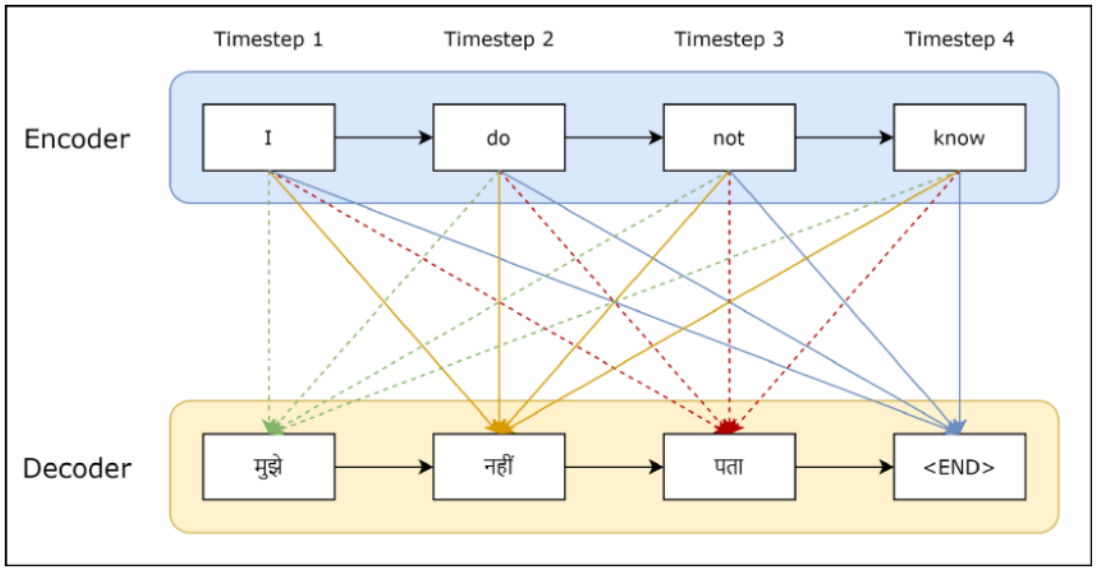
Machine Translation

Shambhavi Sud
1910110365

I implemented Machine Translation from **English to Hindi** using three methods, the first one being the Sequence-to-Sequence model, later I added Attention to the seq-to-seq model and finally I tried translation through Transformers. This report is a collection of all my learnings, throughout the semester while building the projects, the problems I faced, resolving them, and concepts. I have made an effort to explain the most complex ideas in the **easiest** way possible , so that even a beginner can understand these ideas.

Sequence-to-Sequence Architecture

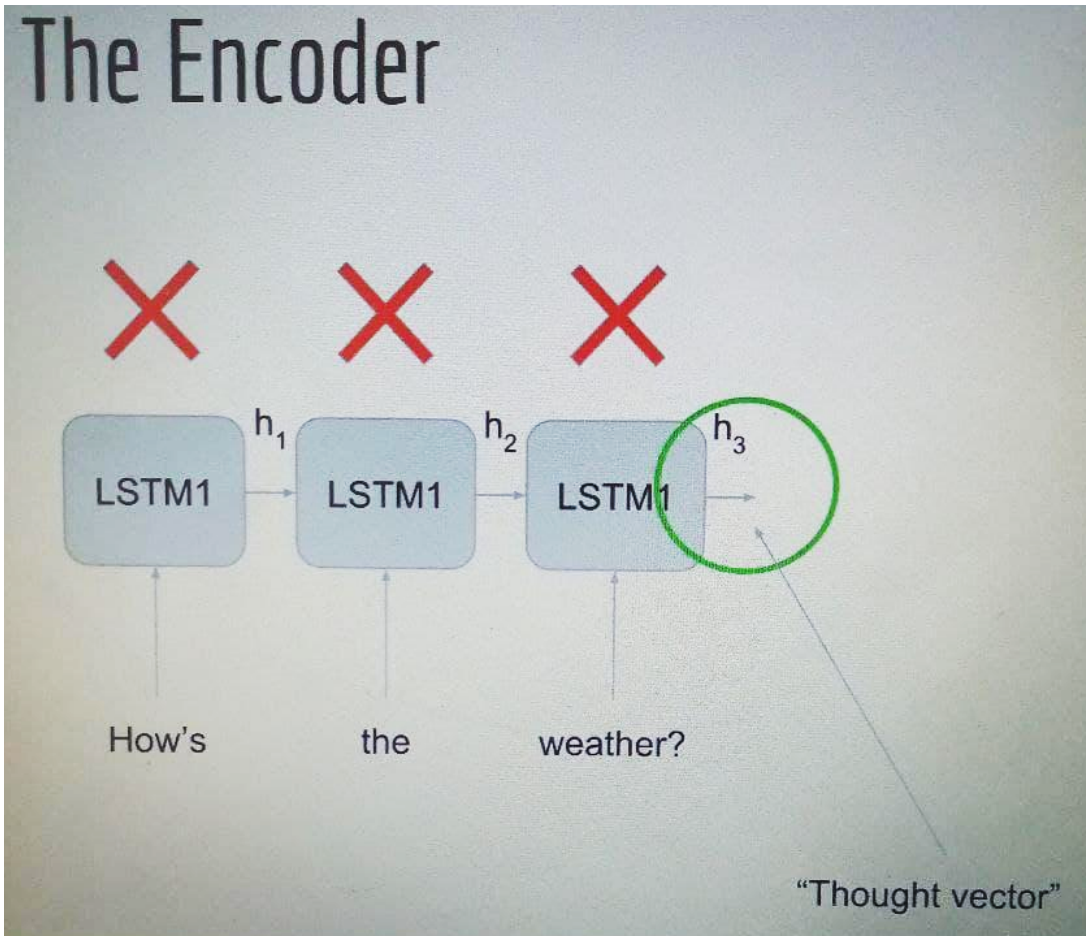
Motivation: While working on any translation from English to Hindi, we know that the number of words in English statements are not going to be the same as those in Hindi statements. While working with a standard RNN , if we keep all the outputs at each point in time , the length of the output is always going to be equal to the length of the input. The solution to this issue is the Sequence-to-Sequence Architecture. It consists mainly of a **dual** RNN system.



