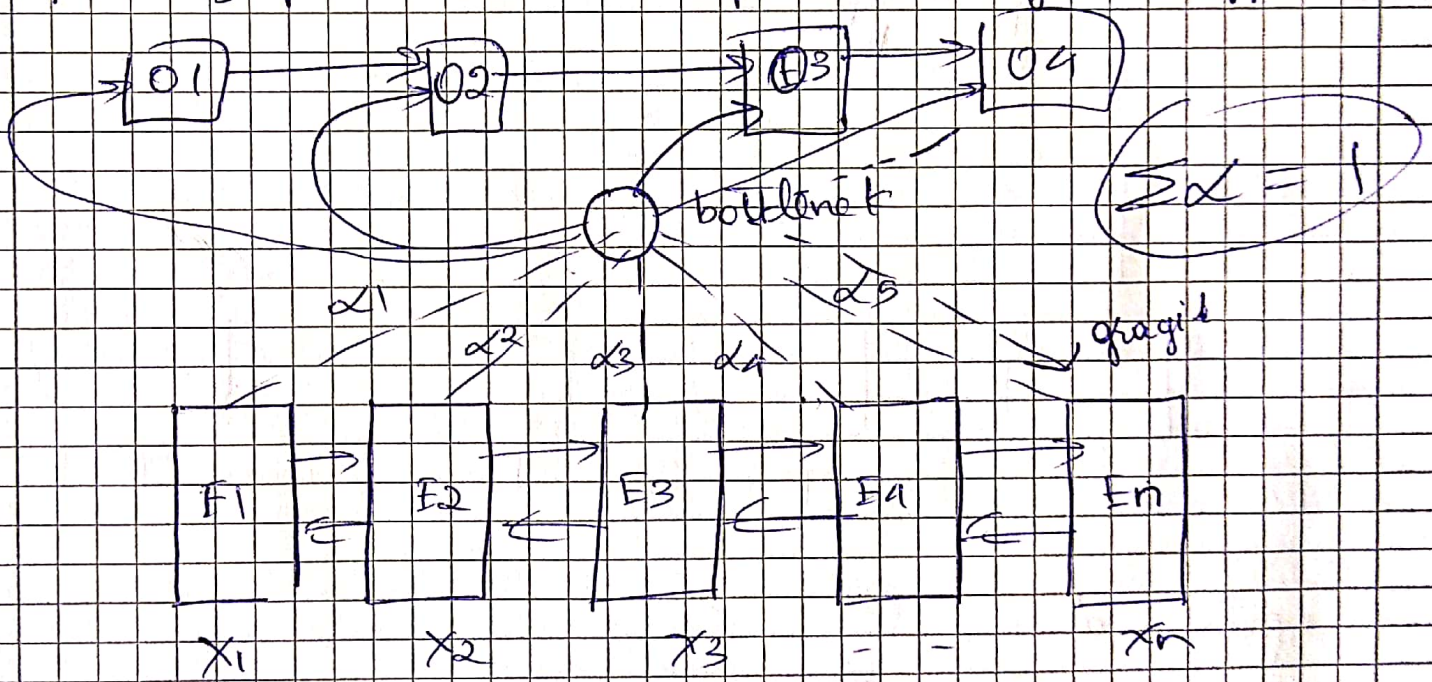
Order of prominence

Ily for decoding
- $P_2 \rightarrow P_2, P_3, P_1, P_4$
- $P_3 \rightarrow P_3, P_2, P_4, P_1$
- $P_4 \rightarrow P_4, P_3, P_2, P_1$

Use α

→ This is dynamic hence allows for better gradient update

→ Encoder can be bidirectional to recieve info both from past & future. α is for this dynamic effect



→ At each timestep at decoding we get different weights to α hence gradients flow/update due dynamic

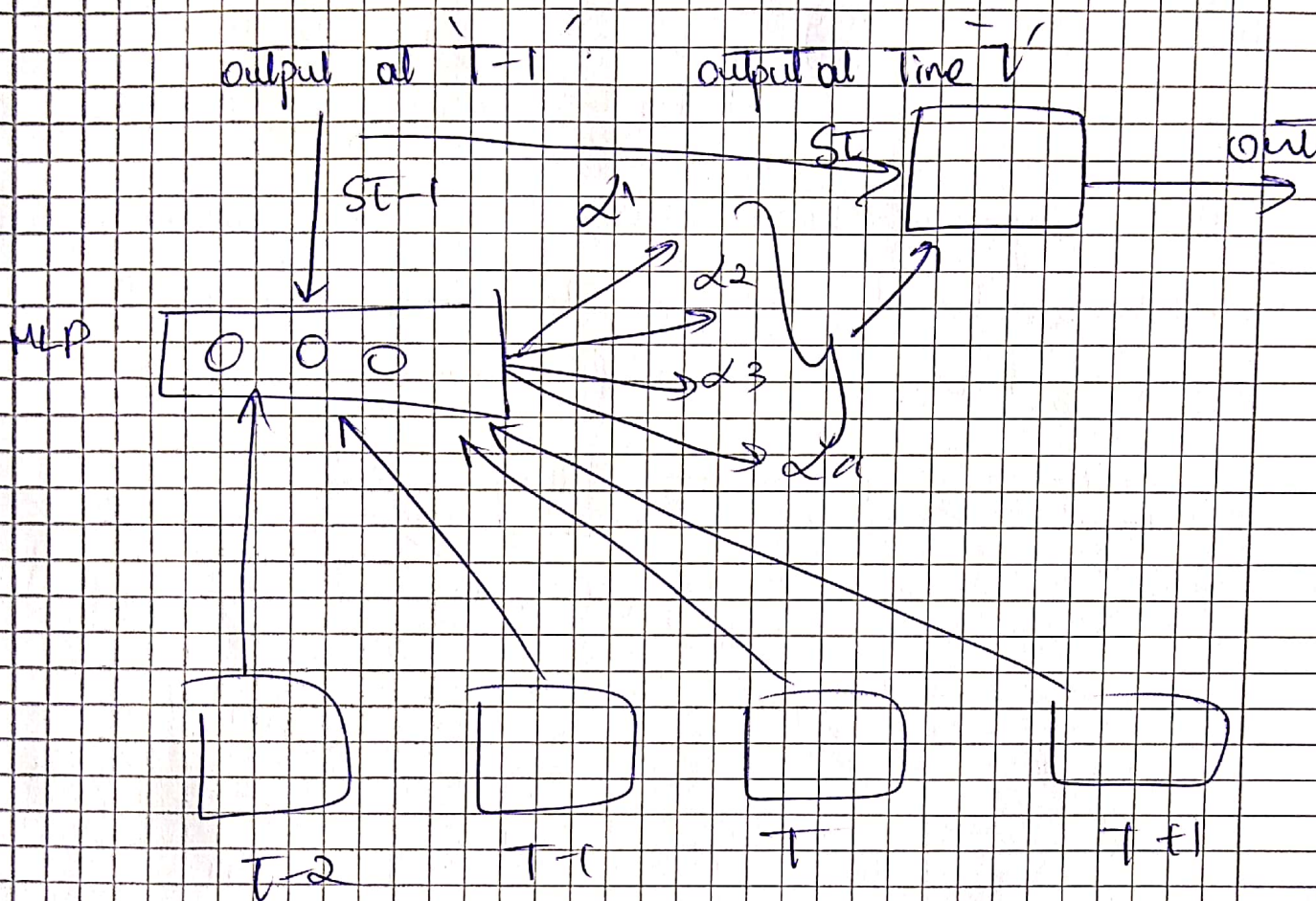
→ At each step decoder consumes a different combination of encoder states. Glimpse of all parts of sentence/sequence available

→ $$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

$e_{ij} = a(s_{i-1}, h_j)$

$a(s_{i-1}, h_j) = v_a^T \tanh(W_a s_{t-1} + U_a h_j)$ } One layer MLP

(v_a: scale; U_a: weight)



Computational aspects

→ Quadratic $O(n^2)$ computation if we have n decoder steps & n input then $O(mn)$ operations

→ for each pair of i (o/p) j (i/p)

** Sum two vectors → apply Tanh → dot product

→ (can be done in 1 step) for all inputs j given i is fixed (o/p is fixed time)

** → Due to vectorization, $O(mn)$ could be computationally fast

→ Important : Attention is not just simple MLP layer in context. It mostly defines a sort of similarity function

Dot Product Attention

→ To give attention to current time step decoder, we take output of previous time step & take dot product wit every encoder states

→ Hence there are no parameters / to be trained (MLP). It's just a simple series of dot product (similarity)

→ Think this in context of Info retrieval. which decoder state has more similarity to which encoder state

For example from previous page:- P4 is less similar to P1 than P3

→ Downside : Since there are no weights involved both decoder - encoder dimension matrix must be matrix multi compatible.

$W_{Dec} S_{dec} + W_{enc} h_{enc}$ ✗ (additive)

$S_{dec} \cdot h_{enc}$ (dot)

Alignment matrix | Sentence - english sentence
| sentence → prison sentence

Location based attention

Neural Turing M/C

Why To see all encoders (global)? Check some defined neighbourhood based on previous o/p. Use This To calculate next step

Content based attention

Dot product

additive

Looks at neighbouring regions for some localized region based (global) ✱ all encoders

Hard attention (Visual domain)

→ Consider human vision. Human vision / receptive field even though is large, can only focus on couple of objects at a time for rest of them would be blurred

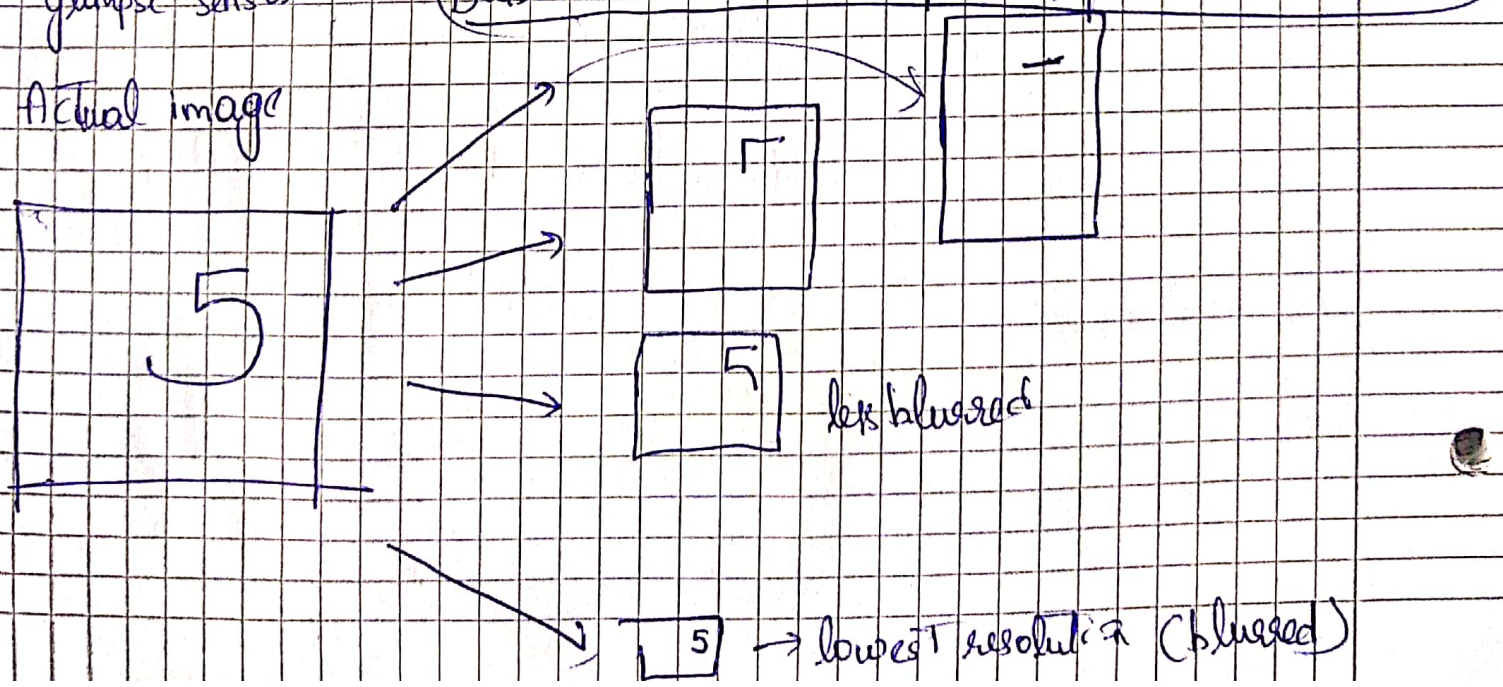
✱✱ → Hence humans use a hard attention mechanism, To see other objects we need To Turn our eyes elsewhere

How To do it?

