#RNN

- For text based problem,

→ First text processing (text to vector)

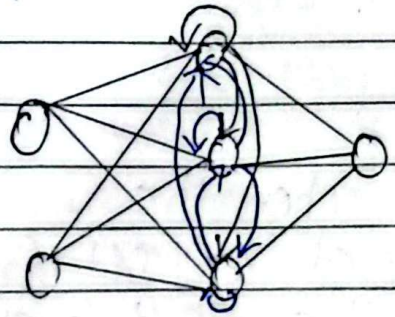
→ Even after converting to vector, the content or relation of word with other word is not possible.

We need different architecture.

i-e

RNN (Re-current NN)

→ We've [feedback loop] the output from any neuron is sent to the neuron itself as input and to others as well of that layer



We've time stamp and base on time stamp we pass each word (vector)

"The food is good"

x_{11} x_{12} x_{13} x_{14}

(*) Representation

Note: In ANN, we feed all at once,
but here, sequence matters so Data can't be fed in sequence one by one.

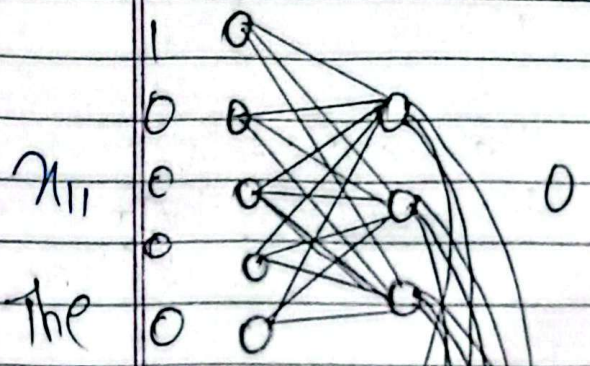
So, every other word has a content of other previous word because we're passing the output to itself as input.

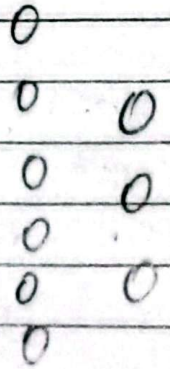
Forward Propagation in RNN

"the Food is Good" / "Bad"

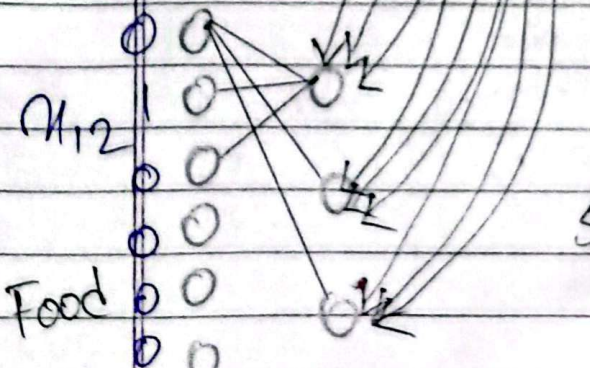
[[1, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 0, 0]]

For t = 1 i.e 1st sentence





t = 2



So, when the first word is passed content is preserved for the next word [memory] using feedback loop.

So, when the second word arrives it would have, every neuron would have 9 inputs (6 input) + 3 content

This way every word would have content of each word.

Also, every feedback loop or input (context) will also have weight.

(*) Maths (vertical, same layer)

$O_1 = f(x_{11} \cdot w + b_1)$

$O_2 = f(x_{12} \cdot w + O_1 \cdot w_h' + b_2)$

Here, O_1 is the $\sigma(z)$ and w' is the weights for the feedback loop content input.