Machine translation using the Encoder-Decoder model

The first RNN takes input called **Encoder** and the second RNN which produces the translation is the **Decoder.**

Encoder: Works like a standard RNN whether that be a LSTM or a GRU where we pass a sequence of inputs and get back a series of H's. In case of Seq-to-Seq we want to keep only the last state of the sequence .In case of LSTM it would be h and c but for GRU it would just be h. So the encoder gives us hT(thought vector) i.e a vector of size M. It is just a single small compact vector representation of the input sentence, primarily why it's even called an Encoding. It does not contain any time information.

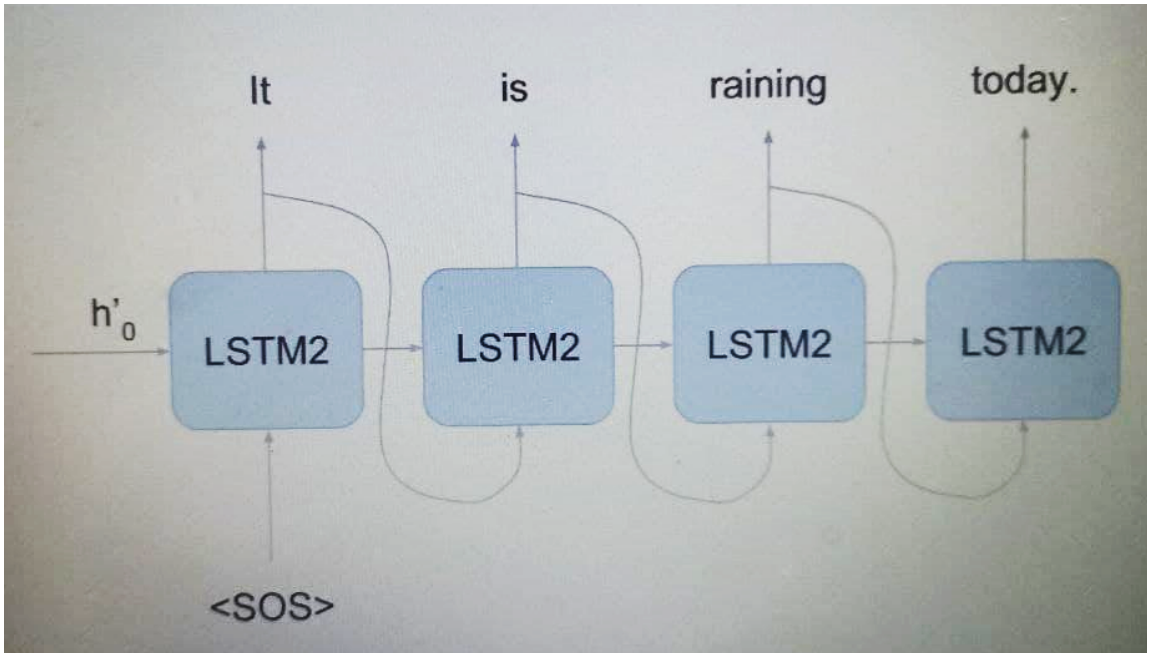


DECODER: On the decoder side we have a completely new RNN but of the same size as the Encoder RNN. Here we pass on our Thought Vector from earlier. Here we have an h'0 that is equal to hT from the encoder. For the x1 or the first input , we pass

a specific token to denote the start of the sentence. Now from this info of h'0 and x1 our RNN will calculate h1 which will be used to calculate y1. Y1 will be a vector of probabilities so from there we take argmax() to pick the most likely word in the target language.

How to predict the 2nd, 3rd ,4th words of the sentence?

The RNN decoder needs to take two things, one is the previous state h_1 and and x_2 (input). We pass on the argmax(y_1) as x_2 and calculate y_2. Then we pass on argmax(y_{12}) as input x_3 and so on. So, y_1 becomes x_2 and y_2 becomes x_3 and so on we get to the maximum length sequence.



DECODER

Hence Seq-to-Seq solves the problem of mapping an input of length T_x to an output of length T_y where T_x!=T_y.

Each RNN has 2 inputs and an output if we consider the previous hidden state to be an input. So we have our encoder, decoder and our input sentence goes into the encoder RNN , the outputs of encoder RNN get ignored since we only want the final hidden state. In the decoder RNN we take the outputs and compare them to the target sentence, which in the case of machine translation is just the input sentence translated into whatever language we want to learn. The decoder RNN has 2 inputs, one is the hidden state and the

other is the input sequence. What I initially planned was to take the previously generated word and feed that into the input at the next time step.
There exists something even better for training i.e is the Teacher Forcing.