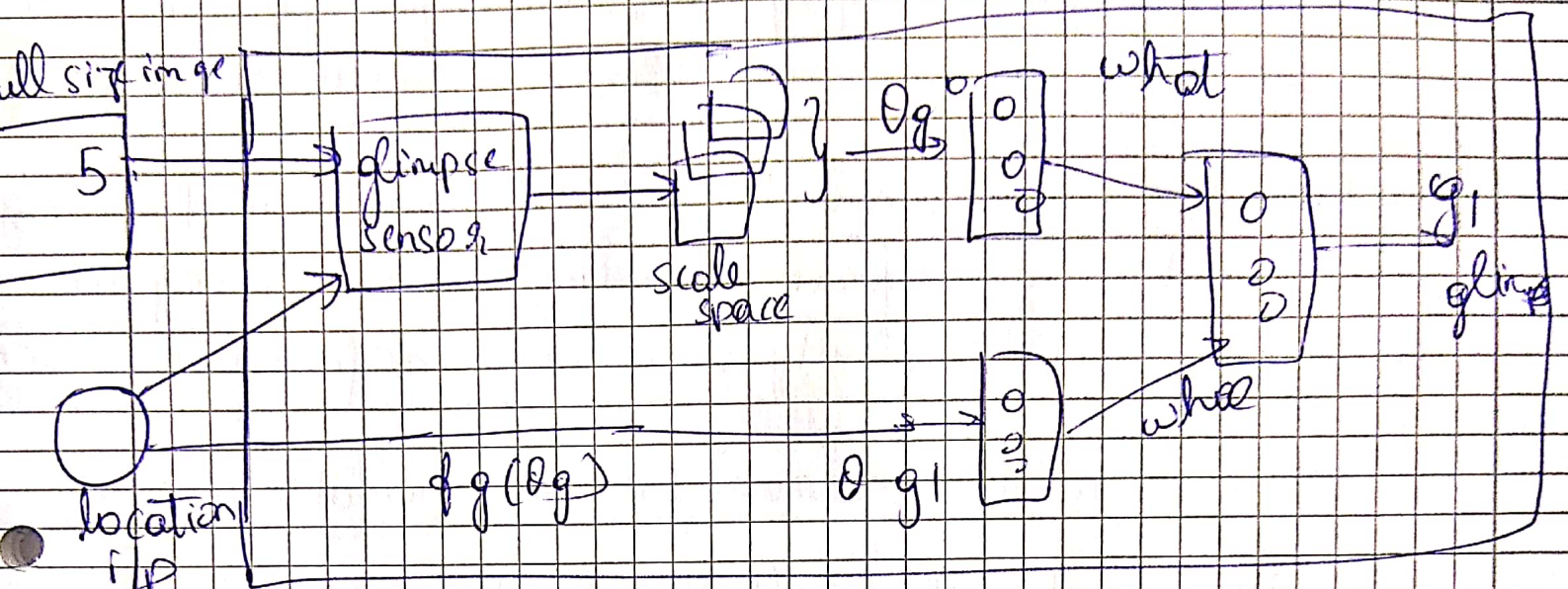
Recurrent attention model (RAM):

Glimpse Sensor - Builds a scale space of three-4 levels ✱

Actual image



Concatenate scale space like in dense net & pass through n/w

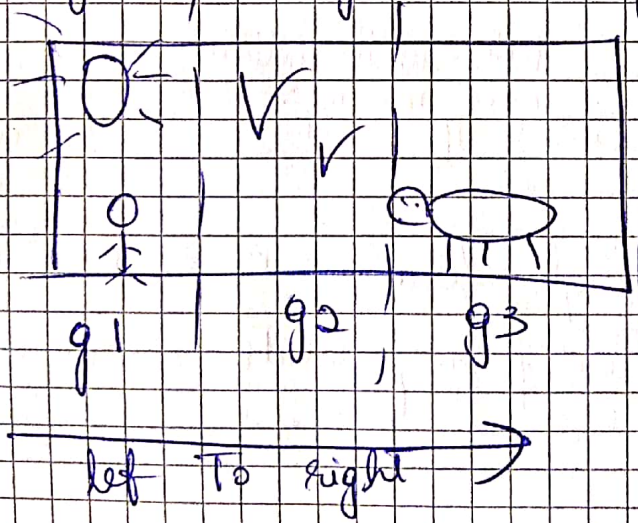


Activation n/w :- what do you want to see / or seeing now?

location n/w :- Predict where should we look next step

Recurrent n/w :- Propagate state fwd

Imagine watching a scene from left to right



g_1 o/p → Sunny day / man walking

g_2 o/p → Sunny day / man work / birds fly

g_3 o/p → Sunn — — — / cows graz

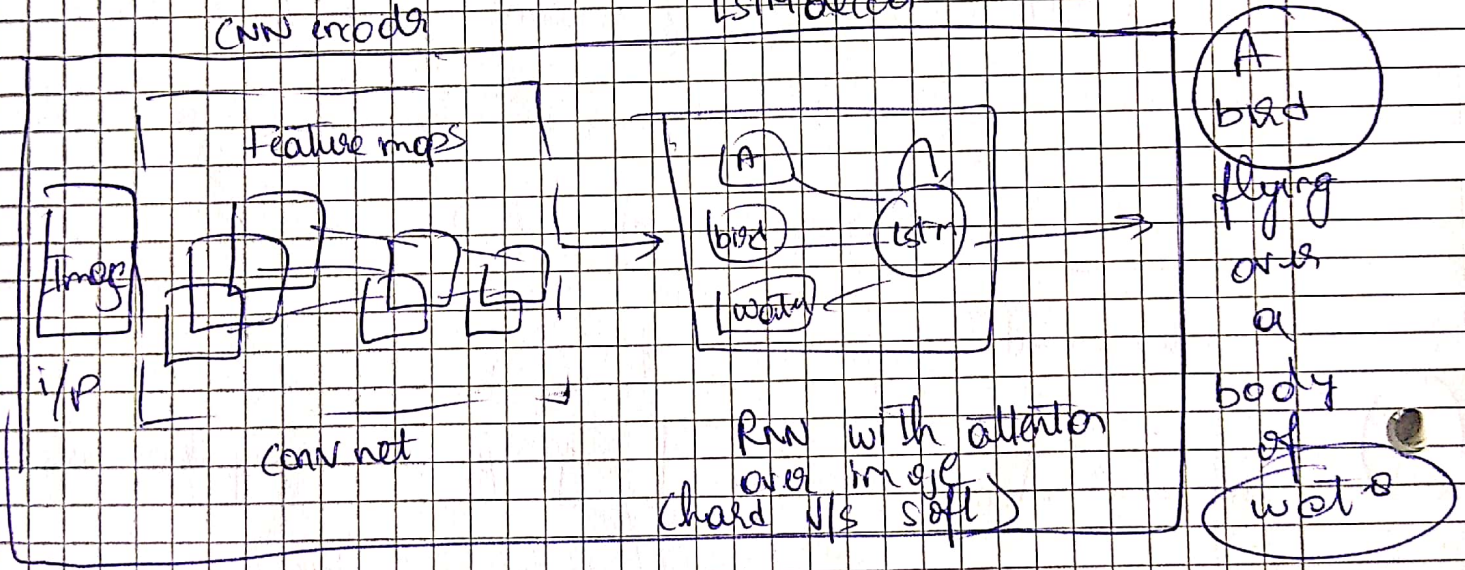
→ Here sunny day is propagated from $g_1 \rightarrow g_3$ by RNN whilst attention is only given to g_3

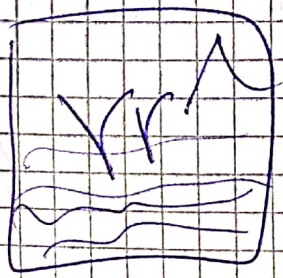
** → Problems : Scale space is gaussian sampling & random, Hence cannot back propagate (sampling → random → non deterministic) easily Hence train it using reinforcement learning. gaussian filter change

** Deep RAM : Provide a context vector to provide a good starting point. Just using reinforcement learning ends up starting at random state. O(1) complexity or O(K) as focus on only some states

** → CTC is a hard attention mechanism but with tractable gradient

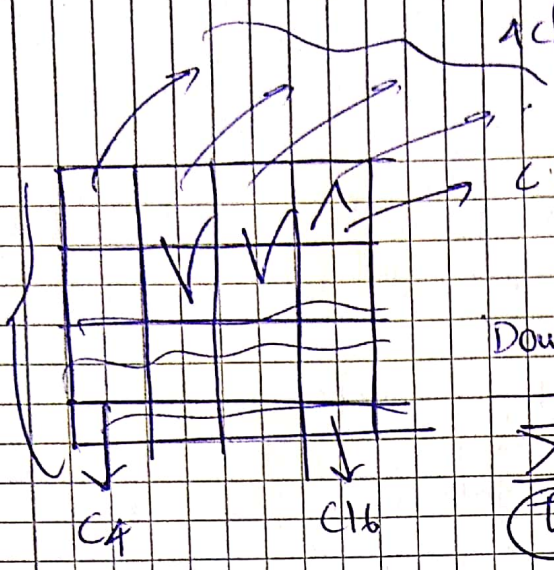
→ Show, Attend and Tell (Soft V/s hard visual attention comparison)





A bird → high focus → more attention
water → low focus/resolution → more blur less attention

→ With soft attention they create a grid over image and each grid cell becomes a softmax attention



16 class softmax for soft attn

Doubly stochastic regularization

$\sum_{t} \alpha_{ti} \approx 1$; Index now is 'i'

↑ class

** → At each time step ~~consider~~ we need to pay attention to all C_i - C_{16} grid cells equally

Aggregated over Time for single grid cell

** loss: $L_d = \underbrace{-\log(P(y|a))}_{\text{Typical crossentropy}} + \lambda \sum_{i}^{\text{Location}} \left(1 - \sum_{t}^{C} \alpha_{ti}\right)^2$

sum over all location

for each location compute deviation from 1 across Timesteps

Memory based n/w

Von neumann architecture : Separate memory & program

→ Neural n/w Takes care of computation

→ RAM Takes care of memory

** → Can operate on any knowledge base, KB can be swapped in/out

→ Doesn't need neural n/w to represent data ; (DBpedia)

↓ complete

→ We can scale up/down / swap KB

Concept

→ Neural n/w acts as controller (↑ RNN, MLP) which controls where & when to read or memory based on external/internal input

→ Heads select portions of memory & read or write on them

→ Memory is real valued matrix & not binary **

→ Everything differentiable, which means we cannot make (discrete/hard decisions), only soft based

access only one memory cell, not differentiable

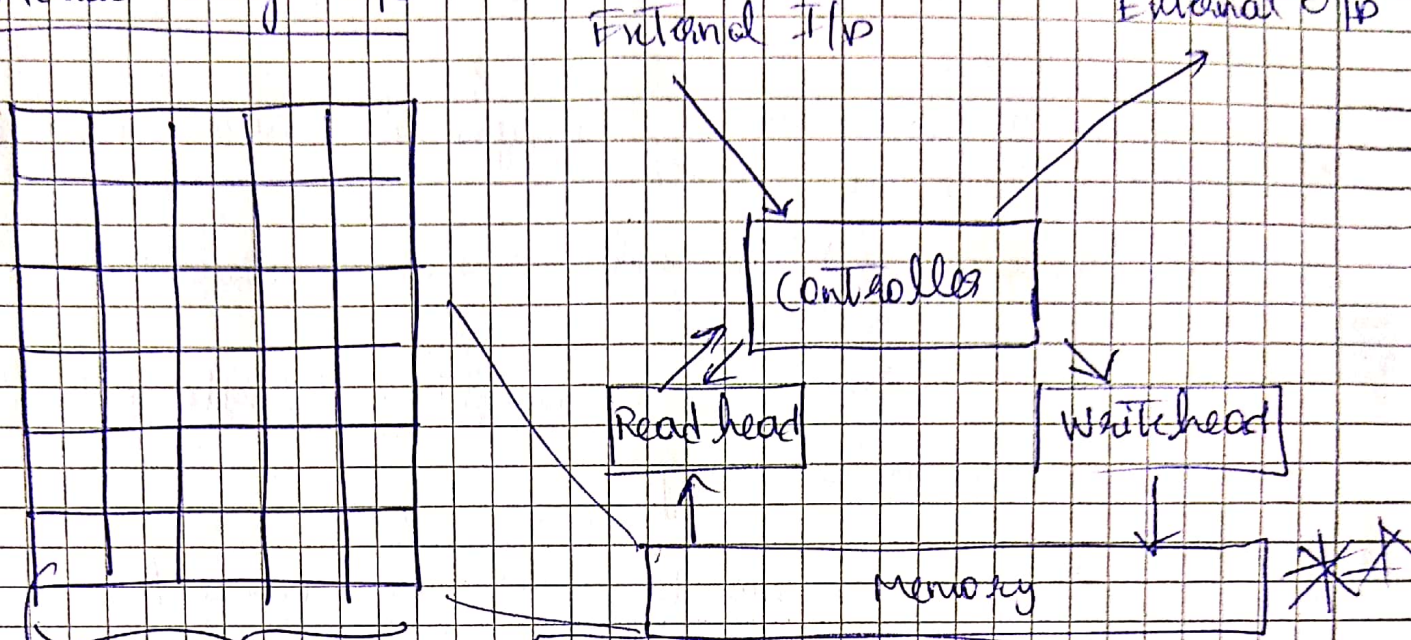
Memory networks in practice

→ Strong supervision needed To access a specific memory cell To address respective query ↘ direct / reward based

→ babl data set

7/06/20

Neural Turing m/c



Explicit memory Why? We may need to access large amount of info if we already stored in explicit memory, we can swap in/out similar KB

each cell n-dim vector (real value)

N finite memory position

we need n/w to store & represent actual data/pattern in terms of weights. If it's

NTM Read/write :- Train by gradient descent, differentiable every where

Soft read (content-location attention) :- Read all memory cells every time but with a softmax mask for ex:-

c_1 ↓

0.01	0.05	0.03
0.2	0.3	0.1
0.3	0.05	0.05

} we read from every cell but focus on (21, (22, (31

soft write $\quad r_t \leftarrow \sum_i w_t(i) M_t(i)$

Soft write erase :- Basically we write everywhere but with again a softmax mask (attention mask)

$M_t(i) \leftarrow M_{t-1}(i) + w_t(i) a_t$

Soft erase : $\tilde{M}_t(i) \leftarrow M_{t-1}(i)\left[1 - w_t(i)e_t\right]$

forget gate ; Memory matrix ; row i in matrix

Memory at previous time ; Bigger value leads to more forgetting

e_t, a_t & Erase & Write control vectors comes from controller

** → May lead to slow training as we need to read/write external memory can be alleviated by use of GPU

** Important Question : How to know where to read/write & how much to read/write ? This is where controller comes in. We use attention mechanism (content-based addressing)

i) Content based (Associative Memory)
Retrieve by partial query/input — Like to search engine

ii) Interpolation
Based on previous memory cell accessed, Use previous attention to determine next access point

iii) Location based addressing (Much more powerful than others)

→ Needed for variable (varying content) (By their nature vars hold variable values & their values are not constant)
read

Since vars keep on changing, we cannot query by their value we need to address the location they are stored — content based

Content based can emulate location based if content has some part of location embedded along with content

query q_t = [1 0.2 0.1 0.09 | 0.1 0.2]
content info | location info

a) Content based :- Use Cosine Similarity, key → query, Query compared to Memory cell y → apply softmax

b) Interpolation :- Use old attention to update new attention

No need for content $w_t^g = g_t w_t^c + (1 - g_t) w_{t-1}$

old attention

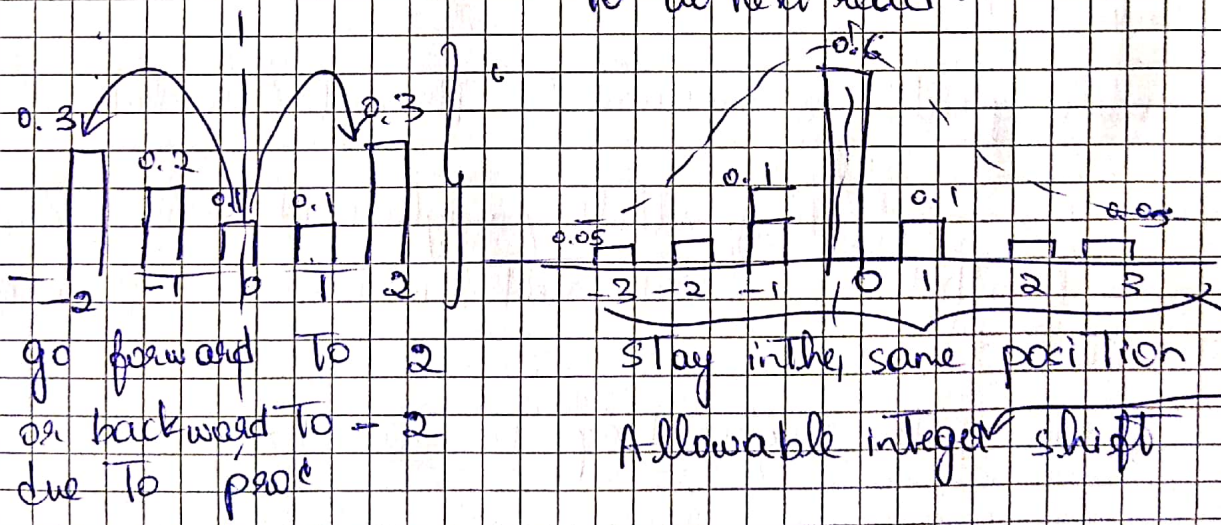
How much current info should we keep? → NTM should learn this