~~$O_3 = f(x_{13} \cdot w$~~

$O_3 = f(x_{13} \cdot w + O_2 \cdot w_h' + b_1)$

$\hat{y} \Rightarrow \sigma(O_3 W_f)$ $L = -y \log \hat{y} - (1-y) \log(1-\hat{y})$

(*) Example

Review { sentiment

Movie was good } 1

Movie was bad } 0

was not good

movie was good bad not

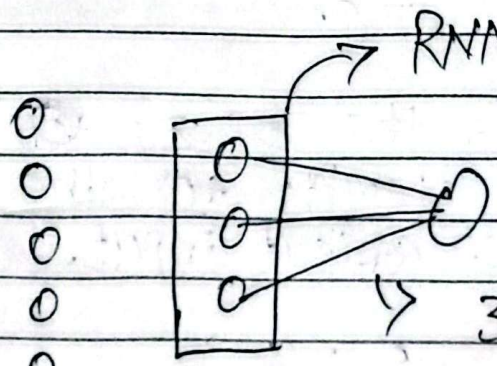
$R_1 \Rightarrow$ [[10000] [01000] [00100] [00010] [00001]]

→ Feed on the basis of timestamp. Single word as input.

• Activation for RNN is Tanh

Note:
(i) For the first timestamp random value are sent as [feedback loop input]
(ii) From second timestamp [memory content] starts to pile up.

(*) Trainable Param for SimpleRNN



↳ 3+1 => 4

↳ $W_2 = 3 \times 3$ because each neuron accepts entire 3 [feedback loop input]

$W_1 = 5 \times 3$
+3
i.e 18

-> Each neuron in the [H Layer] of RNN will receive [5+3] inputs i.e 5 from natural input, 3 from feedback.

(*) Code

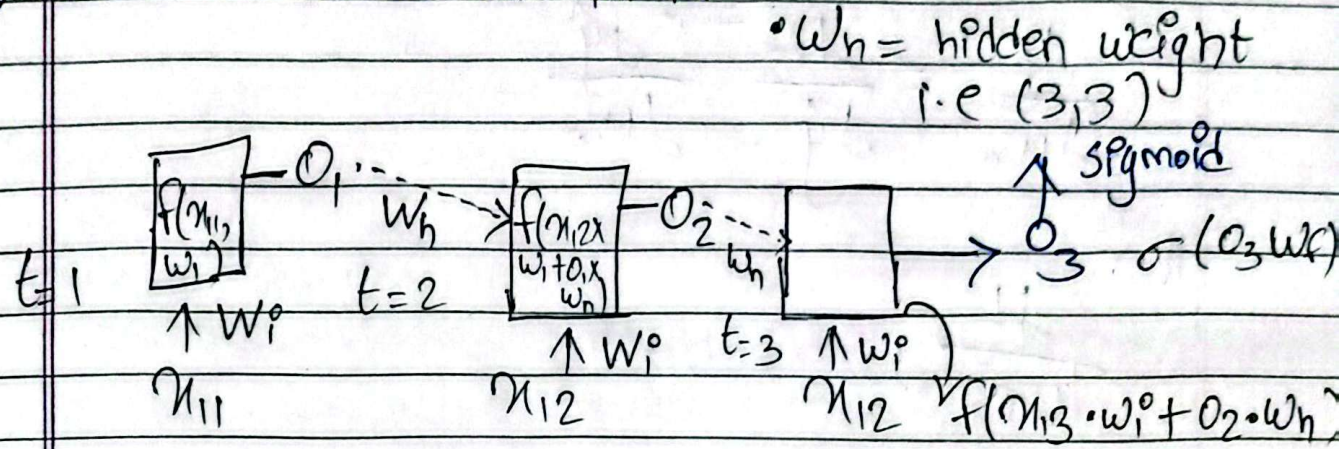
```
tf.keras.layers import Input, Dense, SimpleRNN

Sequential().add(SimpleRNN(3, input_shape=(4,5)).add(Dense(1, activation='sigmoid'))
```

Total params = 31

• Until the words in the sentence are not completed the feedback loop continues and calculate the output containing content for each words.

Feed Foward in RNN



so, for each sentence only after the words in sentences are finished the algo moves to the next layer.

