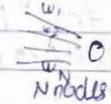
- Also in backprops^, we need each value in chain rule, so we need some memory to store our temporary results

→ Challenges in DL

- Deep learning is data hungry
  ↳ can be solved by augment^
- Overfitting / Lack of generaliz^ - solved by
  ↳ using bottleneck layer
  ↳ dropout
  ↳ generaliz^ loss (regularizˆ)
- Vanishing/exploding gradient problem
  ↳ severe for deep network
  ↳ depends on activ^ fun^ (vanishing problem) & the weight initializ^ (exploding problem).
- So, weights are chosen randomly.



., we try to initialize weights with mean=0, & variance $= \frac{1}{N}$

$\therefore \mu = 0, \sigma^2 = \frac{1}{N}$

- This is a good initializ^ technique.
- For solving vanishing problem, use ReLU, as its derivative is 1 (x>0).

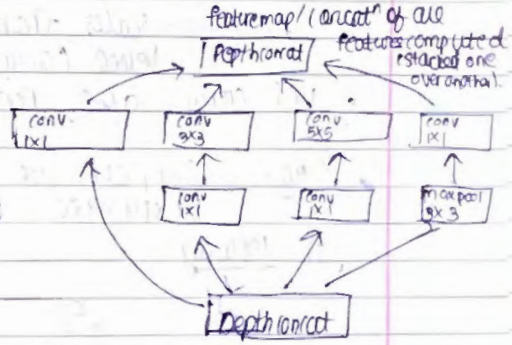
→ GoogleNet

- It has 22 layers with parameters (Conv. + FC)
- With maxpool, it has 27 layers
- There are 9 identical modules/units
  ↳ called Inception modules

- Inception modules calculate features of the network.

Inception module :-

- There are 9 such modules
- They use 1×1 filters



- The depth concat then stacks up of all the conv layers to give feature map/o o/p

→ Inception module

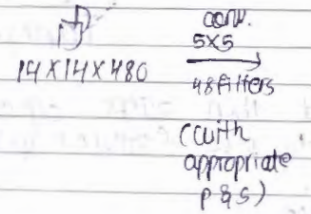
- computing 1×1, 3×3, and 5×5 convolut^s within the same module of the network.
- Network inside network.
- Covers bigger area, at the same time preserves fine resolution for small information on the images.

→

- Uses conv. kernels of different sizes in parallel from the most accurate detailing (1×1) to a bigger one (5×5).
  - 5×5 covers a big receptive field; so higher level feature detection, where 1×1 detects fine grain features.
  - From 1×1 to 3×3 to 5×5 we extract features on different scales, from minute to higher level (coarse features)
- 1×1 conv. also reduces computation

↳ eg- suppose we want to go from 14×14×480 to 14×14×48.

1) 1st way



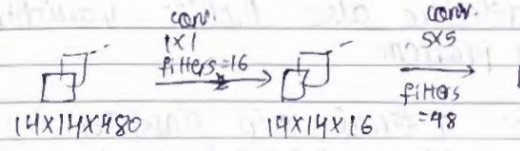
∴ total oper^n s = (14×14×48) × (5×5×480)

↓ for each o/p cell

↓ the 5×5 filter convolved over 480 channels

= 112.9 million

2) 2nd way

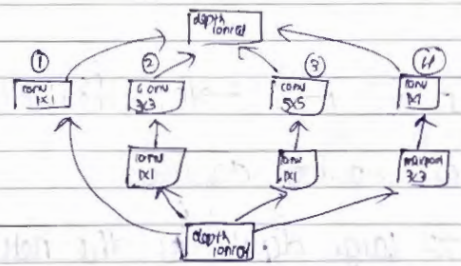


∴ total oper^n s = for 1×1 + for 5×5

= (14×14×16) × (1×1×480) + (14×14×48) × (5×5×16)

= 1.5 + 3.8 = 5.3 millions

- Thus, no. of oper^n s significantly reduced, and repr. was still preserved as 1×1 filters detected fine features



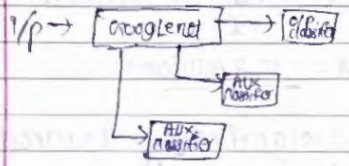
- ~~Out of~~ o/p of ①, ②, ③ & ④ are stacked along the channel dimen^n to get final o/p of module
- So, we have a multilevel feature extractor
- In the full architecture, there are 9 such inception modules

- The top 5 error rate is less than 7%.