How much we keep from last attention

c) NTM location based addressing :- Once you are at the position/location of memory cell, you need to read discrete number of memory cells. We can only work with continuous due to different

How to deal with this?

i) Get a distro over allowable integer shifts. Offset from where we are now → How much we are gonna go further to do next read



Why we need modulo :- Consider memory size as 16. Since controller is a neural n/w it can give any value, like 100/500 / -20 etc which are physically not present on cell. Use modulo to reconvert them back

Another option is to make network learn a regression model which takes a real valued i/p & converts into distribution

eg:- I/p → 6.7 o/p → (6 × 0.3) + (7 × 0.7)

regression

Take some real number regression it into distribution

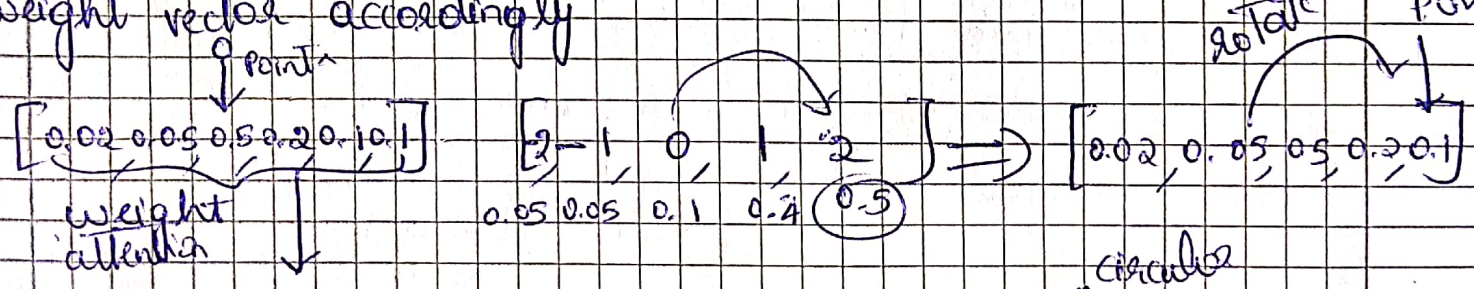
memory cell 6 memory cel

How to shift attention based on distribution?

Use weight vector on distribution

length of weight vector is same as no of memory cells

if we want to attend to next / previous location we rotate weight vector accordingly



This modulo operation is implemented through circular convolution

ii) Sharpening : Soft fuzzy attention / weight masks lead to disperse of actual attention (Attention mask which is spiky, begins to flatten out)

→ Use softmax with higher temp to lower dispersion

NTM Addressing flow diagram

→ We start with previous state, memory of last attention vector

→ We start with o/p of controller (neural n/w)

Step1:- Do content based addressing by using key vector (query) and key strength To get content based attention/weight vector

Step2:- Do interpolation using gates from controller that tells whether we need To only pay attention To last memory cell or The new content vector/memory cell that was generated by step 1 as part of answer To query

Step3:- Once we have gated attention vector Use location based To determine shift parameter from current location

Step4:- Once we have location based attention from step3 Do the sharpening To remedy dispersion. Use final weight vector To operate on data.

Pointer Network

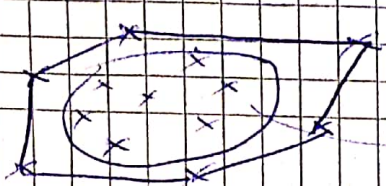
Encoder - Decoder :- Outputs are positions/indices of i/p seq

NN can learn sorting/permutations problems

Problems

Convex hull :- Network has To output a shape which goes around all points & includes others

Ex



i/p (Convex hull) → Output order of points of hull

→ All points contained in hull

Delauny Triangulation, Travelling Saleman, Graph Traversal, Scheduli
are problems for pointer n/w

Differential Neural Computer (DNC) (NTM 2.0)

Changes / Enhancements To NTM

a) **Added Memory management** :- Helps when scaling up, Machine doesn't waste memory, if memory becomes free it should be reused for writing again

b) **Temporal link matrix** :- Allows To track how memory was written from previous Time steps (Are there some memory that are read/written again and again?)

**
- → Facilitate sequential access to data not written To contigous block
- → Facilitate retracing steps through time
- ↓ Use address pointers To allow writing To non-contigous block (fragmentation problem)

**
- → It has a (free list) To know where are free memory location To write new info
- → Gates keep Track and control how data is accessed

DNC searching by Time :- Change attention based architecture

Architecture

One write, 2 read heads for controller
read multiple things

Attent Transformer notes

- Why Attention? Dis advantages of Vanilla LSTM

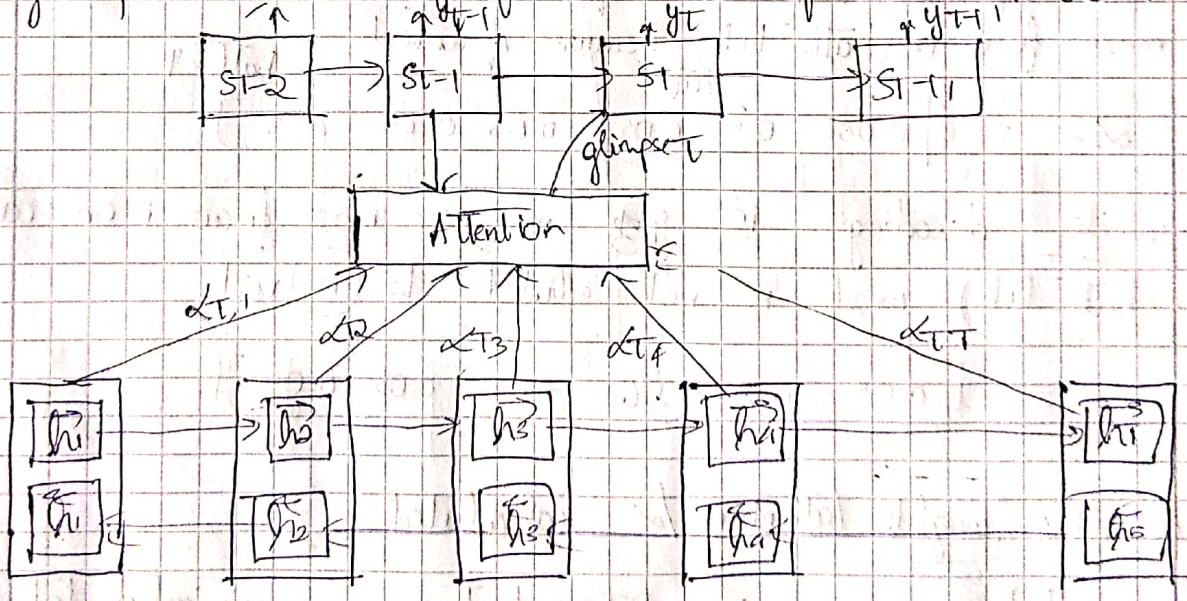
→ In vanilla RNN the output of the encoder is carried out till the last encoder which acts as bottleneck! Obviously for long data there are chances that the last encoder could have forgot about data at begining

→ Gradients should flow from last encoder to first causing problems in Training (Vanishing Grad)

- Intra attention :- Here all encoder states are fully connected to decoder state. Each encoder consists a kind of summary of entire sequence with focus on some relevant parts. Hence

→ No bottleneck, because of direct connection to decoder state & summary at encoder

→ No grad problems, because of direct access of encoder to decoder



→ At each time step decoder consumes a different summary with different focus from encoder state, Hence it has glimpse over entire sequence (glimpse vector)

How to interpret / calculate output at time 't' (y_t) given input X?

- WKT, y_T is dependent on its current encoder state S_T and glimpse / context vector at T ie

⇒ probability of predicting y_t given its past output & inputs are a fnc of decoder state & glimpse

$$p(y_t \mid y_1, y_2 \ldots y_{t-1}, X) = g(y_t, s_t, c_t)$$

(o/p, i/p)

How to calculate c_t? (glimpse vector)

→ Intuitively, glimpse will take a weighted input from all encoder states in parallel, Hence $c_t = \sum_{j=1}^{T_x} \alpha_{tj} h_j$ → weighted sum

so for every time-step of decoder there is a different weight / focus on encoder state

Ex: The(y1) cat(y2) is(y3) in(y4) The bath Tub(y5).

↑ s1 ↑ s2 ↑ s3 ↑ s4 ↑ s5 ↑ s6

→ while decoding cat (y_2) α focuses on (cat) more than others it also gives more focus on (bath Tub) because of context

$\alpha_2 = [0.05 \quad 0.5 \quad 0.05 \quad 0.05 \quad 0.05 \quad 0.3]$ (0.5 ← cat, 0.3 ← bathtub)

→ Here while decoding is y_3 we see more focus is on cat & is because (bath Tub) word is not related to (is) verb.

$\alpha_3 = [0.05 \quad 0.2 \quad 0.6 \quad 0.05 \quad 0.05 \quad 0.05]$

How are the weights / attention / α calculated ?

1) Additive attention (Bahdanau paper) 2) Dot product 3) Scaled dot product

→ Three Types of attention, each calculated differently & have their own advantage & dis

Additive attention:- An MLP model would take in previous decoder state and one encoder state & calculate attention for this pair.

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

e_{ij}: i → decoder time step, j → encoder timestep; V_a^T → scalar; s_{i-1} → previous decoder state; h_j → for every encoder state

→ fix $i = T-1$ to calculate o/p
→ iterate through all j's (encoder state)
→ increment i to calculate attention for next decoder state
} $O(n^2)$ solution

** Problems
→ Need to learn weights of MLP which is an added burden

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ij})}$$ } Attention

→ $O(n^2)$ operation is fast as done on gpu, but still quadratic

Dotproduct attention :- Instead of MLP take a dot product to find cosine similarity. $e_{ij} = s_{i-1} \cdot h_j$

Advantages :- No weights like that of additive, hence simple

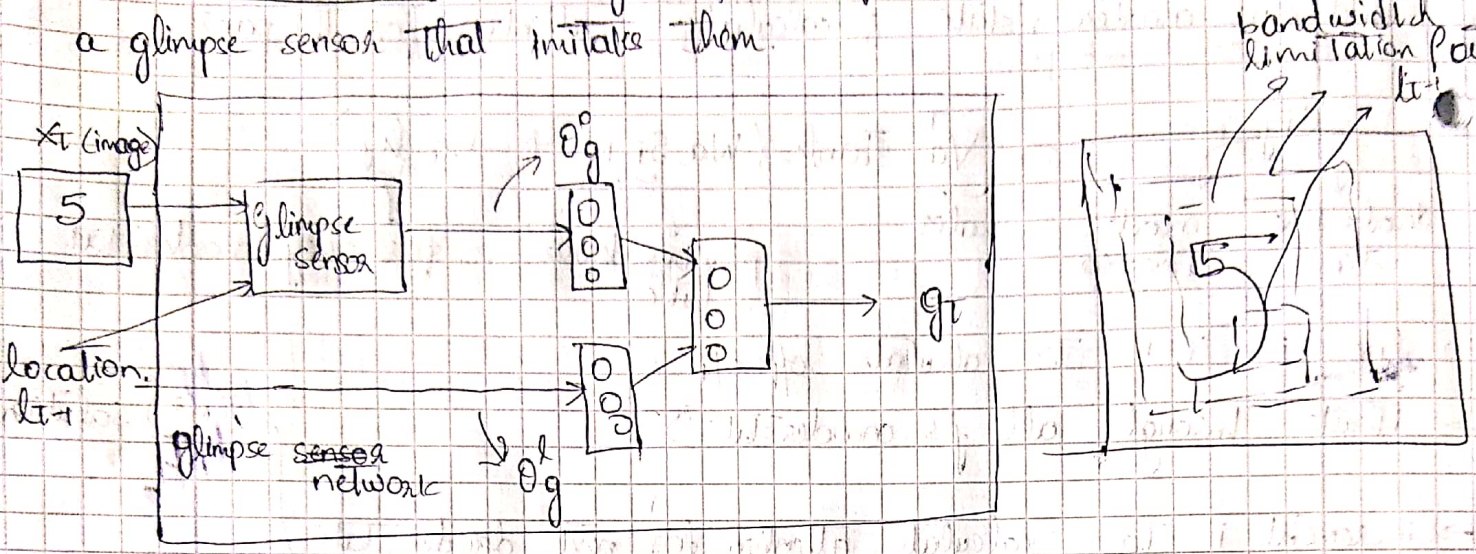
Problems :- s_{i-1} & h_j should have compatible dimensions as we are directly multiplying them. we cannot adjust weg dims using weights like we did in additive

