How human mind processes the images?

* Light from retina goes through optical nerves (cells) to the thalamus and from there to the V1. Electrochemical single is the method through which light moves thorough the nerves.
* Main takeaway is different cells process different aspects

CNN: Architecture is as such where you have Image arrays as input and then you apply filters on it and do the convolutional process. (with padding, strides or without them). And then add pooling to address few issues raised such as

* Reduced array shape
* More impact to the centered pixels
* Overfitting
* Computational efficiency.
* Memory issue
* And translational invariance.

How does the convolutional operation happen:

* 2D: A black and white image is applied with the filter and gets you reduced 2D array with cross multiplying each corresponding element and then adding all the products (dot product).
* For 3D you do the same but with 3D filter but the output you get is 2D array only after every entire operation. Cause the 3D convolutional operation happens with the filter having the same depth as your input and you don’t stride/move your filter backwards but you only move them in 2D and i.e. you get 2D array output as a **FEATURE MAP.** Size of the feature map is always a **2D array X no.of filters.**

Trainable parameters: What you calculate weights and biases rights. (Keep the same principle here as well):kk

What could be the trainable parameter?

Feature maps does look like a trainable parameter but it is not is it?, Those arrays are something similar to the output the ANN. Not something but kind of exactly like the output of the ANN

And then what is similar to the weights?

* Filters are what we want to create and developed for a particular image cause those are the arrays or rather values which are getting applied on our input and producing output right. So, we have to decide those values and those exact values are nothing but the comparatives of the weights.
* And the biases of each filter

So, trainable parameters are size of the ( filter array \* No. of. Filters ) + biases.

Main thing about the CNN that trainable parameters do not depends upon input parameters at all which was the case in ANN that was creating the high computational complexities and memorization error.

Padding and Strides :

Pooling :

* Pooling solves two problems
* 1. Memory Issue:
* 2. Translation Invariance ( Only for minor translation chanes)

**RNN: Recurrent Neural Network**

How is the input given to the RNN, now as we have seen that we cannot give zero padded input like even if we try ANN.

So input is given to the RNN in timestamp, and for each record (row) the no. of timestamp is as equal to the no. of words in the record.

First of all, let’s understand how do we encode the data?

Record: My name is BMF.

The ultra ultra strong MF.

No. of unique words = 8

my = [1,0,0,0,0,0,0,0]

name = [0,1,0,0,0,0,0,0]

is = [0,0,1,0,0,0,0,0]

and likewise…….

Now the record would be represented as 2D array of [4 X 8] each row is word of 1 X 8.

And the final input to the RNN would be a 3D tensor of [5 X 8 X 2].’

So words are technically arrange in a stacked manner one top of another. First one being the first row second being the second layer and similarly. i.e. why max 5 rows as max 5 words in a single record is there that is why.

5 for max no. of word in all the records, that means whenever we have a word less than the max of words in a row there will be a empty row for that record.

Depth 2 is for the no. of the records.

And third dimension is no. of records.

This is how the tensor is given as the input to the RNN.

Now let’s see how each input is feed as a timestamp to the RNN.

For row one, there would be 4 timestamp and first would word “My” with it’s encoding and let’s say we’ve RNN architecture as such as

input timestamp[ for word “My” 8 X 1] followed by 3 RNN nodes and followed by 1 fully connected node.

So when for the first timestamp the “My’ enters with 8 inputs of [1,0,0,0,0,0,0,0] and connects to the RNN layer nodes.

Extra thing here is the output of the first-time timestamp [01] will be fed back to the nodes.

And then the processed value is sent to the next node.

Before the formulae of forward propagation let’s calculate the no. of trainable parameters, here.

W1 = 8 X 3

W1R = 3 X 3

W2 = 3

Biase1 = 3

Biase2 = 1

Total = 40

Formula is

(W1 . X11) + (W1R . O1) + b1

W1 = [3 X 8] . X11=[8 X 1] + W1R=[3 X 3] . [3 X 1] + [3 X 1] = [3 X 1]

One thing for the first RNN layer as we don’t have any of the outputs, we select either zero-zero or random outputs. And this is how we calculate forward propagation.

How do

Architecture Diagram?

In RNN we have hidden recurrent layer which gives back the output of the first input as a second input along with weight for second timestamp.

Backpropagation is very similar to the ANN, nothing sort of complicated.