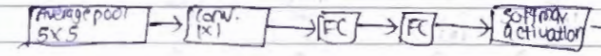
→ - The architecture also tackles vanishing gradient problem

- We have a final o/p classifier after a maxpool, FC & softmax layer.

- But we have 2 more auxilary classifiers, in between the architecture → they give Local o/p



- These aux. classifiers also have various layers:-



- There are 2 such classifiers

→ Due to large depth of the network, ability to propogate gradients back through all the layers was a concern

- Auxiliary classifiers are smaller CNNs put on top of middle inception modules

- Aux. classifiers in the middle exploits the discriminative power of the features produced by the layers in the middle

- They also give classifn error

- So, while backpropagating error, we add the 2 auxiliary losses to the total loss, ↳ injecting gradients for regularisation

∴ total loss = final classifier loss + aux.1 loss × 0.3 + aux.2 loss × 0.3

- These 2 losses are given a weightage of 0.3

- Then the model is trained.

- Aux. classifiers are used only during training time, to solve vanishing gradient problem

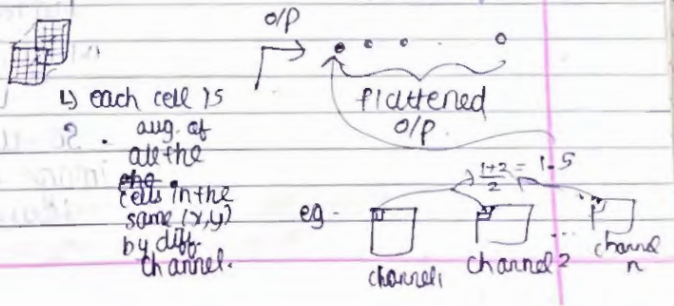
- During testing / inference time, o/p of aux classifiers are discarded & only final classifier's o/p is considered.

→ Extra points

- The model also used dropout rate of 0.7

- They also used global average pooling instead of FC layers. ↳ This reduced the no. of parameters

↳

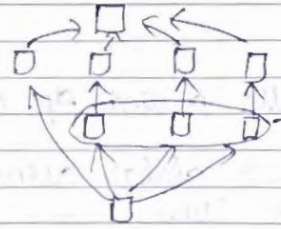


- So we had 4 types of layers:-
  1) I/p layer
  2) stem layers → like the general conv.-maxpool - conv.-maxpool-conv. .....
     (2 conv. + maxpool → reduce image size.)
     (our normal layers are called stem layers in GoogLenet)
  3) Inception modules (9 modules)
  4) Auxiliary classifiers

- Finally, avg. pooling, dropout, softmax etc.

- The multiple local networks work parallelly

* ⇒ - In inception module:-



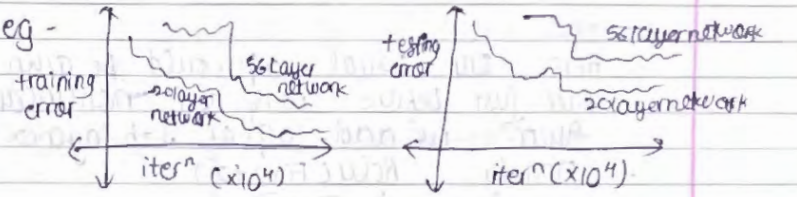
These are 1×1 filters, our bottleneck layers, who maintain image size, but reduce channels
$n \times n \times nc_1 \rightarrow n \times n \times nc_2$
where $nc_2 < nc_1$

- So, we project image from higher to lower dimension.

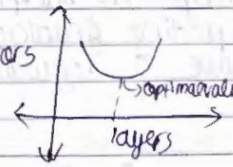
→ Resnet (2015 winner ILSVRC)

- Intuitively, we may think that deeper networks are better, as they are more complex & have more parameters to learn.
- But this is not the case
- Also, we expect that deeper network may perform better during training (as even if overfit occurs, then training accuracy ↑), but this is not the case

- eg -



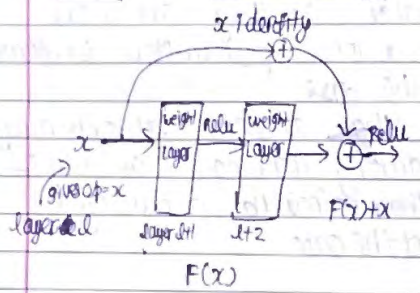
- Here, we observe that for 56 layer network, the training error is more than 20 layer network, also even for testing error is more
- So, our hypothesis that deeper networks are better fails.
- In reality,



- The failure of this hypothesis is due to vanishing & exploding gradient problem.