DRAM

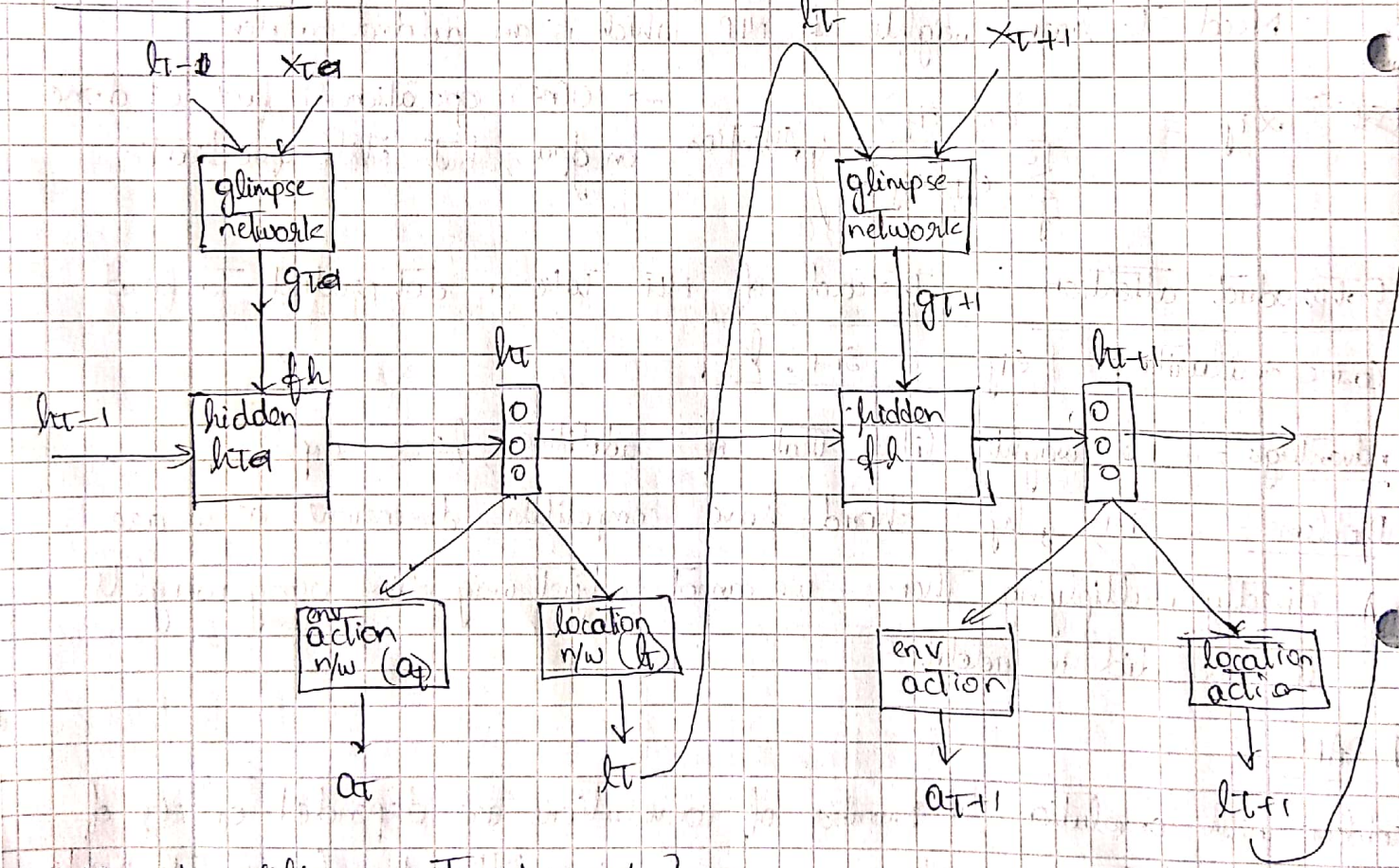
Problems with convolution :- Number of convolutions are depended on size of image. Or atleast even if we increase ~~size~~ filter kernel size To make same amount of conv, increasing is still a function of image size.

Solution :- See an image like human does, humans take some initial context about image and only focus on required part of image, dull the rest. This dulling causes less info need to classify / action on any image.

Glimpse DRAM paper : Taking inspiration from human vision we can build a glimpse sensor that imitates them.

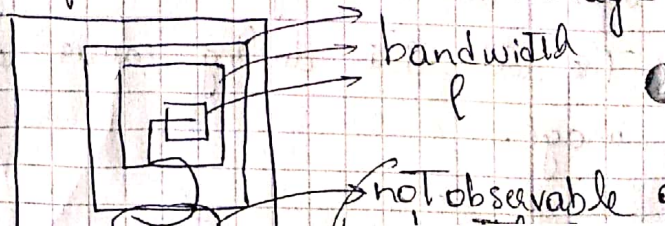


Model architecture :



How does glimpse network work?

→ At each time step 't' given an image x_t glimpse sensor builds up a scale space focussing on location l_{t-1} Hence we observe same image at different scales, but we won't have full bandwidth access to image (States are partially observable



** → Can learn Translation invariance like CNN, but will take more glimpses

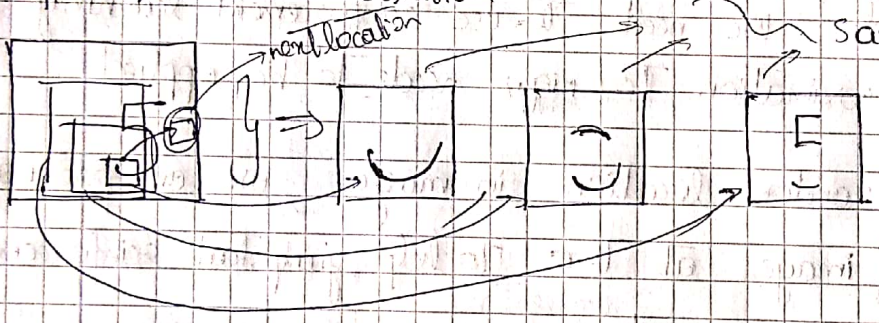
→ This function $f(x_t, l_{t-1})$ helps encode retina like representation in vicinity of l_{t-1} and dim other places. Hence uses high resolution at pixels near l_{t-1} & low resolution else where.

→ Both location l_{t-1} (θ_l) & bandwidth (θ_e) are combined together by bunch of Naural net layers $\{\theta_l, \theta_e \rightarrow \theta_g\}$ & final o/p θ_g layer forms g_t a glimpse vector having both location info & info of image at diffrent scales

How does the model work?

i) Sensor Network :- Takes x_t and l_{t-1} as input. Applies bandwidth function at location specified

→ Builds a scale space representation of image each representation of scale space has same resolution



→ Uses the location l_{t-1} and Linear layers to provide glimpse vector g_t which encodes high res near l_{t-1} and low res elsewhere

ii) Hidden State :- RNN hidden state comprises of info about past observations if combined with current glimpse g_t, it can encode what are the actions need to be taken next $h_t = (g_t, h_{t-1})$

iii) Action networks :- Two Types i) location decider ii) Environment Action decider

a) Environment action :- $a_t = f_a(h_t)$

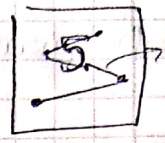
→ Decides what action to take next (Classification, moving left/right etc) based on hidden state

b) location action :- $l_t = f_l(h_t)$

→ Decides which particular pixel/location to go next/focus on next based on hidden state

iv) Reward : If agent's action / classification leads to success then it is rewarded with max reward. Over Time T we sum up the rewards accumulated

$R = \sum_{t=1}^{T} r_t$. We try to maximize this R



Why can't agent cheat and keep going to same location again and again since classification is right ?

Soft vs hard attention and problems with DRAM

→ DRAM's attention is specific to a location hence discrete. Discrete cases are non-differentiable. We need to use reinforcement learning and some gradient approximation technique needs to be applied

→ Instead of starting at random location in image, we can preload some context / hidden state of image at t=0 to help jumpstart reinforcement learning process in location we need.
↳ vicinity

Soft	Hard
Deterministic	Stochastic
Exact gradient	Approx gradient
O (input size) → all context considered	O(1) → single or constant num of context/loca
Typically easy to train	Hard to train

Show Attend and Tell

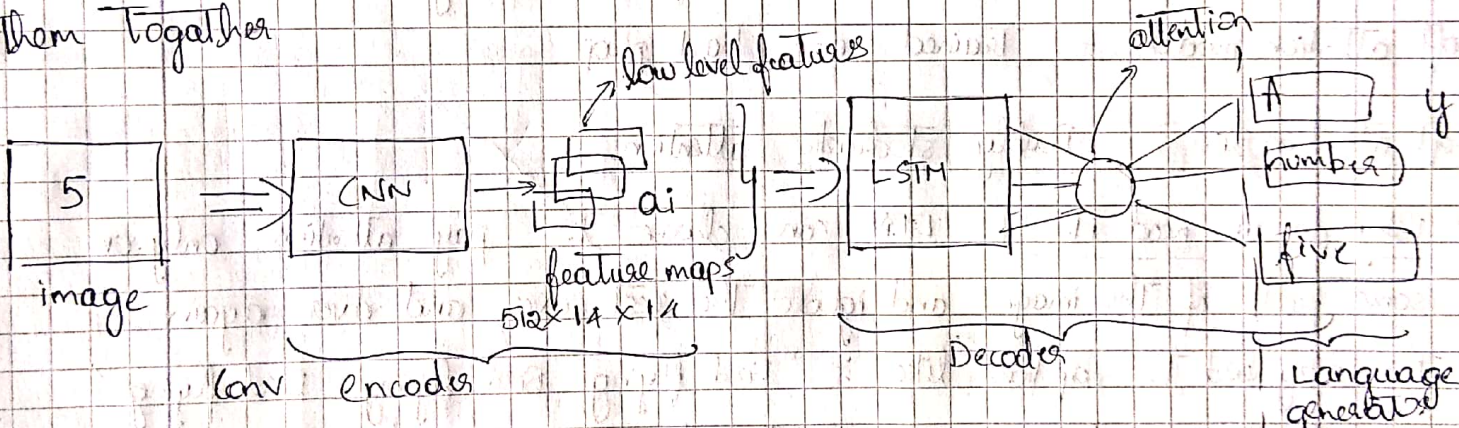
- Image captioning is a difficult problem because we need to understand relevant parts of the image and relationship b/w them. After that this should be translated to text describing objects and relationship

Problems with older approaches

→ Training Conv net and Language model separately & combining them
→ Throwing away lots of low level representations that are important to caption model and taking one whole image at each step

Current approach: Both Convnet which produces encoded low level features and LSTM attention decoder which utilizes these low level features to produce captions

Hence model learns abstract concepts rather than detecting objects & stringing them together



- **Implementation:** Both hard and soft versions of attentions were used

I) CNN Encoder:
→ Image is encoded into a tower to Transformed to lower dim feature maps. Each map has some part of image represented
→ Each map is straightned out and put in a 2-D Embedding matrix of 512×196 which represents a vector (a_i)
→ This allows decoder to selectively use only relevant parts of image from this matrix/feature space.

II) **Decoder LSTM:**
→ Decoder tries to predict next word conditioned on previous words generated, previous decoder hidden state (h_{t-1}) and context vector (z_t) $\Rightarrow (y_i - y_{t-1})$

→ Context vector is a function on Embedding matrix and attention $z_t = \phi[a_i, \alpha_i]$. This allows vector to select suitable subset of low dim features and weigh them accordingly

Ex:- Bird flying over water ⇒ (Water, over) is not important hence any subset of (ai) having this image features are weighted less. Since we are focusing on bird, (Bird, flying) related lower dim representations are weighted higher

→ Attention is calculated using additive type of model (Bahdanu) conditioned on lower representation a_i and previous hidden state h_{t-1}

attention $\alpha = f_{nc}(a_i, h_{t-1})$. This can be thout as probability that location i is the place to focus for producing next word.

Training:-

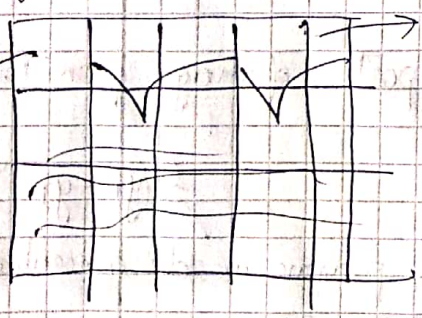
Hard attention based model:- Trained on Reinforcement learning

Soft attention models:- Trained on back prop

Soft attention loss:- Doubly stochastic attention

→ Why do we need it? LSTM can choose to pay attention only on some parts of the image and ignore the rest over and over again. This may lead to captions like:- "Bird flying Bird flying. Bird flying" because it focuses only on some aspects of image.

→ Solution:- Force LSTM to cover every part of image with great focus equally over time $\left(\sum_t \alpha_{ti} \approx \tau \approx 1\right)$ Over time the attention at location i should add up to τ, if $\tau = 2$ Then we will provide great attention twice over all parts of image



ⓘ location

loss:- $L_d = \underbrace{-\log[P(y/a)]}_{\text{cross entropy}} + \underbrace{\lambda}_{\text{scalar}} \sum_i^L \left(\tau - \sum_t^C \alpha_{ti}\right)^2$

force LSTM to provide good focus on every part of image over time

Transformers and Self attention

LSTM with self attention

Inter v/s Intra attention

Self attention :- Encoding current token by considering other relevant tokens in the sequence

→ Correlation b/wn Tokens in the sequence

→ Attending To self

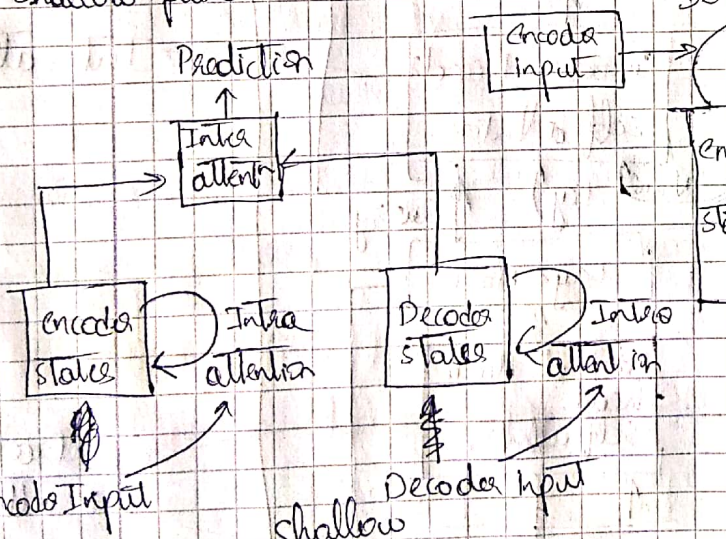
Intra attention :- Decoder attends To encoder state

Shallow fusion :- Fusion happens only when we do prediction

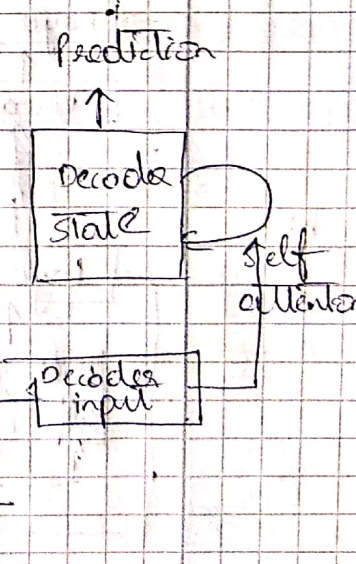
Deep fusion :- Fusion happens at every stage irrespective of prediction, Hence encoder-decoder Tightly bound

→ Intra / Self attention is present in each encoder and decoder pasts

Shallow fusion



Shallow



Transformers : can do sequence - sequence modelling w/o recurrence or convolution.