Encoding techniques :

OHE we represent a word by a vector and sends the 3D tensor as a input to the RNN.

Where as in

**Types of RNN:**

**Issues in RNNs :**

**There are majorly two issues in RNN’s:**

1. **Vanishing gradient descent. Or inability to capture longer term dependencies.**
2. **Exploding gradient problem.**
3. **Vanishing gradient:** It is the main problem lying under which is inability to capture the long-term dependencies.

* Usually for longer timestamps, when you update the weight by gradient descent those values are interdependent on all the previous timestamp derivative values. In the above scenario if you have if you the derivative values to in between 0-1 and tanh is always in between 0-1 then the gradient comes to be very small and that implies the impact of that timestamp would be very less.
* Solution:
* Activation (relu, leaky relu)
* Bette weight unitization
* LSTM

1. **Exploding gradient:** It is the main problem where the gradient values are way more consistently and therefore those values comes out be very high for later stamps.

* What is the solution:
* 1. LSTM
* Tanh
* Clipping

**LSTM’s Core Idea:**

**LTM,STM and interaction between both.**

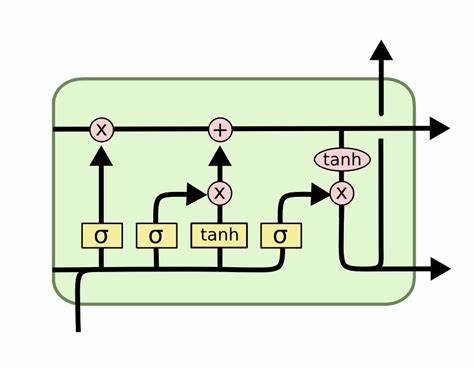
As the main reason that RNN’s cannot keep track of the old context, we must make sure somehow that RNN’s keep track of the longer-term dependencies. That is exactly what is done in LSTM’s. So, what does LSTM’s do is, along with short term dependencies like in normal RNN’s they also keep track of the long-term dependencies. So, in LSTM we have one **LTM** (Longer term memory) which track the long term meaning/memory/context and **STM** (short term memory) tracks the shorter term memory.

So now in LSTM instead of one STM recurrent state we have STM as well as LTM inputs feedback to the nodes of the layers.

In LSTM we have three gates and two operations:

Operation 1: Forget things from cell state and add things in cell state.

Operation 2: Calculates the hidden state.



1. Forget gate.
2. Input gate.
3. Output gate.

Cell state is LTM, and hidden state is STM:

So, every state we either forgot something from the LTM or add something to the Cell state (LTM),

Whereas updating the STM.

**How does the everything looks like mathematically in LSTM.**

Cell state, hidden state looks same where they are vectors of both same sizes. **(Ct, Ht).**

**Like the above two there are four more vectors like Ft, It, Ct, Ot and interestingly they are of same shape as Ct, Ht. i.e. means these six vectors are of same shape**

**Xt –** input vector ( can be of any length not restricted to any).

**Pointwise operations :**

Arrays pointwise operations like (+ , X)

**Nodes:** Each gate array is neural layers with activation function. It is a hyperparameter where the no. of nodes are decided and they are same for all the layers. The no. of nodes are same as the no. of elements in those arrays like LTM, STM, Ft etc.

LTM and STM are of same shape at both the states but the input Xt can be different.

**Forget GATE:**

Let’s suppose we have [1 X 4] input in Xt with 3 nodes in forget gate layer, in that case

The total input to the forget fate layer would be (Xt and ht) right which is called concatenation input, in this case that would be (4 + 3) which would be [1 X 7]

And weight matrix would be (3 X 7) as there are 3 nodes in the layer.

So the output of the forget gate would be:

= (3 X 7)

= (3 X1)

= (4 X 1)

Then the output of this operation would have point wise operation with Ct. So, basically what is happening is we have previous hidden state on processing that hidden state with input of current state we get Ft which is nothing but what to remove from the cell state and that is why we do point wise operation between the Ft which now knows what to remove from the cell state.

Let’s suppose our cell state is [ 5,10,20] and our Ft comes out be [0.5,0.25,0.1] then current cell state after removal would be [2.5,2.5,2] and this is what happens in the forget gate.

**Output GATE :**

On magnifying the gate we we’ll two layers in output gate first one is input layer with It array and other one is candidate cell state.

Ct is nothing but the potential information to be added in the cell state whereas It decides what and which of that potential info to actually add. It has sigmoid loss function whereas Ct has tanh.