- So blindly ↑ layers is not good.

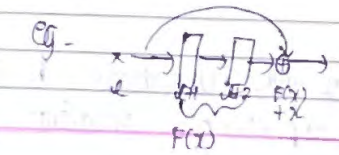
→ The ~~ResNet~~ Residual Block



- Here, our usual o/p would be given by F(x)
But just before using our nonlinear funct^n, we add o/p of lth layer(x)
∴ Finally, Relu(F(x)+x)

- ~~So effectively~~
This is a residual connection, or skip connection

- The original Resnet architecture used 152 layers
- The skip connect^s help in backprop^g (help vanishing & exploding gradient problems)
- They also ~~having~~ have a regularizing effect



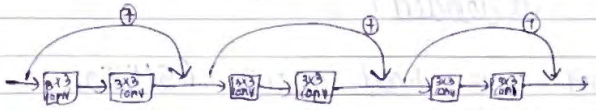
Eg.

- In F(x)+x, if we make F(x)=0, then o/p is x.

- So, the weights of l+1 & l+2 layers did not contribute to the effective o/p
- So, overfitting due to these weights was prevented.

- So, skip connections help the gradients to flow & thus makes training possible.

- Thus, with skip connections, even if the network is deep, it can behave as shallow network
- So, the 152 layer orignal layer also works



∴ H(x) = F(x)+x.
∴ F(x) = H(x)−x
we try to learn F(x)

- Also, we are not multiplying, only adding.
- So, gradients will flow properly, & no vanishing or exploding gradient problems occurs.
- Less than ~~4%~~ 3.6% error Top 5 in Imagenet Database.
- They also used bottleneck Layer (1x1 conv) to keep the no. of computations sustainable.

→

* Skip connect solve overfitting

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

Here: $H(x)$ is a complex fun^n when when take I/p, gives o/p similar to without $H(x)$

so weights must be such that they produce same I/p.

- One point needs to be taken care of -

- During back prop^g, we can skip & propagate so, no vanishing gradient.
- So, if dimens^n do not match, then addit^n not possible
- So, using various heuristics, this concern is solved.
- Also, how many layers to skip is important (According to the problem this is decided)
- #
- Resnet 2015 had about 65 Million parameters

same imp points

- We have 3 residual connect^{n}s, each with 2 filters of 3x3. Their o/ps are stacked.
- Using strides=2, the total values are maintained.

eg- $n \times n \times n_c$ —(stride=2)→ $\frac{n}{2} \times \frac{n}{2} \times 4 \times n_c$ (still $n \times n \times n_c$ values (cells))

error = Error + λ ($\sum$ weights)^2

- now, for deriv^n, we have a weight factor subtract from respective wt.
- so this is a penalty

* weight decay = kind of regulariz^n

- When overfitting, weight ~~doesn't vary~~ so, we bound weight but adding weight to loss, so they get regularized.

- No use of FC layers as it has more parameters, so Global Avg. Pooling was used
- They used bottleneck layer (1x1 conv.) if network deeper & more parameters (more than 50 layers)
- Stochastic Gradient Descent (SGD) + momentum = 0.9
  - momentum 0.9 is multiplied in training gradient (to control displacement in direct^n of velocity)
- mini batch size = 256
- use of batch normaliz^n after each conv.
- Xavier/2 weight initializ^n
- Learning rate initially = 0.1, but if valid^n error increases, the learning rate divided by 10
- Weight decay = 10^{-5} (similar to NL regularz^n ↳ the λ regularz^n const.)
- Inception V4 = GoogLenet + Resnet (↳ higher accuracy)
- VGG16 & 19 have more memory consump^n
- GoogLenet consumes less memory. (↳ less parameters)
- Resnet has less memory consump^n & high accuracy.

→ · we preserve identity of x.
· so, H(x) is a complex funct^n.

$x_1 \xrightarrow{F(x)} x_2$

$x_1 \xrightarrow{F(x)} (+x) \rightarrow x_2$

· so, we have a very complex funct^n, which nearly gives same O/p as without previous layer, without distorting much data
· This kind of funct^n are approxim^n function.