Process sequence parallely

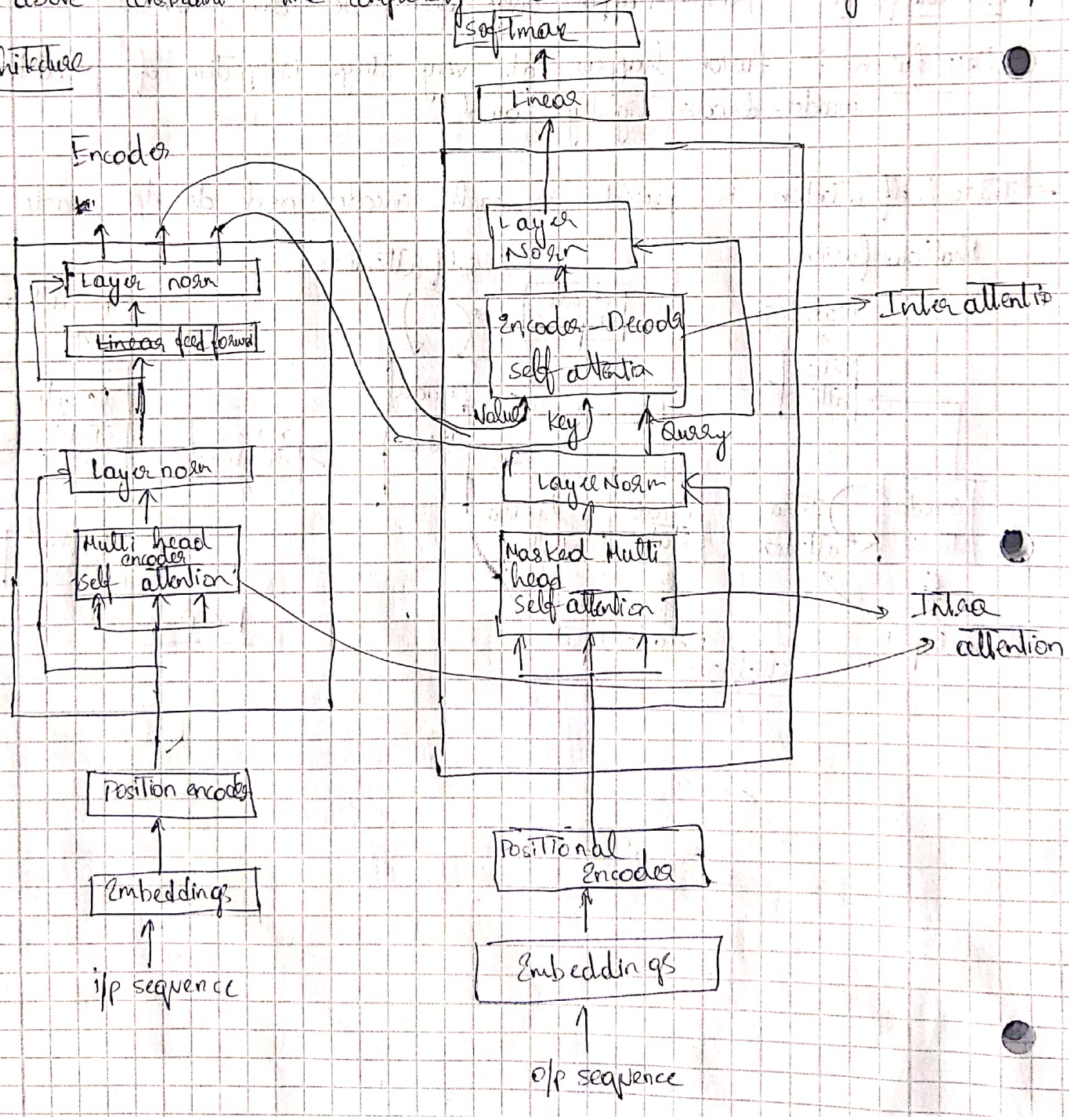
Need for new architecture : Even with advent of attention and shortning of gradient paths, the processing still has to be done in sequence as All encoder states must be populated by respective positions of sequence, Then only we can parallely apply attention over them.

⇒ hidden states of encoders are often huge with dimensions of 512/1024 unit

⇒ (Usually) sequence length ≪ no of hidden dimensions (512/1024). But often in some cases like poems we have a long sequence. This needs sequential access still.

with above constraint Time complexity $O(n * d^2)$ seems to be huge. Inter Transformers

Model Architecture



Peeling the architecture

At encoder side

High level Explanation: The input sequence is tokenized and represented as a vector using some embeddings.

→ Another vector describing position of token in the sequence is modulated with embedding. Each embedding has same dimension, $d_{embed} = 512$.
↳ positionvector

→ Each token is then converted to unique Trio of (key, value, query) vectors having same dimensions ($d_{key} = d_{val} = d_{query} = 64$). This is done through a simple matrix multiplication with some learnable weight matrices W_K, W_V, W_Q

→ For each token in encoder self attention block, we find its ~~most~~ similar counterpart in sequence by cosine similarity of Query token v/s keys of ~~core~~ sequence. We use this similarity score to find weighted sum of values in sequence as o/p. This is done ~~not~~ with multiple blocks in ||el, hence multi headed.

→ The weighted values of each token of each attention block is concatenated and reprojected back to same embedding dimension space $d_{embed} =$ by a feed forward network. These meta/intermediate embeddings serve as input for next stack encoder

→ After repeating the process over $N_x = 6$ encoder stacks we extract final key, value and Query Trio from encoder's last stack

At decoder side:

We kick off with "<s>" start token during the begining and calculate the embedding (positional modulated as usual. ↳ with

→ During attention calculation at decoder end we mask the future states that are not predicted to avoid copying behaviour

→ Query of the decoder is issued upon key / values of encoder to in the encoder-decoder inter attention blocks. Again there are multiple attention heads. Aim is to find next key, value ~~token~~ emb from input given current word's query

→ After reprojecting the ~~K & Q~~ weighted values of this block back to embedding space, process is repeated $N_x = 6$ times till last decoder.

→ At the end a linear + Softmax layer is applied to predict next token which would be used as input to decoder in next cycle.

Lower level details (Decoding Architecture)

Embeddings :- We cannot use words as it is hence we need to project each word as a fixed dim vector ($d = 512$) in this case to be used as i/p to transfo

Positional Encoder :- We provide and process embeddings inside encoder & decoder block parallely, hence we have lost the inherent order that was present during sequential processing of sequence in RNN based networks. This becomes a sort of like bag of words model. Hence we modulate a couple of sinusoidal waves which are continous (hence can handle arbitrary sequence length) instead of using a fixed length positional codes like (000, 001, 010, 011 etc)

$$PE_{(pos,2i)} = \sin(pos / 10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos / 10000^{2i/d_{model}})$$

Encoder Self attention :- **Why self attention here?**

a) Computational Complexity per layer = $O(n^2 d)$ v/s RNN $O(nd^2)$

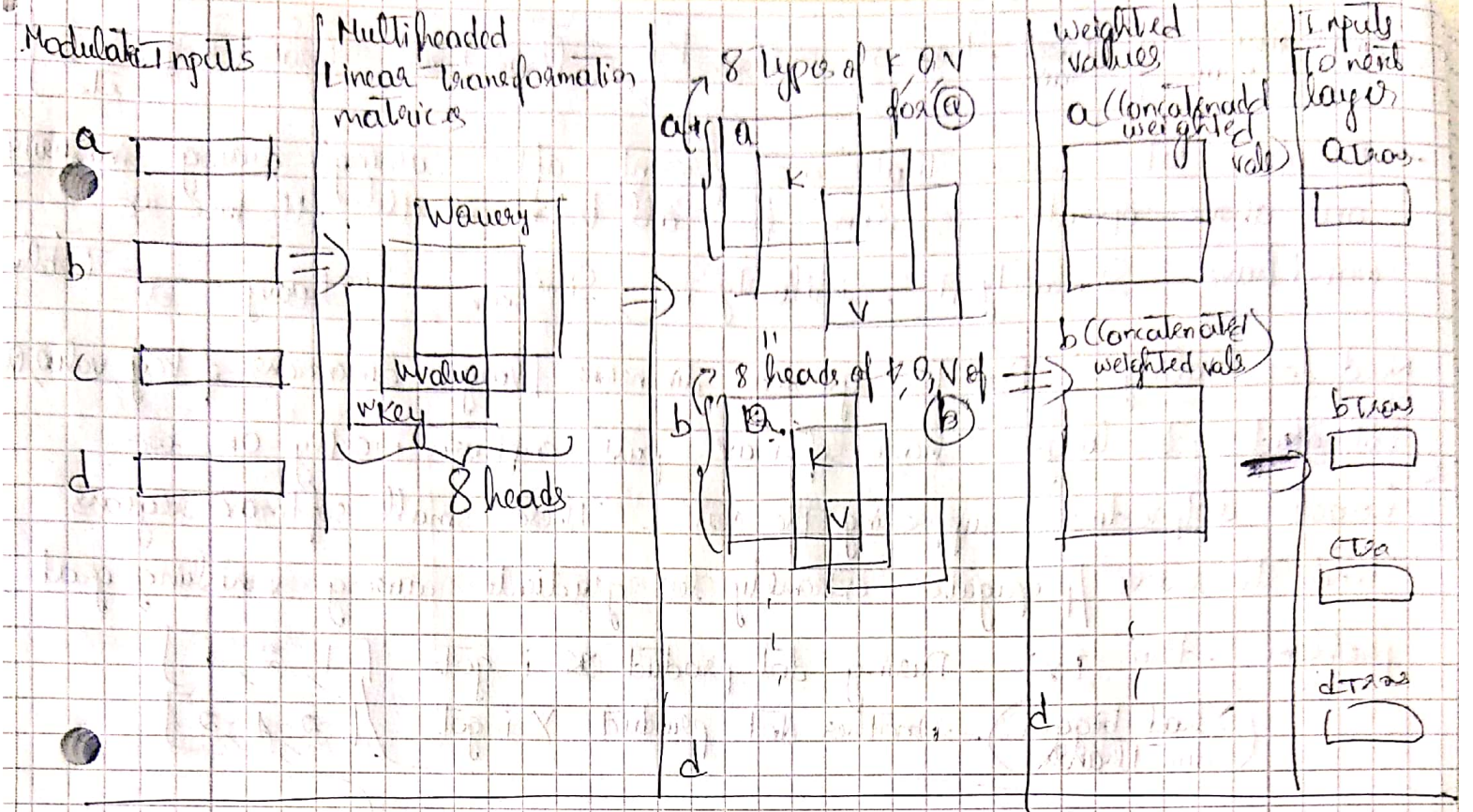
→ (for most cases not all ✱✱) sequence length (n) << dimensions of hiddenstate or K, Q, V

hence Transformer performs much better.

b) **Parallelizability** :- Even though in attention layer we find similarity for each key with entire sequence, this $O(n^2)$ operation is just matrix multiplication and can run ||elly + efficiently on GPUs whilst RNN cannot leave the sequence /order in encoder state $O(n)$ sequential operations

c) There is a constant time path from last softmax layer to first layer of encoder/decoder, unlike RNNs. Hence gradient updates / training is faster.

d) **Interpretability** :- Self attention brings the interpretability aspect to our prediction as we can easily visualize the reason for prediction through attention weights



What happens in a single attention head?

I) Calculating key, Query & Value for each Embedding

→ Once we recieve an embedding of $d = 512$, we use learnable linear matrices W_K, W_Q, W_V to transform the embedding to a Trio of Key, Query, Value of 64 dims each.

→ Here (key, value) are closely related. They are like answers to questions posed by query. key acts as an index of database which is actually pointing to value.

Example:- It is (raining) heavily on Monday?

(how?) (when?)

Consider (raining) : Value, key (mostly) (not completely represents word rain itself and maybe it's neighbor words by small amount. But for Query there are two possibilities how? → Heavily, When → Monday. Hence we see Query would be influenced by these two words rather than "raining".

II) Calculating similarity

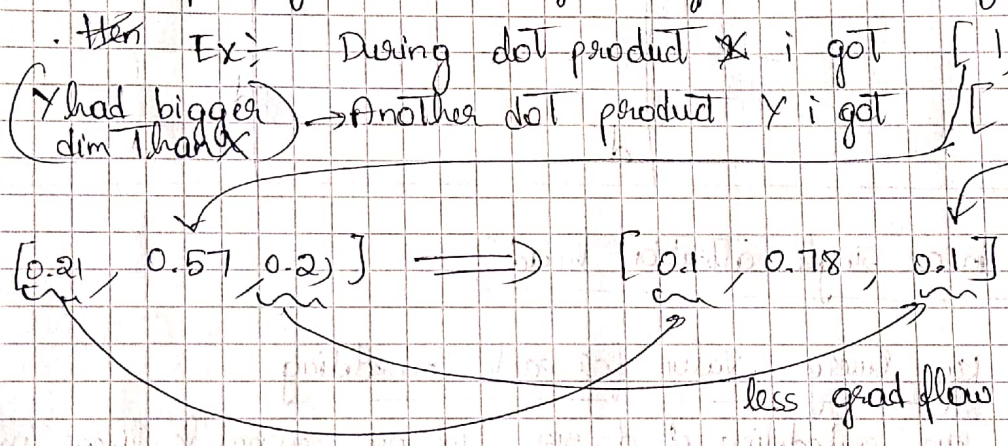
→ Now once we have this Trio. lets take a token query and compare with every other words in sequence including itself through dot product

(key)

$\text{Similarity}_{rain} = [\vec{Q}_{word\ rain} \cdot \vec{K}_{IT}, \vec{Q}_{rain} \vec{K}_{is}, \vec{Q}_{rain} \vec{K}_{rain}, \vec{Q}_{rain} \vec{K}_{heavy} \ldots \text{etc}]$

→ We also take tokens that are present after "raining" during similarity and above operation is done for all tokens in the table. Hence we would have Similarity it, Similarity is, Sim_{rain}, Sim_{heavy} ... etc Totally

Need for scaling :- Due to growing dimensions / larger dimensions of key, value, a Dotproduct gets larger, hence softmax puts emphasis mostly on ~~two~~ largest dotproduct suppressing the rest. These small softmax regions tend to receive / propagate extremely less gradients causing vanishing grad problem. ~~Then~~ Ex:- During dot product ~~X~~ i got [1, 2, 1] (Y had bigger dim than X) → Another dot product Y i got [2, 4, 2]



Divide each similarity by $\sqrt{\text{dim}_{key}} = \sqrt{64} = 8$

$\Rightarrow \text{Sim}_{rain} = \text{Sim}_{rain} / \sqrt{64}$

III) Normalizing Using Softmax :- As expected and seen before, we will normalize the similarity scores to add up to one using softmax.