TEACHER FORCING

Teacher Forcing works in such a way that instead of feeding the previously generated input at the bottom , it instead feeds the true previous word. So even if our model didn't get the true previous word right, Teacher Forcing corrects it. Hence the model predicts the next next word based on actual translation.
This helps the model to train, because it would be difficult for the model to generate the entire sentence at once.

We now want to pass in the true target sequence into the bottom of the decoder, but it has to be offset by one so that at the target, we are always trying to predict the next word. If they're aligned perfectly, then the decoder, which is learning to copy its input, which is useless. We know that RNN works with constant sequences. So if our input is of length 100, that our output will also be of length 100.
The problem is, what do we do when we want to make new predictions at test time during training?
We're always going to pass in the true target into the bottom of the decoder.
But for testing, we obviously can't do that.
For testing, we go back to the original architecture where we pass in the previous output into the next input.
Let's assume for a given RNN unit, its input sequence length is one. So for prediction, the input sequence is always going to be one. This is because we'll be doing a loop and generating each word one at a time. We must do it this way because we can't pass in the full sequence all at once.
That doesn't make sense because we haven't generated it yet.

So to summarize this issue into just a few sentences

1. Kera's must have constant input.
2. Decoder input size during training is of size Ty, if you are using teacher forcing.
3. The decoder input size during prediction is of size one, and these are both in conflict with 1.

So how do we solve this problem?

Well, the answer is to simply create two different models.The first model I have created will be for training purposes only. The second model I created will be for sampling, and we will make use of the previously defined decoding layers that were already part of the train model.
So for the second model, we can define a new set of inputs and for these inputs, their input length will be one.

Pseudocode

```
Emb = Embedding(); lstm = LSTM(); dense = Dense()
Input1 = Input(length Ty)
Model1 = Model(input1, dense(lstm(emb(input1))))

input2= Input(length=1)
Model2 = Model(input2, dense(lstm(emb(input2))))

H = encoder model output; x = <SOS>

For t in range(Ty):
     X, h = model12.predict(x,h)
```

So both models 1 and 2 use the same layers that were defined at the beginning. The only difference between the two models is that they have a different input. The pseudocode for generating a translation is basically just a for loop.
First we get the final hidden state from our encoder model and so we assign that to the initial h and then we assign the start of sentence token to the initial x.
From this, we can get back a prediction for the output word and the next and state.
So that's the new x and the new h.
And then we pass this output word along with the new hidden state back into the decoder, RNN, to get the next output and the next hidden state. We are always passing in x and h and getting back a new x and h.
This is why the decoder RNN expects an input length of one, it's because it can only take in one input at a time because we can only generate one input at a time.

DATASET

Data link: http://www.manythings.org/anki/

I made use of the Hindi - English Dataset . I also used pretrained glove vectors that were trained by wikipedia data domes .These are provided by Stanford where glove was originally invented.

Observations Seq-to-Seq Code:

- On seeing the results we observe that we are overfitting, and that's why we want to dropout earlier. This could be due to not having enough training data, or our model being too expressive etc.
- The final step while doing machine learning is to see if we can even overfit in the first place.
- If we can't even do that, then we know our model isn't powerful enough.
- So in this case, this is actually a good thing because it demonstrates the viability of this model and it shows that it's possible to learn how to translate languages.

Limitations of Seq-to-Seq

- The encoder decoder architecture forces us to encode the entire input sequence into just one small vector. This hurts the performance. It would be alot better if our decoder could make use of all the information rather than just the final hidden state.
- We add words one after another in the encoder, which means we will certainly lose information from the beginning of the input sentence for each word we got. So for very long sequences and very long sentences the information will not be complete at the end of the encoder and so when we want to predict the next word in the decoder we could miss information that has been lost in the encoding phase.
- To address the same issue we add ATtention. So basically what it does is that during the decoding phase, we add a new input to our cells in the RNN and we call that the context vector, and that's actually vector that is supposed to convey a global information about the whole input sequence and more precisely how each step of the input sequence is related to our current status in the decoding phase. Hence we lose alot of information for very long sequences.

It's just during the decoding phase we simply add a new vector that conveys more global information about the whole input sequence.
And this context vector changes at each step of the decoding phase so that each cell of the RNN then gets information from the input sequence that is related to the current state.

ATTENTION

There exist several ways of solving the many-to-one task.
The first way is just to do what we've always done in the past, pass the sequence through the RNN and pass the final output through a dense layer to get a prediction conditioned on the entire input sequence.This makes sense because we want our RNN to make a prediction after considering the entire input sentence. It would be odd, to say the least, if we consistently read only half a sentence to determine its meaning.

- While LSTMs and GRUs are capable of learning "long term" dependencies, they still have limitations.
- It's a question of how long …..An LSTM can be good for about 1-2 sentences, but it probably wouldn't work out in the case of 1000 sentences. Like I did max pooling in the previous model, to be able to look at a hidden state somewhere in the middle, so in that case we do the global max pooling over the RNN states.
- By taking the last RNN state, we hope that the RNN has both found relevant features and "remembered"it all the way to the end.

Normal Max (Hard Max) takes the maximum and forgets about everything else.
Softmax gives us the probability distribution over each element for "how much to care".