→ · No. of params do not change by adding skip connections.

· In papers diagram,

----> dotted = dimens^n change.
——> solid line = same dimens^n

· They do not use too many maxpooling
- To reduce dimens^n, use of conv. with more strides.
· filters used - 64, 128, 256, 512.
· Skip connect^n - every 2 conv. layer

· while changing dimens^n, for downsampling, we use bigger stride.
· use of bottleneck layer (1×1) conv. to downsample (eg - 56×56 to 28×28 using 1×1 conv.)
· No. of oper^n remain same
(56×64 = 28×128)
half double

Recurrent Neural Network

- It works very well with sequential data
- we need sequence modelling.

eg - speech recognit^
- Lang modelling
- machine transl^
- named entity recognit^
- sentiment classific^
- video activity analysis

→ Why RNN?

eg - NLP (natural lang. processing)
↳ text data i/p (eg - spam classifier).
↳ we can use
- BoW (bag of words), TF-IDF, Word2vec

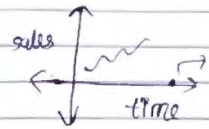
They convert text to vectors.

	word1	word2	...
sentence1	0.3	0.9	...

→ vector.

now we can use ML algo. (like Naive Bayes).
- Here, the model understands the words,
* but the sequence inform^ has been discarded
- so, accuracy may be low.
- we have seen many chatbots / voice assistant answer us in proper sequence.

eg - Time series data



- for predict^, we want the sequence of previous sales
- so, here RNN can be used (this will give better accuracy than usual ML models)

eg - google image search

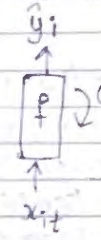
→ explain this image in text.
- needs RNN

eg - English to spanish
- RNN needed to maintain sequencing

→ If we use feed forward NN (ANN) then we need to pad the words to make a fixed i/p size.

eg - Pankaj lives in munich
Pankaj lives in munich DE
- while feeding these sentences to the ANN, we need to pad.
- when data is very large, ANN may not work properly, as context may change ahead in the sentence, and previous weights may not be useful.
- For context knowledge, we need previous o/ps, but ANN ~~da~~ or CNN doesn't store it.

Recurrent Network:-



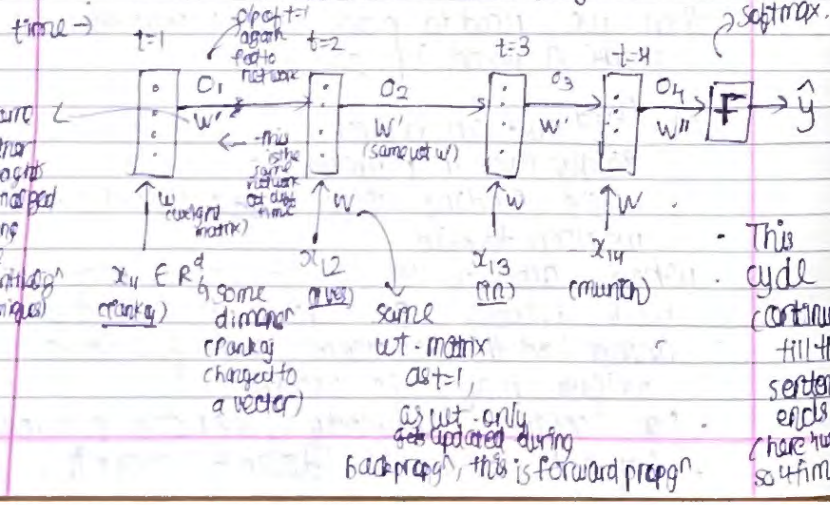
- The main idea is that we basically feed the o/p of the previous time stamp again to the neuron/ layer.