$(\text{Sim})_{rain} = [0.1\ 0.1\ 0.5\ 0.1 \ldots 0.1]$ $(\text{Sim})_{heavy}$ [0.1 0.05 0.2 0.3 0.2 0.1

IV) Finding answer for query :- Unlike databases fixed results, we would return a sort of fuzzy / probabilistic weighted (Values) of matching keys

⇒ ~~Value Weights~~ $\text{Answer}_{(rain)} = W_{IT} \times V_{IT} + W_{is} V_{is} + W_{rain} V_{rain} \ldots$

$\text{Answer}_{(rain)} = 0.1 \times [\underset{64\ dim}{\ldots}] + 0.1 \times [64\ dim] \ldots$

Formulas :- Scaled similarity per token per head

$$Sim_T = \frac{[Q_T \cdot key_T^T]}{\sqrt{d_{key}}}$$

- Softmax normalized sim

$$SSim_T = \frac{e^{Sim_T}}{\sum_{j=1}^{N} e^{Sim_j}}$$

- weights

$$\text{Attention}(Z)(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_{key}}}\right)V$$

→ So at the end for each token of each attention head we have weighted values of entire sequence

- Multiple Attention heads :-

Why is it needed? :- This allows model to attend to information from different representation subspaces at different positions

What does this mean? Consider the sentence below

" Ram and Sita went to forest and she saw a golden deer."

→ If we have a single attention head then it could probably predict only Subject (Ram, Sita), Verb (went, saw) Object (golden deer)

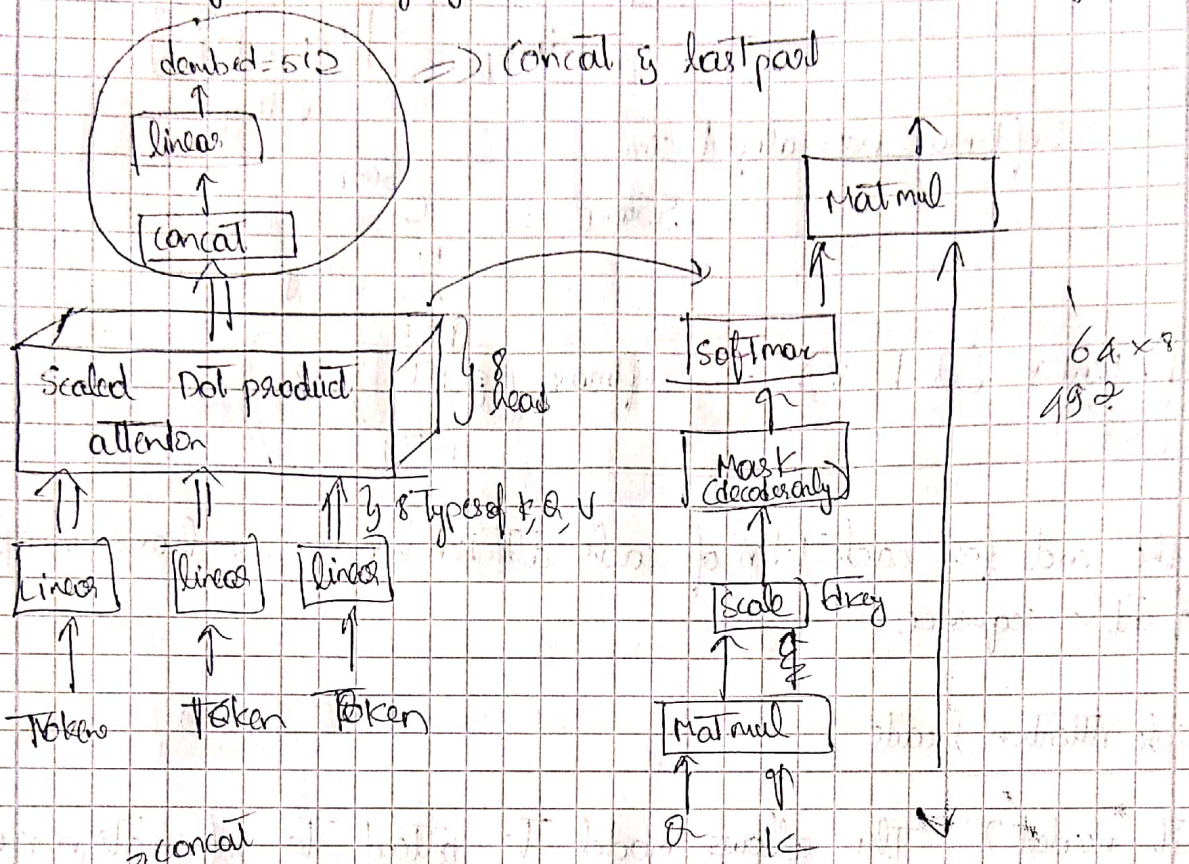
- But is this sufficient?

For example :- if we ask the question "who saw the golden deer?" We will get the answer as "she" Then we ask "who is 'she'?" model has no clue, because it's not tracking gender. We humans tend know it intuitively, but for machines we need to provide multiple perspectives of the same sequence

→ In Transformer model we have (8) attention heads, hence for each token we have 8 perspectives hence 8 different vectors of weighted value per token

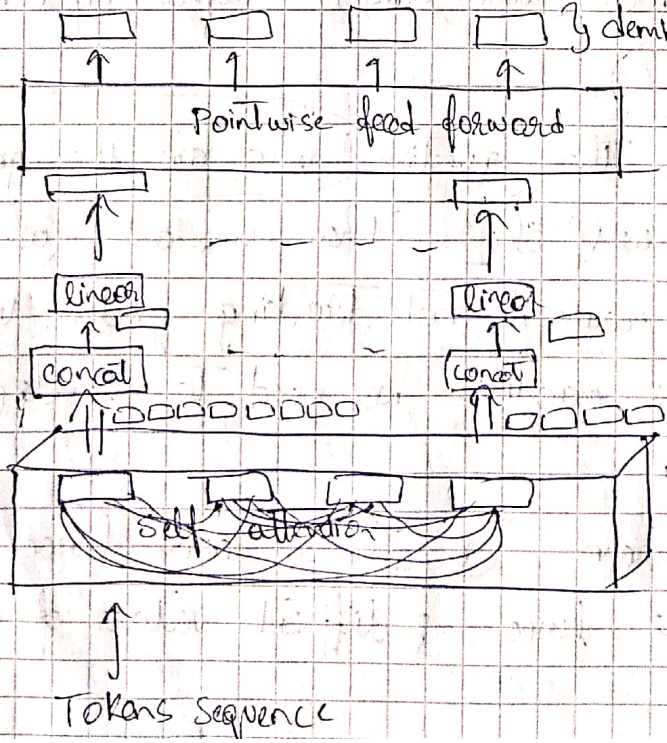
→ At the end we just concat all these 8 representations of each token to one vector per token. Then bring it back to embedding representation

dembed = 512 by multiplying this with a learnable weight, matrix W_o



Example :- $Z_{Ram} = [Z_{1Ram} + Z_{2Ram} + \dots Z_{8Ram}]$ → concat

Pointwise linear feed forward :- this same network is applied to all weighted value tokens in parallel. We also use a Residual connection with layer normalization and dropout. Hence this operation is also done. Relu activation is used



Decoder side : At decoder side we have both intra(self) attention and inter attention (Encoder decoder). Both are again multihead attentions

→ As usual the decoder input is modulated by positional encoder and fed to decoder self attention layer.

→ Masked decoder self attention :-

Consider decoder predicting next word in below steps

a) <st> b) <st> Je c) <st> Jesuis d) <st> Je suis student

e) <st> Je suis student <eos>

→ We can see that during prediction we don't know the next word hence we don't know all future words.

→ Hence its better to mask out and not consider all future states during our attention calculation, This is done by using $-\infty$ as during softmax

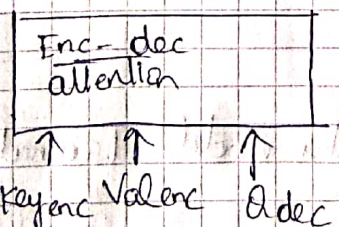
$\frac{e^{-\infty}}{\sum e^x} = 0$ hence zero attention

a) [<st>, $-\infty$, $-\infty$, $-\infty$, $-\infty$] b) [<st>, Je, $-\infty$, $-\infty$, $-\infty$] - -

→ Apart from masking which forces to look into only previous decoder states we see that the attention calculation is same as that of encoder. This is like forming a question based only on previous experience.

Encoder-Decoder Inter Attention :- We recieve key, Values from last layer of encoder and use decoders query to lookup for answers on next token

→ This is like a question-answer type of scenario where the question is asked by attention models decoder query and answer is taken from encoders states regarding next token.



→ Here masking is not present as we are actually looking for future token

→ At the end we use a linear + softmax layer to get next token and calculate loss & propagate gradients.

Training : We actually train the model by teacher forcing for example : "Je suis étudiant" the first token "Je" is predicted as "Chaud" which is wrong. We won't be feeding it back to decoder model for next cycle. Instead no matter what we will feed correct token "Je" as token for next cycle.

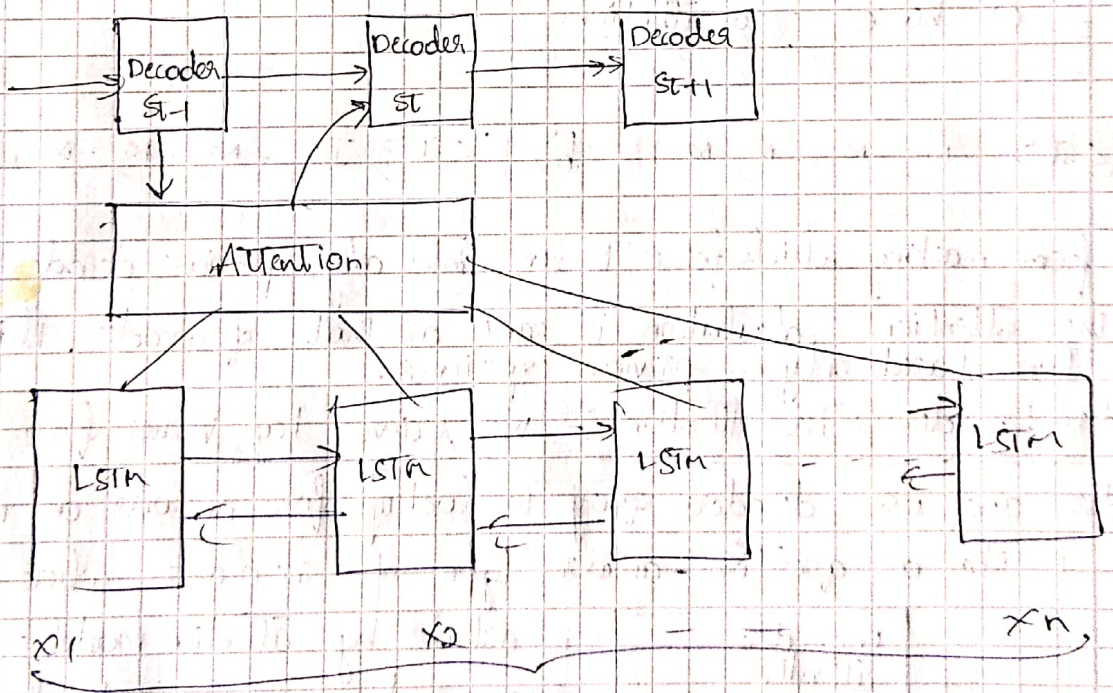
→ Also decoder model is shifted to right by one place hence we stop predicting as soon as we find <EOS> token

- 3:28

Language Models

Contextual Word Vector (Cove)

→ Contextual word embedding learnt from sequence - sequence machine translation model. Represents tokens in perspective of input sequence



→ It is nothing but a machine translation model trained on English-German or some other translation task

→ Encoders are bidirectional whilst decoder can only look into past. We have additive attention to facilitate translation

→ Here the theory is during translation Encoder stack has to learn about

Tokens syntadic (PosTag, Sub-Vb-Obj) and semantic (gender, location, person) in Hence after Training we detach the decoder and keep only encoder

- Usage during Run time

→ glove vector and Cove vectors are both concatenated as one provides skip gram perspective and another one contextual

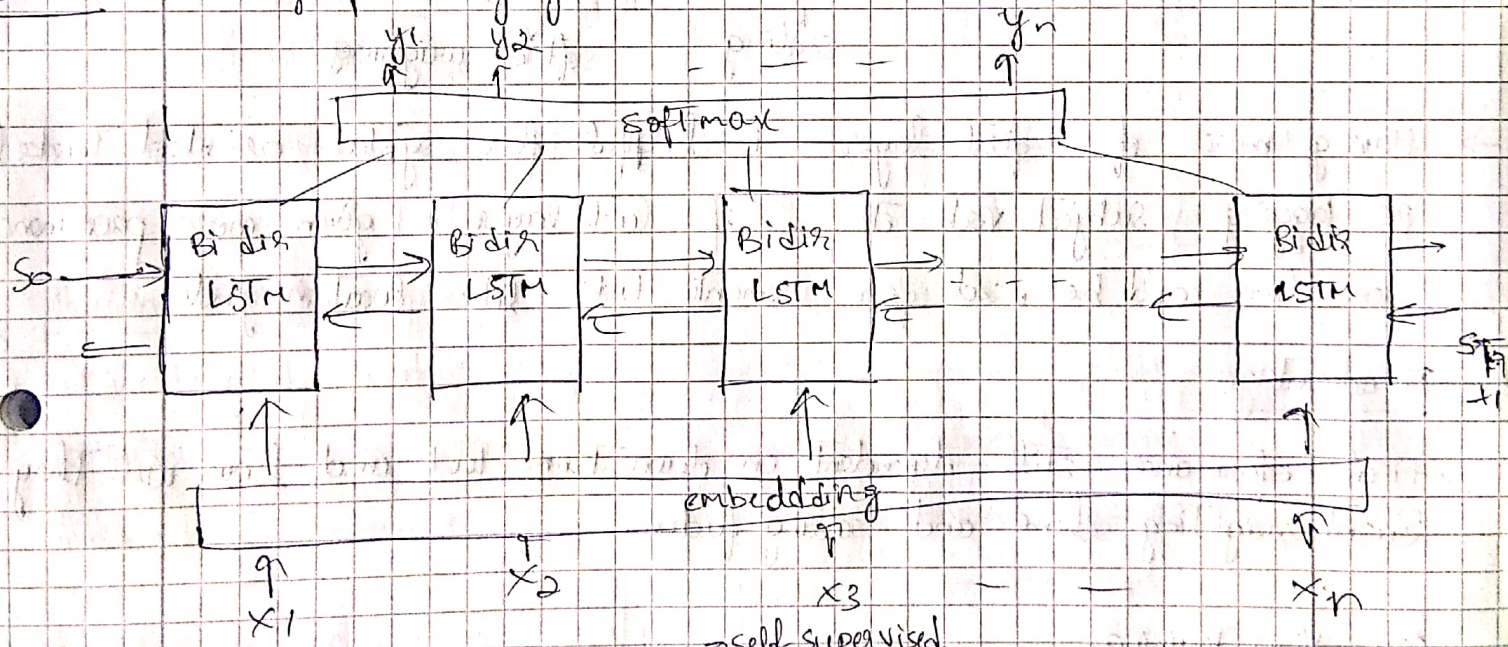
→ This is piped into Task specific model for further prediction

Disadvantages

i) Pretraining phase : Supervised learning, hence corresponding datasets must be present for Training

ii) Downstream prediction : Prediction quality directly depends on Task specific model, hence it's a major constraint

ELMO (Embeddings from Language Model)



→ ELMO can be pre trained in unsupervised fashion. → self supervised

→ The training task is to predict next word in the sequence given history and future.