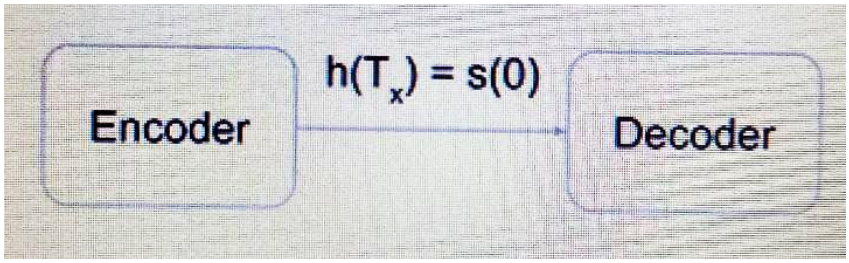
Here I'm still working on Seq-to-Seq, I'm still working on the same data set from the last section. So we're still going to have an encoder and a decoder, just the way we build them is going to be more advanced.
The encoder here is going to use a bidirectional LSTM rather than a regular LSTM.

This is going to give us an entire sequence of hidden states where the length is Tx.
And bidirectional means, we are reading the state both forwards and backwards and passing those states on to the Decoder.

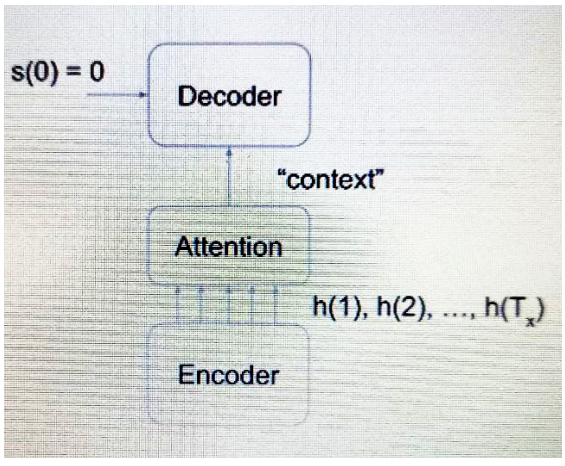
Also, that since there are two LSTMs within this unit, the output is going to be of size T_x * 2M. Here we completely ignore the cell states of the Bidirectional LSTM and encoder & shall only take the H's.

Next, we have the decoder side, which is responsible for taking the H's and producing the translation.

In regular seq-to-seq we take the last hidden state from the encoder and pass that in as the initial state to the LSTM.
In order to differentiate between the hidden state of the encoder and the decoder,lets go by calling the encoder's hidden state H and the decoder's hidden state S..
So with regular seq to seq, H(Tx) is the same as S(0).

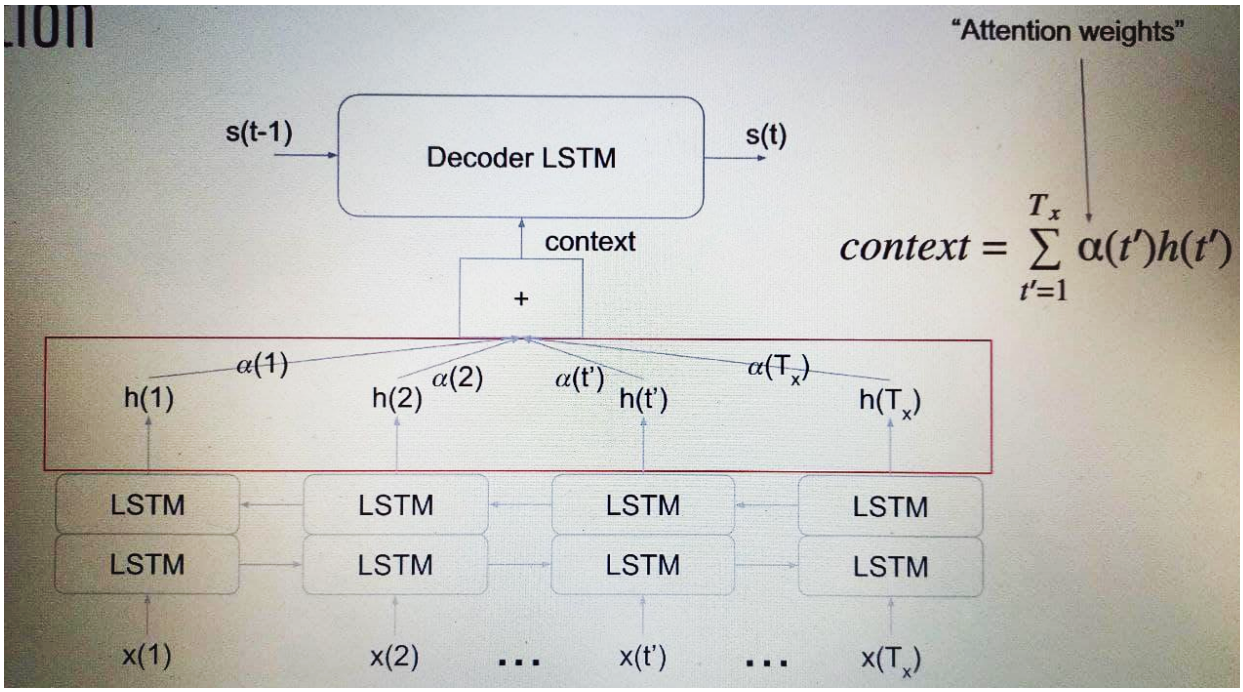


Now with Attention we no longer care only about H(Tx) but we care about all of the Hs.
So all the H's are going to get fed into some kind of attention calculator.
And this attention calculator is going to give us one final vector called the **context**, which tells us which H we care most about. Now, because the H's are all getting passed into the bottom of the decoder LSTM, there's no longer any need to pass it through the side.
Through the side we just set as a S(0)=0.
When we're talking about attention, it's often more useful to draw the encoder at the bottom of the page and the decoder at the top of the page rather than side by side. The reason is you can think of the data from each of the hidden states of the encoder as flowing upward into the decoder.