- suppose we are doing ~~serant~~ sentiment analysis

sentence$_1$ = Pankaj lives in Munich ↳ let it be sentiment = "happy"

x_{11} x_{12} x_{13} x_{14} ← 4 words

∴ The RNN works in the following way :-



- here, we need O_0 as dummy i/p, as all the RNN's have some i/p
- O_0 can be 0 padded value, or some randomly initialized values.
- ∴ $O_1 = f(x_{11} \times w + O_0 \times w')$

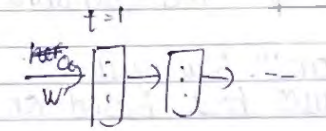
↳ activ^n funct^n

- $O_1 = f(x_{11} \times w)$
- $O_2 = f(x_{12} \times w + O_1 \times w')$ → we can see that $O_2, O_3 \& O_4$ preserves the seq. info.
- $O_3 = f(x_{13} \times w + O_2 \times w')$
- $O_4 = f(x_{14} \times w + O_3 \times w')$

∴ This o/p O_4 is then finally given to a softmax funct^n F, (O_4 multiplied with some w'') (for multiple sentiment classes like happy, sad etc.)

- Then we get find predict^n $\hat{y}$.
- we calculate loss $(\hat{y} - y)$
- Then we backpropogate this loss, to minimize it. (Also, we maintained sequence info.)
- NOTE – 1) we represented each word ~~using~~ in the vocabulary using a vector, this needs special techniques

2)



- But here the numbers are not normalized.
  - a=0, zebra=1000
    ↳ not in same range.
- so, we may use one hot encoding.

a	0
ant	1
:	
happy	25
:	
zebra	1000

→ For happy, [000...1...00] (1 word) ↳ 25th posⁿ

a=[100...0]

- Adv. of this method
- easy to use
- Disadv.
- very huge dimensⁿ for big corpus
- doesn't have any ~~en~~ meaning
  eg- I am happy
  I am glad → similar, but we can't detect these semantics.

2nd way)
- we use features
- Each all of the matrix has normalized value (in range 0 to 1 or -1 to 1 depending on applicⁿ).

eg- words = apple, mango, elephant.

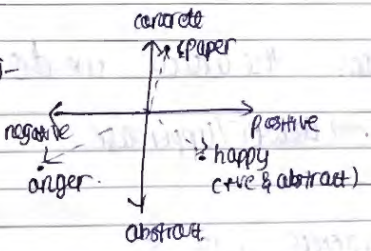
- we define some features for these words.

word / feature →	Is fruit	Is eatable	Is animal
apple	0.9	1	0.01
mango	0.92	1	0.01
elephant	0.01	0.01	0.9

- now, apple is ⇒ [0.9 1 0.01]
  ↳ this vector
- How many features to choose is a hyperparameter (usually 300 features for practical applicⁿ).
- Now, we can see that apple & mango are similar (cosine similarity)

so now if sentence ⇒ I like applejuice
I like mango juice
↳ we can predict this based on apple's & mango's similarity.

eg-



(The axis are some features)

- so, every word can be shown as a n dimensional vector (n = no. of features)

Adv. of this method

- Low dimensn
- meaning given to encoded words.

- Some ML models for word embeddings:-
  - word2vec (google, 2013)
  - continuous Bag of words (CBOW)

- DL models - GPT2, BERT, ELMO.

→ The task of creating our own embedding is self as well as unsupervised.

∴ Flow of the method:-

corpus ⟶ transformn ⟶ new ML/DL models
can be- → word → generate word vectors → use these embeddings
- general (like wikipedia)
- specialized (health, sports etc.)

NOTE- • Before embedding the words, we do some preprocessing:-
  - Lettercase → Lower/Uppercase
  - punctuatn
  - numbers
  - special characters (@, $, #)
  - especial words
    ↳ eg- :) (symbols), :happy: etc.

- Now apple → [0.9 (full) | 0.01 (minimal)].

But how do we get these values?

- How do we generate these real values / feature values.
- 2 solutions
  1) - either take readymade weight
  2) - or learn them from data.

- For learning, there are 2 methods-
  1) CBOW (Continuous Bag of words) } classical models
  2) skip gram model.

② skip gram model → predict context word, given centre word

eg- sentence = The [cat is] eating [the mouse] it caught.
(context) (centre) (context)

- we have a window that (with a window size lets say 2).
- so, we have a centre word.
- To the left side, we have a window with specified size, i.e 2, & similarly to the right we have the window.
- The window have our context words, & the centre word is in the middle.

So, the centre has the four relatns:-
(eating, the), (eating, mouse), (eating, cat), (eating, is)
- combin of centre with each window word forms a relatn.

- Choosing window size is a hyperparameter
- If window size is too large, then too many rel's which may not be proper.
- If too small, then we may miss out imp rel's
- Usually window size is 2-5.