→ Hence we have bidirectional LSTM encoders which takes i/p from embedding layer

→ Furthermore we stack LSTM on top of each other and to get different perspectives

Predicting current word based on past and future

$$P(x_i) = \begin{cases} \prod_{i=1}^{n} P(x_i | x_1, x_2 \ldots x_{i-1}) & \rightarrow \text{past} \\ \prod_{i=1}^{n} P(x_i | x_{i+1}, x_{i+2} \ldots x_n) & \rightarrow \text{future} \end{cases}$$

ELMO representations

→ After training we detach the output layer and have 2L+1 total layers with us (LSTM + embeddings).

→ According to task we must learn to combine representation at each layer in weighted fashion. We also use a scaling factor γ to scale representations to ranges required by the task. This weighing scheme and scaling factor must be done by task specific model

$$\text{Final embedding } V_i = \gamma^{task} \sum_{l=0}^{L} s_l^{task} h_{il}$$

scaling — softmax weighing

→ Having more of first layer is helpful for syntax oriented tasks like pos tagging, subject verb etc. if last layer is given more prominence then elmo could be used for semantic tasks like location, gender

Disadvantages

→ Final performance still dependant on down stream task and how well they learnt weighting scheme and scaling factor.

Cross View Training

→ One of the best as it tries multiple new methodologies at once.

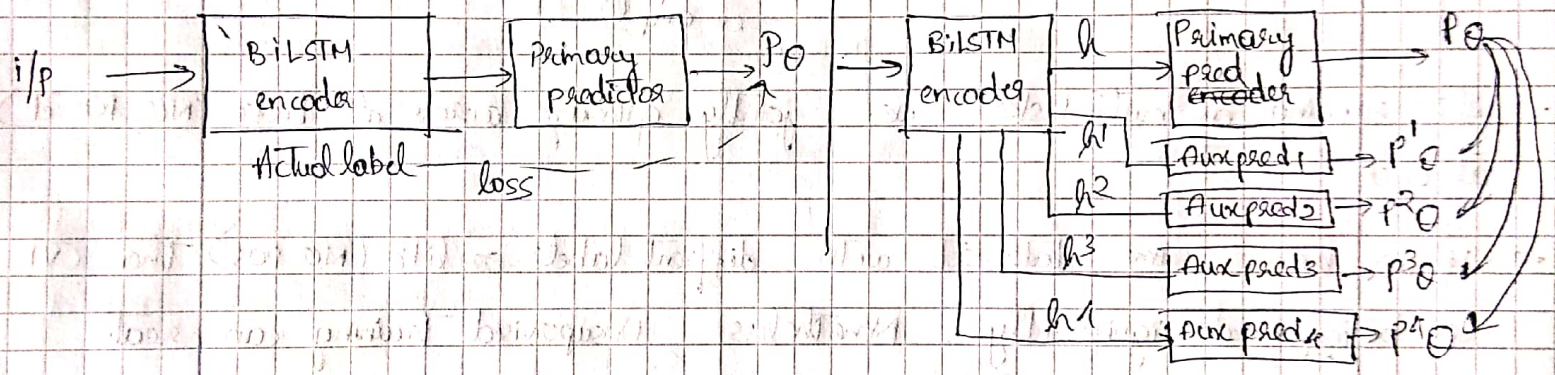
→ We have a huge corpus, most of it is lots unlabelled. We also have lots of small subsets of data annotated differently (like NER, POS, sentiment) We can combine all of the data in a big CVT module which at the end provides good generalizable embeddings over these multi tasks

Kannada NLP / NLP for languages shortening resources

- → During learning phase, model acts both as a both teacher that makes unsupervised main prediction and student that are trained on those predictions (auxilary). For supervised learning is actual with loss b/w actual & predicted propped back.

→ During unsupervised learning teacher forcing, if "auxilary" has same i/p as main then prediction is redundant, Hence we give masked input To auxilary modules and Try To match Them with main prediction.

- Types of learning in CVT



Training method

- → The prediction modules are really simple neural nets with softmax.

→ We alternate The Training b/w batch with labeles (supervised) and batch with no labels (unsupervised)

Example of Training:-

→ First we randomly select a task of supervised learning

→ We do the pass The i/p Through Bidirectional LSTM encoder and make prediction. Loss is calculated as usually by cross entropy and back propagated

Batch 1 → supervised Task 1 (Ner recog)
Batch 2 → unsupervised Task
Batch 3 → supervised Task 2 (POS Tag)
Batch 4 → unsupervised Task
Batch 5 → supervised Task 3 (dependency passing)
Batch 6 → unsupervised

$$Loss_{sup} = \frac{1}{|\text{Num of sup examples}|} \sum_{\text{examples}} (\text{crossentropy}(\text{Predicted label}, \text{Actual label}))$$

Unsupervised Training

→ This is the highlight of CVT which is main factor for improving quality of embeddings.

→ For the batch we have (K) auxilary modules which predict or are made to predict output as close to main prediction and auxilary modules don't have access to complete encoder states, hence they learn the context of the word and its neighbouring tokens as well

→ We get loss by comparing auxilary modules o/p with main module through distance function

$$Loss_{CVTunsup} = \frac{\sum_{examples} \sum_{k=1}^{K} Distance\ Similarity\ (Main\ module\ prediction,\ k^{th}\ auxilary\ module)}{|\ num\ of\ unsup\ examples\ |}$$

→ While running on multi tasks, we jointly optimize across all tasks (NER, POS, etc) at once

→ If we have same data set with diffrent labels or like (NER, POS) Then CVT can be improved significantly. Nonetheless, Unsupervised Training can create this effect from unlabelled data Hence improves efficiency & runtime.

Example = Sequential Tagger (Tagging POS, NER, Locations, general purpose)

Example : He is visiting (Washington) Tomorrow by Train. → Full sequence

Main (h) :

Aux 1 : (h^1) → Past context + current Token — He is visiting washington

Aux 2 : (h^2) — He is visiting → Past context

Aux 3 : (h^3) — washington Tomorrow by Train. → current Tokens future context

Aux 4 : (h^4) — Tomorrow by Train. → Only future context

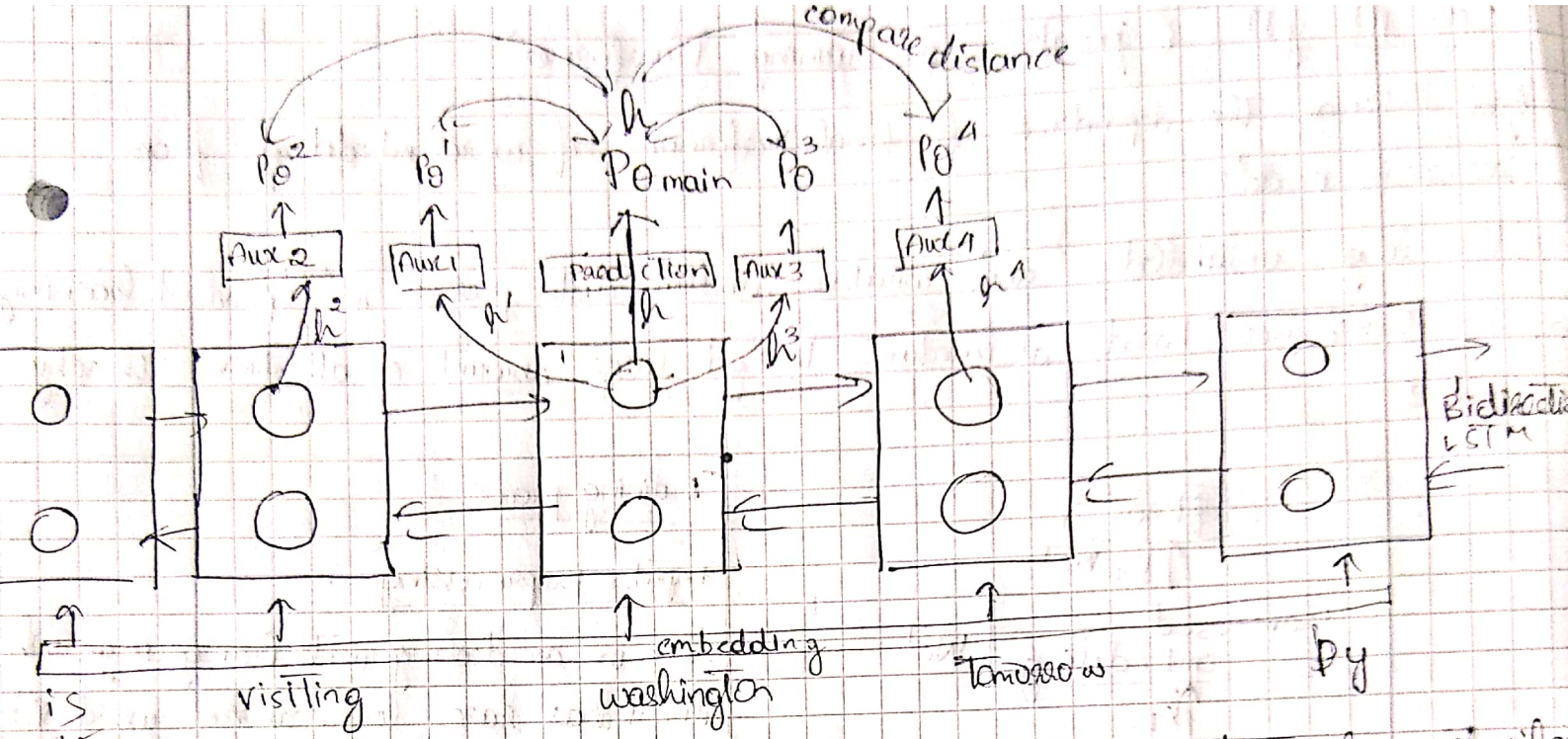
→ (Loss unsup calculation) ** → blanked out o/p

Pred main = [0.01, 0.05, (0.5), 0.25, 0.05, 0.05, 0.09]

Pred aux 1 = [0.01, 0.1, 0.4, 0.1, 0.1, 0.1, 0.19]

Pred aux 2 = [0.1, 0.1, 0.3, 0.3, 0.2, 0, 0]

Pred aux 3 = [0.01, 0.05, 0.4, 0.2, 0.05, 0.19, 0.1]



** → We try to find distance between predicted softmax dist from auxilary modules to main modules softmax. By doing so system learns that "visiting" and "tomorrow" may be ~~tagged~~ to location. contextualizing

→ The encoders are taken from first layer of stack as top layers usually have seen all states and we cannot derive h^1, h^2, h^3, h^4 if top layers already seen all state

→ ~~The~~ h_t → predict w/o correct context even though it has seen current token "washi

h_2 → No access to recent context & also current token

h_3 → Access to current token and not right context

h_4 → No access to recent context "h" and also current token

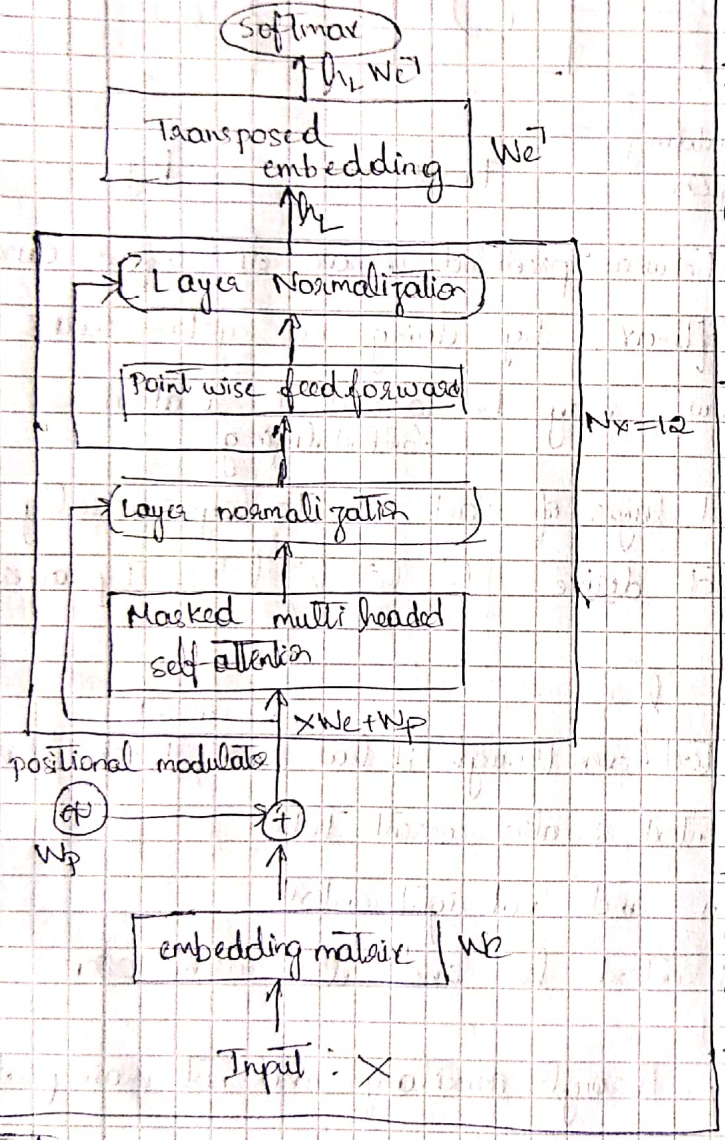
** → CVT loss is back propped only through auxilary and not main predict

Open AI GPT (Generative pre training Transformer)

Goal:- Remove the dependence of final performance of word embeddings from task specific models

→ Use large unlabelled and whatever labelled data present in semi-supervised learning

→ Use Transformer based architecture to get large amount of attention thereby context



Training process

Inputs preprocessing :-

Since we are dealing with language which almost has rare, less occuring words we cannot have every such word in our vocabulary

→ Use BPE to divide word to different segments and get most frequent sub token that's already present in vocabulary.