We know that decoder is going to be LSTM, this cant be bidirectional since we are creating only one word at a time, hence this decoder is going to be a regular LSTM.

Now we are at the arbitrary step of the decoder LSTM and we want to predict the next word.
We know that the LSTM should have two arrows going in , one from the side and one from bottom, which now since we are doing attention, comes from the encoders hidden state. This thing that goes at the bottom has got to be a vector also called as the CONTEXT VECTOR.



How do we get this context vector? Well that's where attention comes into play.

This vector is just the weighted average of all the hidden states from the decoder.The usual symbol we use for these weights is α(Alpha), and they tell us how important each hidden state is for producing this particular output word .These alphas are called attention weights.

How do we calculate α(attention weights) ? A neural network!

This makes the entire attention network end to end differentiable so we can train the whole thing all at once. The entire system is just one humongous neural network containing several mini neural networks.

$$\alpha_{t'} = NeuralNet([s_{t-1}, h_{t'}]),\ t' = 1...T_x$$

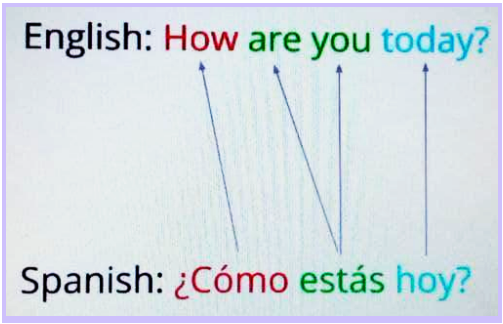
$$context = \sum_{t'=1}^{T_x} \alpha(t')h(t')$$

We have two different **t's :** one is the **t** and the other is **t'** . This is because we have two different sequences , the input sequence and the output sequence for each step of the output sequence, we want to consider all steps of the input sequence. So right now, we're looking at a single step of the output over all hidden states from the input.
So the **t without the prime** is telling me which output I'm currently trying to calculate attention for. **t'** tells me which Alpha I'm currently calculating and how many alphas are there. We have Tx alphas because we have Tx hidden states. α is just a way for the state to tell us how much we care about that state.

t is for the output sequence ($t=1.....T_y$)
t' is for the input sequence ($t=1.....T_x$)

s(t-1)

Attention weights depend on two things. They depend not only on the hidden states, but also where I am in the output sequence.
In other words, if I just started generating the output, I'm going to care about different issues than if I'm almost finished generating the output. So you can imagine that if we didn't condition on s_{t-1} here and the Alphas depended only
on the **h'**s, then the attention, Weights would just be the same for every step, which doesn't make sense at all because in that case you wouldn't need attention in the first place.
So we need information from the **s'**s because that gives us the context of where we are at the output sequence.
For Example



suppose I want to translate the sentence, "how are you today? "in the Spanish

When generating the first word of the translation, my attention is currently on the English word **"HOW"**, when I'm generating this second word in the translation, my attention is no longer in the English word **how** but has moved over to **are you**. So ***which part of the input sequence I pay attention to must change as I generate new words in the output***. The attention weights don't just depend on the **h**'s, but also the **s**'s.

Calculating Attention Weights

Pseudocode

```
Z = concat[ s(t-1) , h( t' ) ]
Z= ( W1z + b1 ) # layer 1 of ANN
Z= softmax( W2z + b2 )  # layer 2 of ANN
```

Within the actual code, this Z is going to go through the regular feed forward neural network. I have used two dense layers and this is what I have tried to show in the above pseudocode. The last step is softmax, because this is going to give me a probability telling me how much to pay attention to each **h**.

But in these equations, I've only calculated the output for one hidden state, **h(t')** . In in order to properly do the softmax, it needs to be over all the activations for all the hidden states from **t'=1 to T_x**.

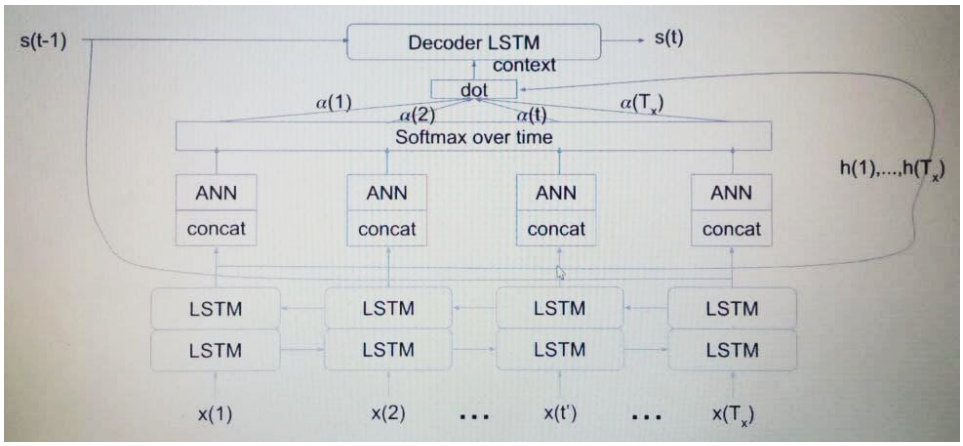
$$\sum_{t'=1}^{T_x} \alpha(t') = 1$$

This is because we want all the Alphas to sum to one that's going to give us a proper weighted average of the hidden states.