Back Prop in RNN

- Calculate the loss based on type of problem.
- Gradient descent with the optimizer

i.e

$$W_{new} = W_0 - \eta \frac{\partial L}{\partial W_0}$$

so,

$$W_i^o = W_i = \eta \frac{\partial L}{\partial W_i}$$

$$W_h = W_h - \frac{\eta \partial L}{\partial W_h}$$

$$W_o = W_o - \frac{\eta \partial L}{\partial W_o}$$

→ derivatives

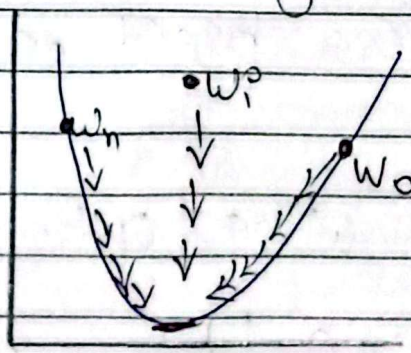
First,

$$\frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_0}$$

Then, $\frac{\partial L}{\partial w_1}$

Then, $\frac{\partial L}{\partial w_n}$

So, update will happen on all three kinds of weights.



Problem with RNN

- Suitable for sequential data [text, time series]

→ Major Problems with RNNs.

→ Long term dependency
If the timestamp is huge and

the dependency are too far (start and end of the word) in such case it fails to capture or forgets the dependency.

So, Long term dependency is not captured

→ the Gradients would be very small and negligible, they would vanish.
so, no weight update for [Feedback weights] or Hidden weight.

→ Chain Rule is very big causes vanishing gradient.

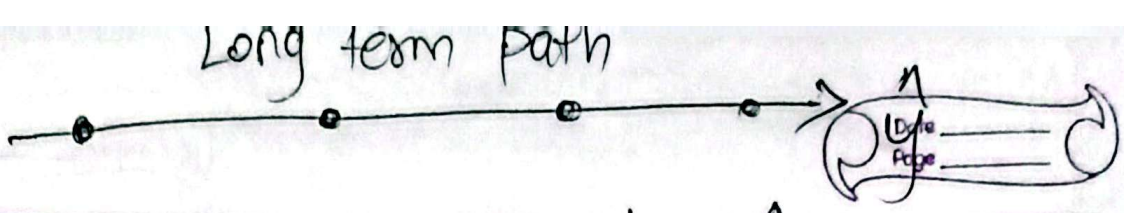
Sol^{n}:
→ Use Relu, Leky RELU
→ Initialize Uniform weight
→ LSTM

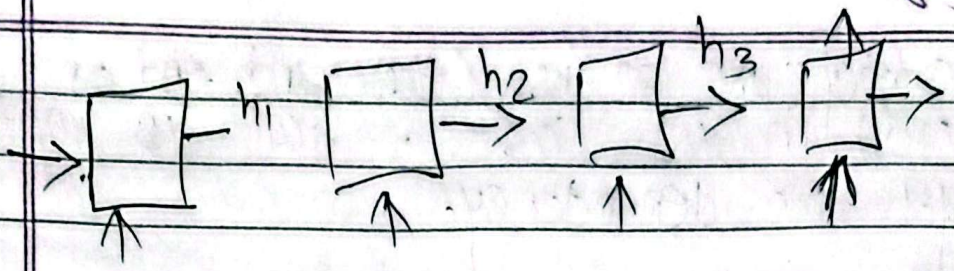
LSTM (Long Short Term RNN)

- RNN is not able to solve long term Dependencies. (Vanishing Gradient Problem)

→ LSTM has Both Short Term i.e. [Simple RNN] and also long term memory.

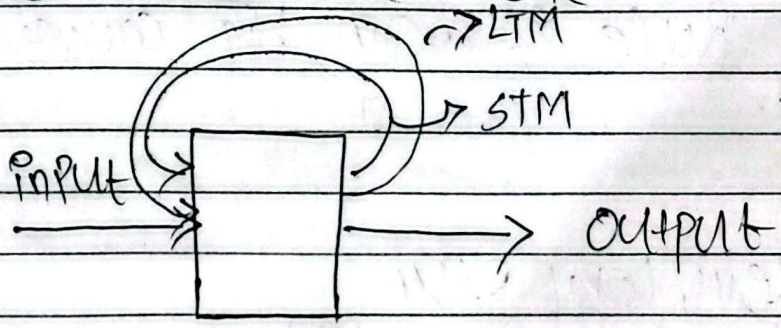
It has a memory cell which keeps track of important content and can add and remove content as required.