→ We also add positional embedding so that Transformer decoder is position aware

Contextual Model

→ We use decoder part of Transformer with masked self attention as we are predicting next word in sequence

→ There are 12 such blocks which are stacked upon each other

Output

→ Finally the last layers output is taken and we calculate next probable word through softmax.

Unsupervised training (Predicting next word in sequence) (for books this will be 10 k but for Tweets around 100)

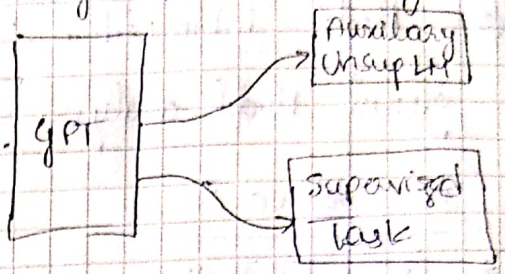
→ We predict next word of sequence given the 'k' predecessor (context window)

$$\text{Loss}_{unsup} = -\sum \log p(x_i | x_{i-k}, x_{i-k+1}, x_{i-1})$$

Supervised fine tuning

→ Depending on final task we either choose last layers last embedding or all embeddings

→ Lets say our supervised task is sentiment analysis. Then there is only one o/p per sequence, hence choose last layers last embedding



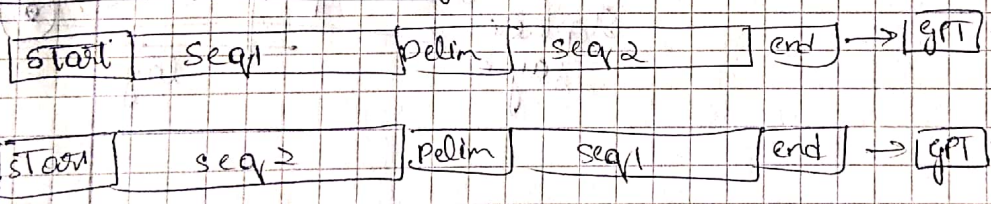
$$L_{sup} = \sum_{(x,y \in D)} \log P(y \mid x_1 \ldots x_n) \qquad L_{aux} = -\sum_i \log p(x_i \mid x_{i-k} \ldots x_{i-1})$$

$$\text{Total loss} = L_{sup} + \lambda L_{aux}$$

→ Having auxilary unsupervised training has lot of advantages as it leads to faster convergence and generalizability, However this is true for larger datasets, For smaller datasets better to reduce Auxilary task by setting $\lambda \approx 0$

→ We can use GPT for other purpose like Question-answer, Finding duplicates, MCQ solving by bringing data to common format as taken by GPT

Consider Duplicate solving

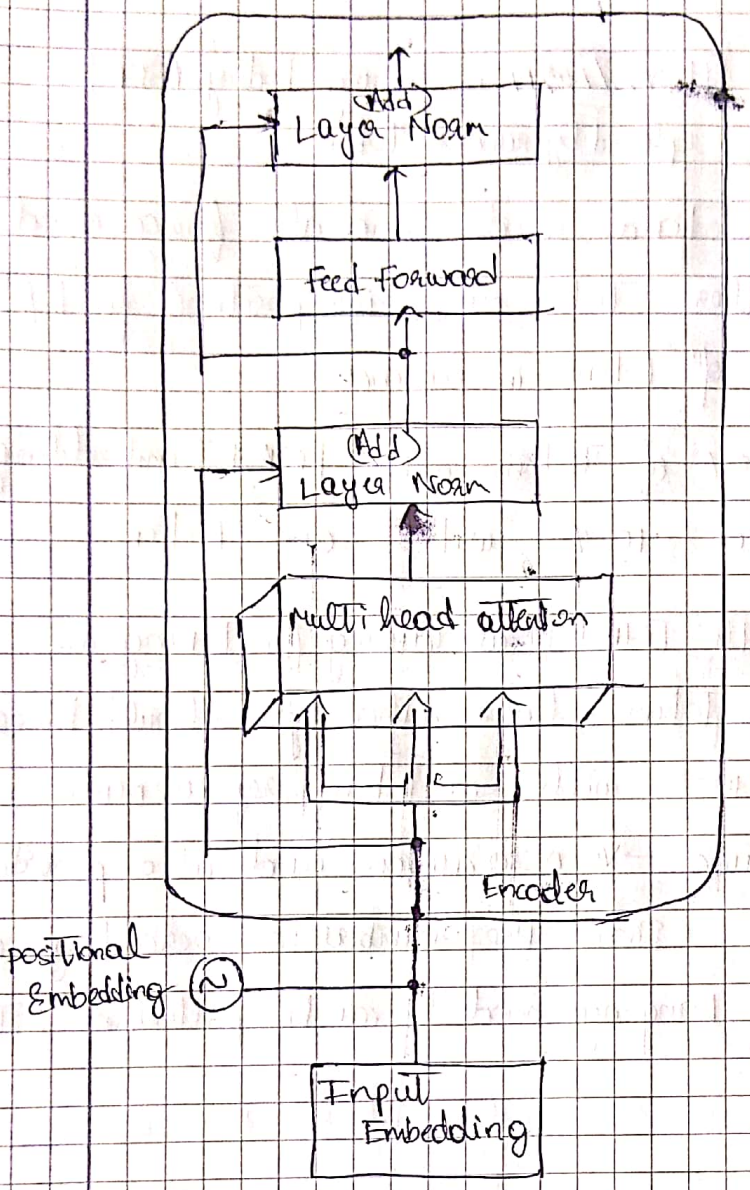


→ Since we don't have order here, we try both and specify '$' as delimiter which Transformer has to learn by itself and not specified explicitly

→ We can also remove as many layers from Transformer stack and put up our own during supervised fine tuning. But generally it's better to not remove any layer as it leads to bad performance

→ Since unsupervised is trained on variety of data like books, Tweets having extreme long & short range attention we see that model is better

BERT (Bidirectional Encoder model for Transformers)



Principle : Use encoder of Transformers to generate contextual info and embeddings

Nx **Pretraining :**

a) Masked Language model (MLM)

b) Next Sentence Prediction (NSP)

Advantages wrt GPT

→ Bidirectional context consideration during generation process

→ Pre trained using NSP provides additional boost on learning sentence relations

→ Learns [SEP], [CLS], [A/B] Tokens during pretraining itself unlike gpt.

→ Trained on additional Wikipedia corpus

Advantages of Bidirectional Training and need for MLM

→ In GPT version 1 we were predicting next word given previous words, We were not considering future context. Hence may not work in some cases as below.

→ This is especially helpful for languages where verbs may appear at end

Ex:- German ⇒ Ich mochte buch schreiben ⇒ I would like to write a book

→ This future context is also helpful in some question-answer / similar sentence prediction scenarios which GPT V1 cannot fully cater to. Hence bidirection

→ Problem with the bidirectional stuff is we would already know word after first layer as its embedding would be used for self attention calculation of other tokens. Hence learning doesn't happen. Need a novel training approach.

BERT Pretraining procedures

a) Masked Language model (MLM)

Ex:- [CLS] The sky is [MASK] and The [RANDOM] is shining Today [SEP]
masked — replaced by random token

→ To avoid seeing the token in bidirectional encoder and also force encoder to learn context at each token rather token at specific positino(first / last / mid we randomly Mask 15% of tokens in sequence.

→ Masking process :- Replace 80% of Tokens with [MASK] embedding 10% with random Token and 10% with same Token

→ We don't replace all 100% with [MASK] as during fine tuning we most likely won't use [MASK] tokens, hence other Types should be represen

→ 10% with random token and 10% with sametoken replacements helps model To learn what may cause -ve association and also provide slight +ve bias To our pretraining. Since the numbers or percentage of random replacements are less, Language model can be pretrained with relative ease.

b) Next Sentence Prediction (NSP) :-

→ Helps To contextualize relation b/wn Two sentences and long range thinking.

→ The embeddings of Two segments sentences are learnt

Ex:- [CLS] weather is good [SEP] Sky is clear [SEP]
(A) (B)

Token embedding	$E_{[CLS]}$	$E_{[weather]}$	---	$E_{[SEP]}$	---	$E_{[clear]}$	$E_{[SEP]}$
segment embedding	E_A	E_A	---	E_A	E_B	-- E_B	E_B
Position embedding	E_1	E_2	--	E_5	E_6	--	E_9

Input :- Tokens are further divided to Byte pair encodings to cover for infrequent words or new words. This also helps to standardize input vocabulary and training

Fine Tuning :- Fine Tuning using BERT is relatively simple

Single Sequences :- As similar to other models, enclosed with [CLS] & [SEP] embeddings [CLS] S_A [SEP]

Pair Sequences :- Concatenate two sentences and train them in usual manner, we don't need to encode them separately as we have learnt segment embeddings in our pretraining Ex :- [CLS] S_A [SEP] S_B [SEP]

Output :- If it's an entailment / sentiment prediction task then we can make use of [CLS] tokens representation, pipe it through a simple Linear layer and fine tune it for our output

→ Complete representations too can be feed to output layers for tasks such as Pos Tagging

Page 3

17/07/20

Language Models

Skipgram model :- Given a word predict its context

Ex:- He is a cool person ⇒

Input	Output (context)
He	(is, a)
is	(He, a, cool)
a	(He, is, cool, person)
cool	(is, a, person)
person	(a, cool)

→ Much more successful in predicting infrequent words than CBOW
→ More Training Time than CBOW
→ More helpful for embedding calculation due to context

CBOW Model :- Given contexts predict a word

→ Much success in predicting frequent words
→ Less Training Time and hidden embeddings are averaged out to get smooth distro
→ More helpful for "Fill in the blanks" type of prediction (autocomplete)

Category ~~CBOW~~	CBOW	Skipgram
Training Time	Comparitively less	More
Prediction Task	Fill in the blanks / message auto complete / (Synonyms)	Contextual rich embedding and useful for finding similar words (neighbouring concepts)
Failures	Cannot predict infrequent words Ex:- TODay was ____ a) beautiful b) good → most frequent c) delightful → rare In CBOW, we use "good" or "beautiful" than delightful even though it was correct This is due to averaging of context	Cannot do auto complete tasks and difficult to use for prediction of words. Ex:- ? ? delightful

Problems with Word2Vec models

i) Often we have a vocabulary of millions of words, hence million inputs and million outputs. This causes weight matrix to blow up and slow down computation & Training

Solutions :- Negative sampling, Subsampling, Hierarchichal softmax, phrases etc

A) Subsampling :- (waste)

Ex:- (The) quick brown fox jumped over (the) lazy dog.

→ Lots of stopwords / prepositions do exist in each sentence which don't add much to context

→ But cannot remove it all together as we need to predict these words too

→ Use a frequency based approach and decide the inclusion of word during training based on frequency. Sample these words using sampling factor

→ Lesser the sampling factor lesser are chances of word to be picked

Prob of picking the word $P(w_i) = \left(\sqrt{\frac{Z(w_i)}{\text{sampling factor}}} + 1\right) * \frac{\text{sampling factor}}{Z(w_i)}$

where $Z(w_i) = \frac{\text{frequency of occurance of word } w_i \text{ in corpus}}{\text{Total words in corpus}}$

sampling fact

=> "Quick" would appear less than "The" hence quick would mostly be used during Traing and "The" would most likely be not

lets say Z("The") = 300 Total words = 1000

Z("Quick") = 10 Total words = 1000

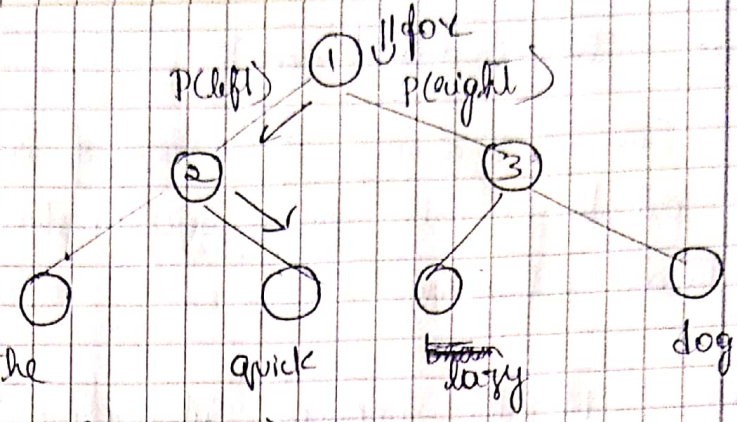
sampling factor = 10^{-3}

$P(\text{"The"}) = \left(\sqrt{\frac{300/1000}{10^{-3}}} + 1\right)\left(\frac{10^{-3}}{300/1000}\right) = \frac{1.07 * 10^{-6}}{300} = 3.39 \times 10^{-9}$

$P(\text{"Quick"}) = \left(\sqrt{\frac{10/1000}{10^{-3}}} + 1\right)\left(\frac{10^{-3}}{10/1000}\right) = 318 \times 10^{-5} = 3.17 \times 10^{-3}$

P("Quick") >> P("The")

2) Hierarchical softmax



→ Put the output words in a Tree structure so that we need to compute only O(log N) softmax probs rather than O(N)

direction → node num

P("fox") = P(left, 1) * P(right, 2) ⇒ "quick fox"

→ For input "fox" we donly go to "quick" path by computing prob of taking branch

** → Applicable only during Training as we know where to go and what is the target node. During prediction, we still need to compute softmax of all words

3) Negative sampling :- For each input target word there may be 10-20 target words based on window size, rest all are negative non required outputs

input word ⟹ 1 million output → 10 → yes
→ 999990 → no ⟹ Sample ⟹ Take only 100/1000 random example

→ Instead of complete random selection, we can attach some probability of picking a -ve word based on Term frequency

$$\text{Prob of picking -ve sample (Word i)} = \frac{[\text{freq (Word i)}]^{3/4}}{\left[\sum_{j=0}^{n} \text{freq (Word j)}\right]^{3/4}}$$

→ More frequent words like "The, in, but, a, an" etc would be picked a little less. and Infrequent -ve samples more