So what I've really done iscalculate the same thing for all the **h(t')** ..all at the same time. This is required because the final softmax has to be the overall $\mathbf{T_x}$ of those values.
So I pass each of the **h'**s through the same neural network, get all those outputs, and then calculate the softmax, which is kind of a special softmax over time.

$$\alpha(t') = \frac{exp(\, out(t')\,)}{\sum_{\tau=1}^{T_x} exp(\, out(\tau)\,)}$$

We observe that while there's h_1, h_2 …… all the way till h_{Tx} while we have only one **s(t-1)** , this is because I just copy each **s(t-1)** to each **h.** That way each of them has a copy to concatenate with and then we can pass each of those vectors through the same neural network and get their outputs. At the end, I calculate the softmax over each of those output's.



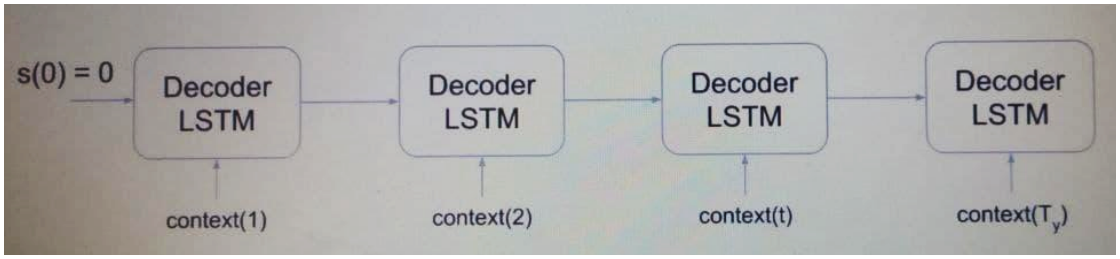
Calculating Attention Weights

So this is why when you look at diagrams of attention, you're going to see s(t-1) getting copied over to each hidden state **h**. So we can see that as of T minus one, which comes from the upper left, gets concatenated to each of the **h**'s, then those get passed through a neural network. Then those neural network outputs get softmaxed. That gives us the Alphas. Those Alphas get dotted with the acres to give us a weighted sum of the parts. That gives us the context which gets passed into the bottom of the decoder LSTM. Now that we have both the bottom input and the left side input, which is just s(t-1), we can calculate the s(t) and so on.

We now have all these alphas which are going to help us weight each of the encoders hidden states, and that can give us the context vector for each output step.
We multiply each of the hidden states by its corresponding alpha, and we assign this to a new vector called the context vector.

Once we have this, we know exactly what to feed into the decoder LSTM. So on the left side, we have the previous hidden state, the initial value for that is just going to be zero at the bottom, we

have the context vector. And so if I calculate the output prediction, I just do this $\mathbf{T_y}$ times based on step one we have as zero equals zero and then we pass in context(1), at the next step we have s(1) then we pass into the bottom context(2).
And we do that over and over again until the last step when we pass in the context of $\mathbf{T_y}$.



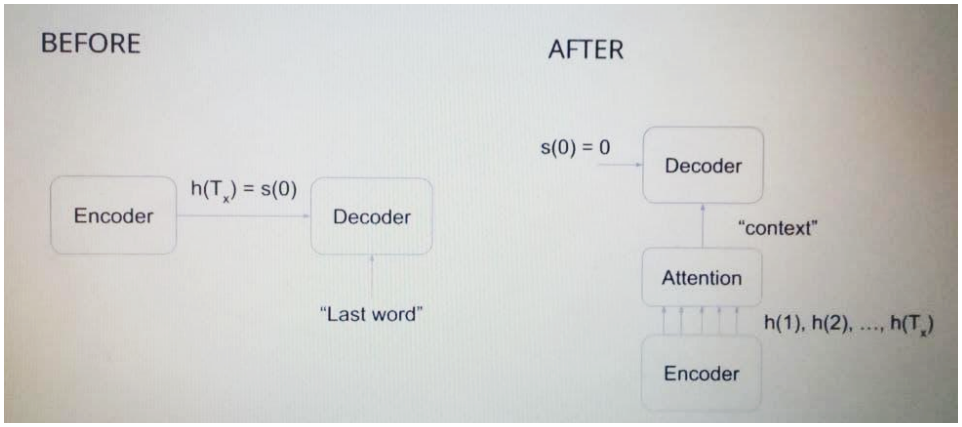
Pseudocode

```
H = encoder( input )
  S = 0
For t in range( Ty):
    Alphas = neural_net( s,h )
    Context = dot( alphas, h )
    O , S = decoder_lstm( context, initial_state = s)
    Output_prediction = dense( o )
```

So the first step is to initialize **S** to a bunch of zeros. That comes after grabbing the hidden states from the encoder, given the input sentence. Then we go into a loop, $\mathbf{T_y}$ times.
The first thing I do inside the loop is calculate the attention weights, given the previous S and all of the hidden states from the encoder. Then I use that to calculate the context vector, then I pass in this context vector into our decoder LSTM along with S as its previous state. From this, we get an output and also a new **S** which we'll be using on the next round. With this output, we can pass this through our final dense layer with a softmax to get the current translated word.