The output equations look the same as forget gate with one change in error function.

For candidate cell state it looks like:

Though the product of weight and input state is same but the values are different.

And similarly for **It:**

On calculating the It and Ct we have multiplicative point wise operation which decides the which information out of the potential info is needed to be added in the cell state.

This quantity ( Ct X It ) which is the actual filtered info to be added into the cell state is added in the cell state with point wise addition operation.

So, ultimately LSTM has the ability to add and maintain the whichever pervious info with the help of this gates. So, the main problem in the RNN which was vanishing gradient or loosing info over the longer distance can be easily fixed in the above case with Ft being one that is nothing is removed from the cell state and the product of Ct and it is zero then the actual cell state is maintained and carried throughout without loosing the info.

**Output Gate :**

Output gate calculates the hidden state from the CELL STATE.

The output hidden layer has equal number of neurons as all the hidden layers with activation function as sigmoid.

Output array is calculated form the hidden state input ht and Xt concatenation with its weight matrix and the multiplicative point wise operation is done with the cell state array after applying tanh activation function. And that gives you the new cell state which is basically calculated from the output cell.

The equations are exactly the same there is not much change as such. Just the direction is different.

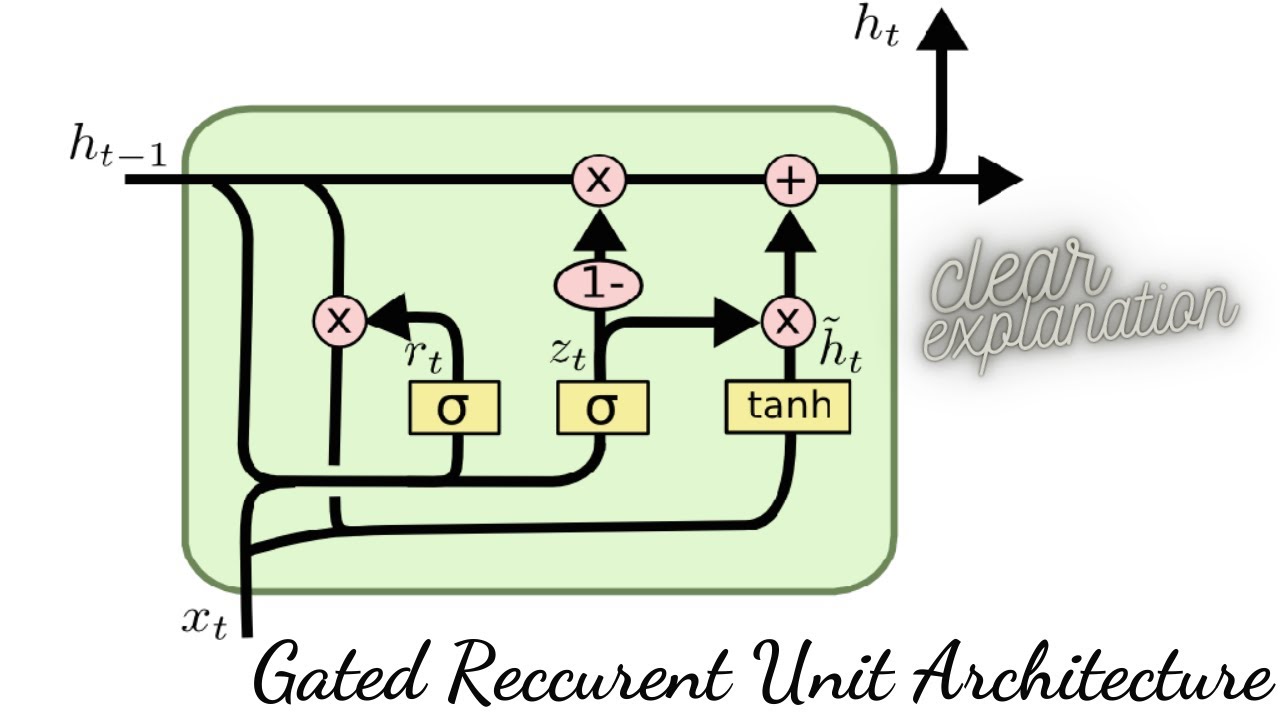
ADD the fucking diagram here.