→ If the content is important for long term it is carried with it.
→ Content is update as new important content arrives.

→ If the content is not removed it make it to the end.



So,

- We carry two states [each neuron carries two states]
  → These two states needs to communicate as well.

→ Inside LSTM, these are 3 Gates

(i) Forget Gate
(ii) Input Gate
(iii) output Gate

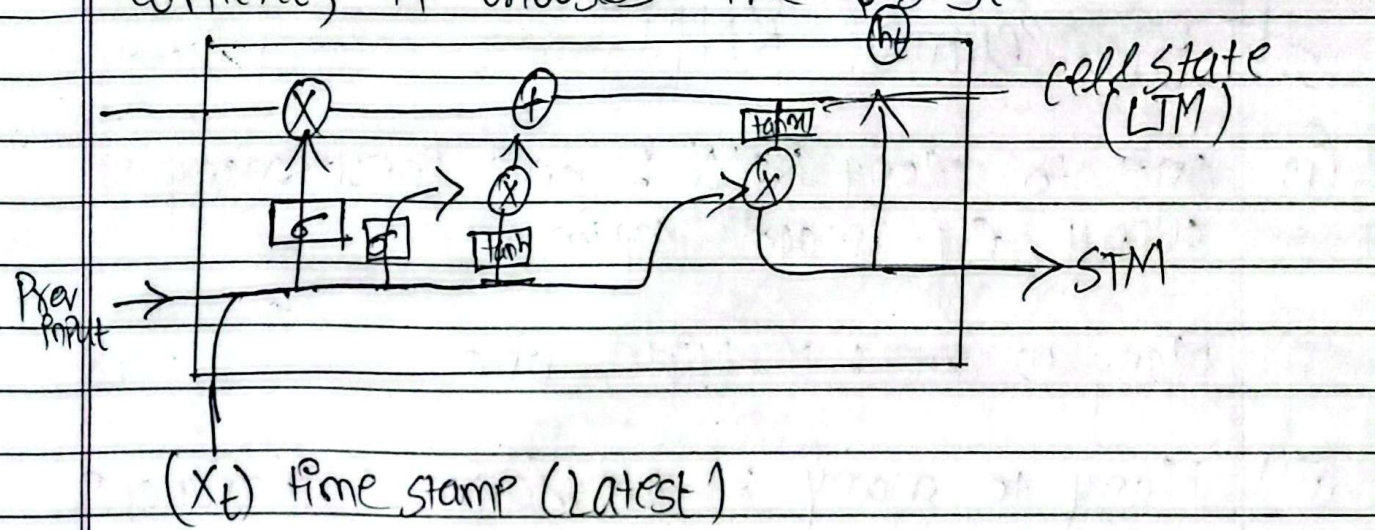
- Forget Gate
  Based on current input timestamp and feedback input, it removes the content from the [LTM]

- Input Gate

→ Based on current input decides what to include new info in the [LTM]

- Output Gate

→ Based on current input decide what to take from the long term memory and give output. Also, creates STM for next. There might be multiple [LTM] content, it chooses the best.



(*) GRU (Gated Recurrent RNN)

- variant of LSTM RNN

→ In LSTM we've LTM and STM and for every gate we've weight.

→ LSTM archi is complex.

→ Because of complex ~~Para~~ Architecture the trainable params increase and resulting in long time to train

So, in GRU we've only single Memory cell.

→ This memory cell has both LTM and STM. (Single cell)
→ Reduces the parameters

→ Reset and update gate takes the content from one timestamp to another.

Bidirectional RNN

(i) one to Many RNN : one input many output e.g Image caption

(ii) Many to one : Multiple, one

(iii) Many to many : Translation of language

(iv) One to one

→ If we need the content of prev and coming timestamp.