In other words, this is almost like the decoding step from our previous seq -to-seq model, except that now we're doing it in terms of symbolic variables rather than real numpy arrays.

Overview



With all the seq-to-seq, the encoders final hidden state of h($\mathbf{T_x}$) gets passed in as the initial hidden state for the decoder s(0). For seq-to-seq with attention s(0) is just zero, since now the **h**'s go through the bottom. At the bottom, we have the encoder, which is now a bidirectional LSTM. It's going to output a hidden state for every timestep one all the way up to $\mathbf{T_x}$ .In the middle we have the attention mechanism, this gives us the Alpha's, which are the attention weights for each hidden state h. They tell us how important each hidden state currently is for producing the translation. So we take a weighted sum of the **h**'s using these alphas, and that gives us the context vector and this is what goes into the bottom of the decoder LSTM.

This allows us to calculate the next S and the next context and so on.
Importantly, on each output step, the Alphas are recalculated because where we pay attention to changes, depending on which part of the output we are currently producing.

ISSUES WITH TEACHER FORCING & SOLVING THEM

RNN unit has 2 inputs, where one goes through left i.e the previous hidden state and one goes through the bottom. However, here we pass a context vector at the bottom of the RNN unit. This poses a problem since previously with Teacher Forcing we had been passing the previous word to the bottom.

So basically, this input has already been taken up by the context where can I pass in the previous correct word, this isn't a problem at all, if I don't want to use teacher forcing, but it is a problem if I do.

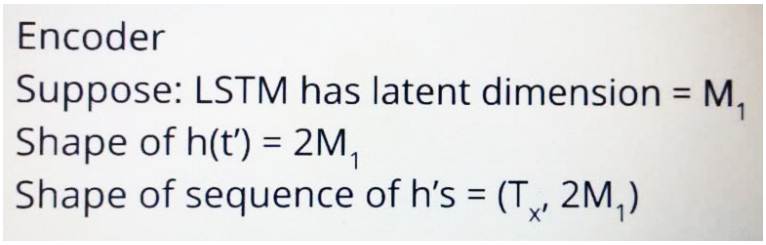
The **simple solution** is to just concatenate these two things together.
So during training, I'll have the correct previous word concatenated with the context, and this gets passed in into the bottom of the decoder LSTM. During prediction, I'll have the previously

generated word concatenated with the context and this gets passed into the bottom of the decoder LSTM. One thing I can do is even add a dense layer in between, so I can concat, dense and then pass it into the LSTM.

Therefore it is possible to use Teacher forcing here as well.

KEEPING TRACK OF SHAPES

1. ENCODER Output: Let's assume that the Encoder LSTM has a latent dimensionality of M1, Since we use a Bi-directional LSTM, we know that the full hidden state h(t') will $2*M_1$. SInce the length of the input sequence is $\mathbf{T_x}$ then the length of the output will be of size $\mathbf{T_x * 2M_1}$



Also each h(t) has to also be concatenated with $\mathbf{s_{t-1}}$

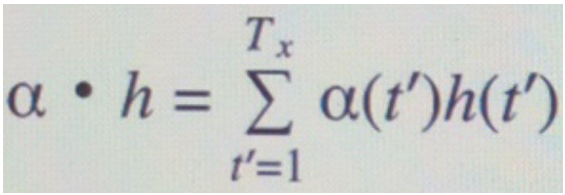
2. Decoder: Let the latent dimensionality of decoder lstm be M2.
   Now concatenating a thing of size $2*M_1$ with a thing of size M2 , the size of the resulting vector would be $2M_1 + M_2$
   The entire sequence will have the shape $(2M_1 + M_2)*(\mathbf{T_x})$

Each of these concatenated vectors is going to get passed into a mini neural network and the output of that is going to be alpha, which is a scalar. Therefore, I'm going to pass in a thing of size $(2M_1 + M_2)$ into a couple of dense layers and the output is going to be a size one.

But we have $\mathbf{T_x}$ alphas , so the actual size of my output will be $(1*\mathbf{T_x})$

Now, the problem is, we know that in order to calculate the Alphas, we need to perform the softmax over the output at each time step of the neural network. So if we have a batch size of N, the output of the neural network will be of size (N , $\mathbf{T_x}$, 1).
Unfortunately, the default implementation of softmax divides by that sum in the last dimension or in other words, it ensures that if you sum all the elements in the last dimension, you get one. However, I don't want that in this case because time goes along the second dimension. So in order to do this, I wrote my own softmax over time function that divides by the sum over axis equals one.

Alphas are for weighting the encoder hidden states and our context vector is the weighted average of those hidden states.



$$\alpha \cdot h = \sum_{t'=1}^{T_x} \alpha(t')h(t')$$

The shape of the above equation is as follows:
As Alpha is a scalar and **h** is a vector of size $2M_1$, the shape of the context vector is also a size $2M_1$.
The next step is to pass the context vector into the bottom of the decoder LSTM while the previous decoder hidden state S goes into the side of the decoder LSTM. We also have to pass in the previous cell state, **C,** for simplicity's sake, let's assume
we're not using teacher forcing for now. So only the context vector needs to go through the bottom of the LSTM.
I need to go through the decoder LSTM; $\mathbf{T_y}$ times because that's the length of the output sequence.