**GATED Recurrent Unit (GRU) :**

GRU’s does the same work as LSTM’s but in more efficient way. GRU’s has much less trainable parameters compared to LSTM’s but their performance is comparable to the LSTM’s. Infect in some cases the GRU’s Outperform the LSTM’s.

GRU’s has only two gates compared to the 3 Gates in the LSTM’s and on top of that GRU’s don’t need any cell state to maintain the long term context it works with the same hidden state.

Setup:



Two gates:

1. Restore Gate (Rt) :
2. Update Gate (Zt) :
3. Candidate hidden state () :
4. Input state (Xt) :
5. Previous hidden state(ht-1) :
6. Current hidden state (ht) :

The shape of all the above arrays are nothing but the same except for the Xt whose shape depends on input state where as the other shape is same as the no. of neurons in the hidden layers.

Input (Xt):

Input state is a vector based on the vectorization. (OHE, BOW, Embeddings).

BAT CAT RAT

RAT RAT CAT

CAT BAT BAT

1. [[1,0,0],[0,1,0],[0,0,1]]
2. [[0,0,1],[0,0,1].[0,1,0]]

So, the input (Xt) would be a vector.

Architecture:

In GRU’s what’s happening is you have input Xt and ht-1 from these two we are calculating the ht.

The entire work of calculating the ht from ht-1 is divided into four steps:

1. Calculating the reset gate(Rt)
2. Calculating the () (candidate hidden state)
3. Calculating the (Zt).
4. Calculating the current hidden state (ht).

Flow of the GRU architecture.

1. We calculate the ht from the ht-1 and with the help of the update gate (Zt).
2. Whereas we calculate the from the input Xt and ht with the help of the reset gate.

The process of the GRU happens in two steps:

1. **First you calculate the candidate** from the current input which basically has the essence of the current state information. This has the potential to be the final ht, or at least some portion of which will become or influence the ht. If we directly take the candidate state as the ht then it will be normal RNN only and we’ll lose the sense of the previous info. So instead of totally replacing the previous ht with candidate memory we’ve add the appropriate extent of it or remove so that it has sense of past and present like LSTMs. So, candidate is nothing but the current hidden state. (RESET GATE).
2. **On calculating the current hidden state from hidden state** we have update the required portion of that state which is candidate state to a previous hidden state in proportionate amount decided by update gate. **(UPDATE GATE)**

Let’s take an example of the story of the Vikram.

Vikram fought kali.

Vikram lost and died.

Vikram son was brave as Vikram.

He fought and lost to kali.

Vikram’s grandson was not as might as previous kings but he was smart.

He fought kali, was losing initially and but eventually beat kali with his smart strategies and won.

So, during all this story ht would look like:

Let’s suppose ht store contexts like [ power, conflict, tragedy, revenge] .

1. [0.7, 0.8, 0.1, 0]
2. [0.5, 0.5, 0.9, 0]
3. [0.8, 0.3, 0.3, 0]
4. [0.7, 0.8, 0.9,0]
5. [0.8, 0.5, 0.4, 0]
6. [0.7, 0.9, 0.1, 1]

This is how the ht gets updated with respect to the current state on top of the previous ht’s.

How does the calculation flow occurs:

Put the screenshot from the notes.

But anyways, we calculate Rt from the Xt and ht-1, with the help of the Rt X ht-1 and Xt we calculate the candidate state .

From ht-1 and Xt we have Zt and finally form Zt,ht-1,and we calculate the ht.

So all we have to calculate is Rt and Zt.

Reset Gate (Rt):

Representation [ 0.1, 0.5, 0.9, 1]. It is an array with numbers from 0 - 1.

It is a gate which reset the previous hidden state to a proportionate degree.

**How do you calculate Reset Gate(Array):**

* You have neural layer with sigmoid as an activation function.

Lets suppose you have a neural layer with 3 neurons and input state is 4 element array.

Then the concatenation input with input Xt and ht-1 would be [1 X 7 ] and Wr = [ 7 X 3 ] so the output would be [ 1 X3 ] reset gate array after adding bias and putting everything into activation function.

Ht-1 = [ 0.8, 0.6, 0.9, 0.4]

Rt = [ 0.5, 1, 0.66, .8]

With the help of the reset gate and input (Xt) you calculate the candidate array with pointwise multiplicative operation that gets you **modularize hidden state.**

= [ 0.4, 0.6, 0.6, 0.32]

Now this output along with the input (Xt) when put into a neural layer of tanh activation function we get candidate hidden state.