Give seq, Get seq,

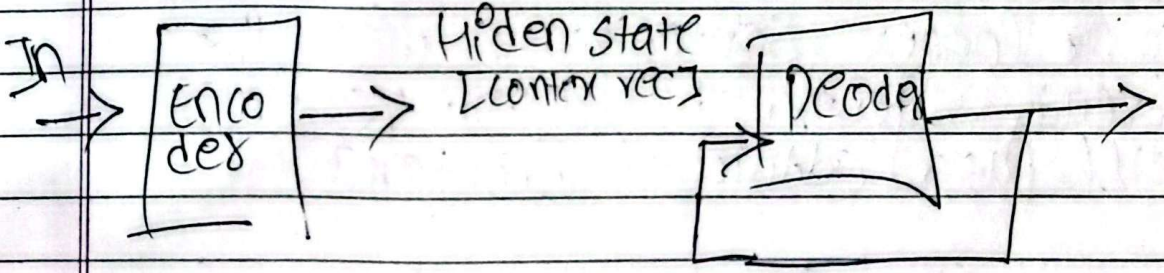
Encoder Decoder (seq to seq model)

- For multiple inputs we want multiple output i.e Recommender sys e.g LinkedIn text msg.

(X) Architecture

→ Input → Embedding layer [Encoding] → create a hidden state [context vector] → Decoder layer

Here, Encoder ; encodes i.e embedding
Then create hidden state [context vector]
The Decoder then creates the output word by word, and keeps feeding the previous word into the decoder again



- Encoder generates the hidden state [context vector]

→ Inside the encoder there is [LSTM] layer

- The LTM and STM of LSTM is called content vector

→ In decoder as well there's LSTM. Gives softmax probability for the word.

(*) Problems with Encoder Decoder

→ Conten vector will have with more info about words that are passed recent timestamp. (nearest)

→ If sentence is bigger the first timestamps will have very less content in the content vector.

→ For longer sentences it does not work work.

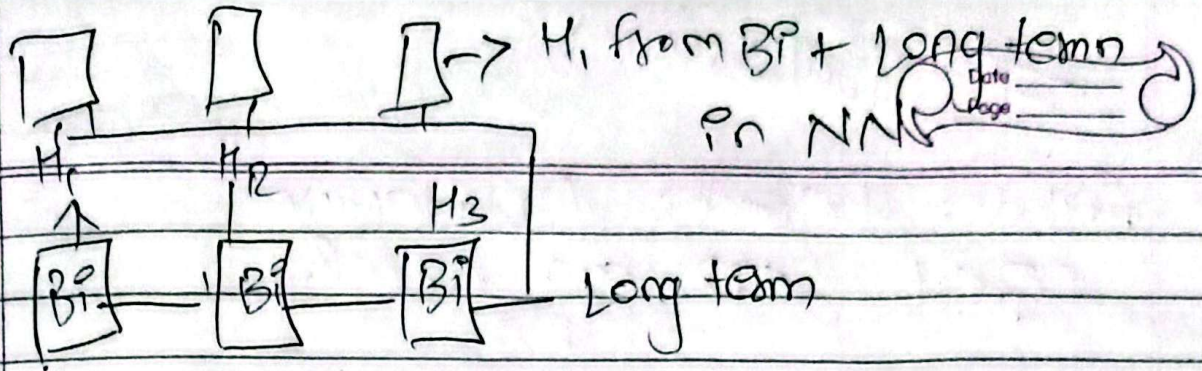
- To fix this we've Attention Mechanism [seq2seq mechanism]

→ For longer sentences, with the already generated [content vector] we'll pass additional [content]

→ Some changes to the encoder-decoder should be done.

Attention Mechanism

- In the ~~decoder~~ encoder instead of simple LSTM, use Bidirectional LSTM (capture content from both side of timestamp)



→ Longterm + each hidden output will be feed to Neural Network.

- output of each LSTM combined with Long Term Memory to ANN (softmax)

The output from the encoder is Attention score
↳ Relevance of each word with all the other words.
↳ How much content of each word we should consider.

This is called Context vector.

We pass context vector + Long term context to the each LSTM of decoder.

[Hi am good]