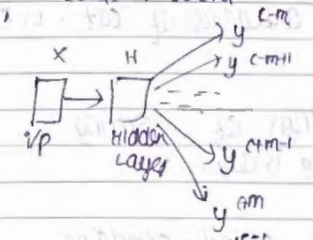
→ The entire corpus is our raw data.
- From that, we generate training dataset for ML model to learn embeddings.

eg- corpus = The cat is eating the mouse. It caught.
(centre 1st time; context words)
- next time, cat becomes centre, & "the", "is eating" becomes corpus
- So, we keep on sliding this window. & we get rel's.
  eg- 1st time, rel's:- (the, cat), (the, is), & similarly, we do for entire corpus, from 1st to last words
- These rel's are our training examples.
- We can see that we can learn rel's through this, eg- eating & mouse are related, mouse & caught are related, etc

[• Corpus is usually very large (billions of words)]

Training process → we feed centre as i/p word

- Let the 1st example / rel^n generated be (the, cat)
- Here "the" is the centre word, & "cat" is the context word.

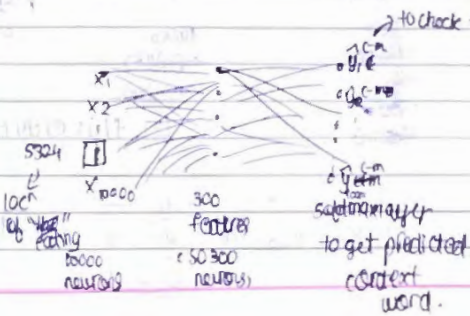


m = window size (here 2)

→ The cat is eating the mouse it caught.
(centre)

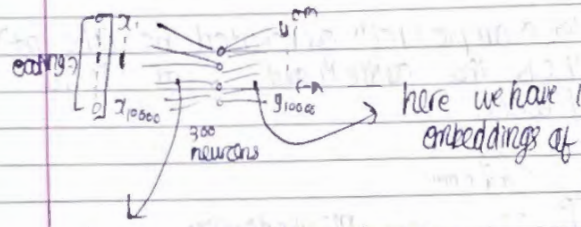
↳ If window (m=2), then we get 2m (context words)

eg- when eating is centre,
- So, when we feed y^c as "the", we want o/p as y^{c+1} to cat (in that case)
- Now, again, we can't directly feed eating or cat or the, so we use 1 hot encoding.
- So, for 10000 words in vocab, we create 1 hot vector for the & cat. eating & cat



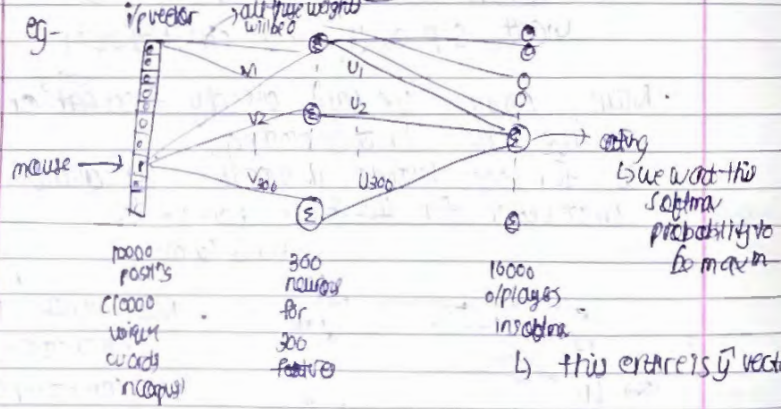
- Now we have actual 1 hot encoding of cat.
- We can compare it with this y^{c+1} vector (of same dimens) to calculate loss.

**. for each example, we learn 2 word embeddings



- So, we learn centre & context embeddings
- So, for every word, we will learn 2 embeddings-
  - when it is centre
  - when it is context
- Let training (mouse, eating) → so we learn embedding only for centre.
  (centre, context)

eg-



Extra → prob of eating = $[v_1 v_2 \dots v_{300}] [u_1 u_2 \dots u_{300}] = v^T u$

(1)

CBOW (continuous bag of words) → predict centre word given context words

- Alternative to skipgram
- Predict centre word given context word window.
- Now we are given context words, we find prob. of centre words

eg- The cat is [eating the] mouse [it caught].
context centre context

- We had 2m context words for window m=2
- But here the 4 words "eating the it caught" is treated as continuous - contn. bag of words.