**Update Gate :**

Intuition: Zt decides which state to give more importance to current potential ht which is candidate hidden state or the previous hidden state in formation of the current hidden state.

If we have high Zt then more importance is given to the current hidden candidate state and if the Zt is low then more importance is given to the previous hidden state.

So, Zt decides which state to give more importance to in calculating the current ht. where as Rt decides which factors are more important from the current state which can be made as potential candidate for the actual current hidden state.

So Rt, decides the important parameters with their weights for the candidate hidden state as Rt being the gate where as Zt decides which of the two i.e. previous hidden state(ht) or the candidate hidden state should have higher impact in calculating the actual current hidden state.

Ht = (1- Zt) \* (Ht-1) + (Zt \* )

So, from the Zt one input goes to ht-1 as (1-zt) and other goes to the and finally we have a additive operation between both to ultimately decide for the current hidden state.

**Deep RNN’s:**

**The** Idea in deep RNN is to stack multiple RNN layers stacked upon each other. Results into a extraction of more complex features.

The output of the one RNN which is hidden state is given to the second RNN which stacked on top of that first RNN. Don’t get confused here with output given to the second timestamp.

This output of the ht is given to the second layer RNN for the same timestamp only. And likewise, we complete the timestamp one processing for all RNN layers and then followed for the second timestamp again for the deep layers.

Check code for better understanding.

In deep **LSTM’s** only the ht is given to the second layer of the LSTM and not the Ct as well. Only **Ht**.

**Story of the LLM’s:**

Based on the input output structure there are many types of the RNN’s:

1. Many to One RNN models: multiple input but has one output. i.e. sentiment analysis.
2. One to Many Analysis: One input generates multiple output. image description module.
3. Many to Many:

3.1 : Synchronous : where input length and output length is same. E.g. Part of speech tagging for each word. Named entity recognition for each word.

3.2 : Asynchronous : Where input length and output length is not same. E.g. Language translation, text summarization, question and answers, speech to text (subtitles).

Sequence to sequence problems:

Solves the problem of many to many asynchronous problems.

Which problems does it solves: Language translation, text summarization, question and answers, speech to text (subtitles).

Story of the sequence-to-sequence models consist of 5 parts:

1. Encoder Decoder.
2. Attention Mechanism
3. Transformers
4. Transfer Learning
5. LLM’s
6. Encoder Decoder: Encoder block compresses the input words and pass it to the decoder block.

* Now what decoder does is it uncompressed the input and returns the output.
* Encoder on compression keep the info in context vector which has the essence of all the information.
* So encoder get the information in the form of context vector. But as the words get more and more i.e. sentences gets higher and higher context vector is not able to keep the context of much previous words and it becomes recency biased context vector.
* Sentences with usually words greater than 30 is an issue.

1. Attention Mechanism:

* To overcome the drawbacks of the encoder decoder architecture we have attention mechanism.
* Unlike Encoder decoder which has only one context vector to summarize all the information of all the timestamp which then given to the decoder, Attention mechanism has access to all the states of encoder i.e. all the ht’s and ct’s of the encoder for each step unlike one context vector.
* So, for each specific word it does not rely on the compressed context vector which has the context of all the states in one vector.
* Attention mechanism has one neural layer attention layer which decides which of those all the stored state outputs is to be selected for that specific word prediction.
* So, it kind of put attention on the specific needed states which are needed and that happens with the help of the neural layer.
* Higher computational complexity.
* Both the Encode decoder as well as Attention Mechanism both has LSTM in them.

**Transformers :**

What transformer did, as they changed the landscape through which we process the NLP task.

Attention is all you need. The paper which first describes the transformers.

It does not have any LSTM’s, but it only emphasises attention.

**Drawbacks:**

**Training time**

**Hardware**

**Cost**

Transfer Learning:

Initially transfer learning would be used only in the CNN. But with

Complete after you watched that content again.

Though the points are as such:

1. Pre-Training
2. Fine- Tuning

Language models:

Transfer learning applicable model which are trained on very large amount of data.

GPT: GPT is a language model which is trained on the very large data.

GPT 3.5 was 45 TB of data with near about the 175 billion parameters.

Whereas GPT 4 is 1.8 trillion parameters.

Chat GPT is chat bot product developed from the GPT language model.