∴ training example → ([eating, the, it, caught], mouse)

order of contn. words doesn't matter,
i.e. (the, it, eating, caught), mouse) is also valid (& all other combn's)

I = i/p
O = o/p

∴ prob. of eating by softmax = $\frac{e^{v_{w_I}^T u_{w_O}}}{\sum_{w=1}^{10000} e^{v_{w_I}^T u_w}}$ → word is here mouse; words eating

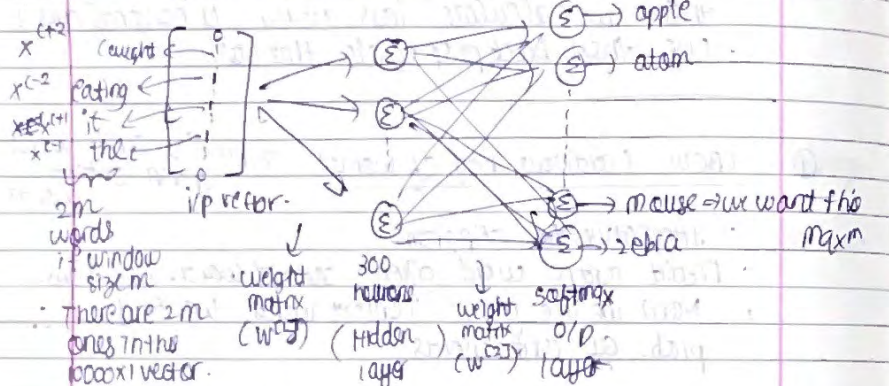
$= P(w_O / w_I)$
↳ This is for 1 pair

- For next training example, we repeat this.
- Also, if prob of eating is not max^m, then we calculate loss using y original ans.
- We then back propogate this loss.

∴ i/p vector:

caught = $\begin{bmatrix}\vdots\\0\\\vdots\end{bmatrix}$, eating = $\begin{bmatrix}\vdots\\0\\\vdots\end{bmatrix}$, it = $\begin{bmatrix}\vdots\\0\\\vdots\end{bmatrix}$, the = $\begin{bmatrix}\vdots\\0\\\vdots\end{bmatrix}$

one hot ~~enc~~ encoding of words
- now we add them.



2m words i.e window size m

- There are 2m ones in the 10000×1 vector.

→ In this softmax o/p, we want probability of "mouse" to be the highest.

∴ $V^{c+2} = W^{[1]} X^{c+2}$

↳ 300×1

vector for word x^{c+2} (1 hot encoded)

as $W^{[1]}$ is 300×10000

$x^{(c+2)}$ is 10000×1

$\therefore W^{[1]} x^{(c+2)}$: 300×1

similarly, $V^{c+1} = W^{[1]} \cdot x^{c+1}$

$V^{c-1} = W^{[1]} \cdot x^{c-1}$

$V^{c-2} = W^{[1]} \cdot x^{c-2}$

net $\hat{V} = \dfrac{V^{c-m} + V^{c-m+1} + \dots + V^{c+m-1} + V^{c+m}}{2m}$

$= \hat{V}$; 300×1

↳ i/p context word vector.

⇒ This is net i/p

- Now for final layer, we get o/p of 10000×1, where we want mouse to get max^m value.

$\therefore z = W^{[2]} \cdot \hat{V} = V_c'$

(z → final o/p; $W^{[2]}$ → 10000×300; $\hat{V}$ → 300×1)

= 10000×1 → o/p

- we want softmax of z to have max^m value of mouse
  ↳ in those 10000 values

⇒ so, we want to maximize →

Prob ⇒ P(C | $w_{c-m}, w_{c-m+1} \dots, w_{c-1}, w_{c+1}, w_{c+m}$)

C → center word

↳ given these context words (2m words)

∴ maximize this

⇒ or maximize log of the

⇒ or minimize $-\log(P(w_c / w_{c-m}, \dots w_{c+m})$

↳ this is our objective funct.

$$J = -\log P(v'_c / \hat{v})$$

o/p ↳ i/p

- minimize J
- This is as for 1 training example
- we do this for all training examples.

NOTE → • In both the models we saw, the embeddings from i/p layer to hidden layer give us our word embeddings.