1. **Encoder Decoder**

Initially for tabular data we had ANN which works fine for tabular data without any sequence in it. For image data we had CNN and for sequential data where there is sequence in the data, we use RNN’s.

But in all the above case we have M inputs and 1 output we can manage variable inputs with zero array but in case there is n outputs it is bit tricky and for the same use sequence to sequence models.

Sequence to sequence models works on m variable inputs and n variable outputs.

Sequence to sequence models basic overview is very simple with one encoder block and other decoder block. Encoder block has LSTM / GRU in it and receives sequential array’s of input and generates a context vector which has the context of the input fed. This context vector is sent to the decoder which receives the one element of the context vector for one timestep and return the final output. But for firsts timestamp it receives one flag called as start which starts the decoding operations and starts producing the outputs whereas at the end decade’s LSTM receives the End flag which stops the anymore output being produced. Now this context vector is nothing but the cell state of the LSTM not both cell as well as hidden state.

Put the images from the Notes here,

Training the architecture by back propagation

**Forward propagation.**

* How does data look like: In our case let’s take an example of language translation model where we input we have language and output we have translated meaning of that sentence or word.
* Encoding the input to the Encoder and pass it in timestep with multiple timestep generating the context vector. After encoding both the input and output sequential data we have numerical sequence to sequence data whose patters needs to be learned by the sequence-to-sequence architecture.
* Encoder generates the context of the data which is nothing, but it learns the meaning of the sentence of the in encoding only to a degree and passes that learned meaning context to the decoder.
* Now decoder has one fully connected SoftMax layer which is used to predict the output corresponding to each time step. The no of nodes in the SoftMax layers are equal to the timestamp of the decoder. The max probability of the SoftMax layers prob vector is the output of that layer or rather the words corresponding to the index of the max probability as per the encoding.
* And likewise, you predict the next word for the next timestamp based on the same SoftMax probability.
* Here while learning if the decoder predicts the wrong word a right word is still given to the next timestamp as input that is overwriting the wrong output of the previous timestamp which is called **forced teaching** and this what led you to have optimized fast learning. And obviously the initial weights are defined randomly or initialization techniques. That forced teaching is sort of having proper target column for the input to learn from in decoder. Instead of learning from the wrong learn from the right.
* The probabilities where the prediction is correct are very high compared to the wrong prediction.

**Evaluation:**

After forward propagation we need to evaluate the prediction accuracy we do that with the help of the loss function.

Put the loss images :

Categorical cross entropy:

Calculate the loss for each timestep with the help of categorical cross entropy and aggregate the entire loss either mean or sum of all the losses of each timesteps.

The loss for the timestep where model was right is lower compared to the losses where model has made mistake.

Once we get the loss now, It’s time for **backpropagation.**

Initialization of the weight 🡪 Forward propagation 🡪 output 🡪 loss 🡪 gradient 🡪weight update (optimizer) 🡪 learning rate 🡪 new\_weight 🡪 new record 🡪 next epoch.

On calculating the loss we calculate the gradient and with the help of the optimizer we update the weight parameters with involvement of the learning rate.

This is how the training takes place with final weights.

**Prediction:**

Pretty much similar to the training with the difference of force teaching is not there.

And that is how we calculate the new output.

Put the diagram form the notes.

Improvement in sequence-to-sequence models:

1. **Embeddings:**

Using embedding gives you more dense representation of the words with low dimension representation.

So, you add an embedding layer at the beginning.

Pretrained embeddings word2vec, glove etc.

Encoding in encoder and decoder.

1. **Deep LSTM’s:**

Three reasons:

2.1 Long term dependencies: Able to remember the long-term contexts as we have multiple layer context vectors which keeps the longer context in memory and that is why it performs better for longer context vectors.

* 1. Layered (Hierarchical) representation: lower layer LSTMs understand the word level context like POS etc. whereas the middle layer LSTM’s understand the sentence level contexts and finally the top layer LSTM’s understand the paragraph level context.
  2. More parameters: On general level when you increased the number of parameters in the Neural network the level of understanding to which the NN understands gets increased and NN understands things on deeper level if we avoid overfitting.

1. **Reverse the input:**

Like give the reverse input in the encoder and the original correct one in Decoder,

The basic logic is that when we give the input in reverse order in encoder and with the correct sequence in the decoder the distance between the initial words and final words in the decoder reduces and that sometimes helps in understanding the initial context far better especially for the languages where the initial words hold more importance.