Pseudocode

```
H = encoder( input )
S = 0, C = 0
Outputs = [ ]
For t in range (Ty) :
      Context = do_attention( s, h )# s( t-1 ), h( 1 ),., h( Tx )
       O , S , C = decoder_lstm( context, init = [ S, C ] )
       Probabilities = final_dense( O )
       outputs.append( Probabilities )
model = Model( input, outputs ) ,……..,
```

Step one is to get all the h's from the encoder. Step two is to set the initial values of S & C, which are going to be all zeros and to create a list of outputs. Step three is to go through a loop $\mathbf{T_y}$ Times .
Inside the loop, we do the following first.
We do one step of attention to get the current context, that takes in the current S and H and gives us back the context vector.Then we pass in this context vector along with S and C into our decoder lstm. This gives us back the output, a new S and a new C.
We passed this output into the final dense layer, which gives us the output word probabilities, and so we just do this loop $\mathbf{T_y}$ times until we've collected all the outputs.
Once we have the outputs, we're ready to build our model objects, train it and so on.

Moving towards Transformers

We have seen how we used to address the sequence to sequence task by using RNN and how we improve them by adding this attention mechanism.
That was a very powerful tool, but it seems that we might be able to go even further in that direction.
So what the Google research teams have been thinking about is, the attention mechanism added global behavior to the coding phase, but the RNN still has this weakness about not being global enough.
So what if actually all we need in order to perform well in this sequence to sequence process is attention.

Hence all we need is attention is actually the name of the paper that Google released when they introduced the transformer. That just illustrates the fact that the transformer only uses the attention mechanism in order to encode and decode sequences and completely get rid of the RNNs.

TRANSFORMERS

Here we don't feed the encoder one word at a time. We just give it the whole sequence and it will process the whole sequence at once.
This is important to keep the global aspect of the processing that we wanted to have. We use the outputs of our decoder as new inputs as we did before.But once again, we do it with the whole sentence. It's just that by adding the start talking at the beginning of the input sequence for the

decoder, you shift the whole sentence right and with that, we know that the last word from the output sequence here will be the predicted next words for our sequence.
The sequence we use as inputs for our decoder is the one that we already had at the previous iteration, plus the new words that we will predict at the end of this sequence.

So just to summarize how it works.

The idea is when you have two sequences, it's just we compose the first sentence according to how each element of this first sentence is related to the elements of the second sentence. We use three times the same sentence, and that's actually what they call self attention, which is the key of the encoder in the transformer. So what we did before with the RNNs is that we used a standard neural network. We had information about the beginning of the sentence. We have new words. We make some computation and we get new information about the sentence. But what we do here is that we have a whole sentence and we will see how each word of its sentence is related to the other words of the same sentence. And then we will recompose this sentence according to that.
There is still an encoder and decoder and we compute whole sequences at once instead of one word after the other. The way we do that is by re-composing sequences, making different items from each sequence according to how they are related to each other. So that's just a way to get global information about the whole sentence.

A more General Intuition

Just an interesting thing actually, that in my opinion would be how the brain processes sentences. When you have a sentence in mind, we don't process the meaning of each sentence by going word after the other. We actually have the whole sentence in mind and we capture the meaning of it at once. And if you think about it, that's actually easy to see.
When you think about reading. I don't know if you are like me,, but if you read slowly, you may experience that you have to go back many times when you read a single sentence.
And that's not only because you maybe are not concentrating enough.
That's also because if you read too slowly word after word, it's not how your brain is optimized to understand a sentence. When you arrive at the end of the sentence, you have already forgotten the beginning and you are losing the whole sense of a sentence.
Actually, a sentence is made to be understood globally as one big book. Actually, it's kind of a burden that we have to use words that we have when we read or when we write to get them one at a time. Because if you think about that, you might realize that when you hear someone speaking, you wait for the end of a sentence and then you get the whole idea of the sentence at once when you process the meaning of it.

That is actually what fast readers do. Those people that can read a page in a few seconds and have trained a lot to improve their reading speech.

What they do instead of reading word by word, is that they try to use the global vision to get the biggest group of words at once that they can and to process the whole meaning of it at once.
And you can do that because your brain is made to understand language like that.
So my understanding of this transformer is that it goes in the same direction.
Instead of processing word by word, we try to capture the meaning of the whole sentence at once by combining each item with the others.

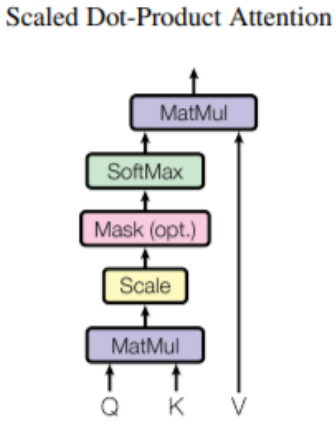
Attention Mechanism of Transformer

Main Idea-

1. 2 sequences equal incase of self attention
2. To see how much each element from A is related to each element in B
3. Recombine A according to this.

Before: a given sequence A(and a context B)
After: a new sequence where element i is a mix of elements from A that were related to element B_i

Dot Product: Mathematically gives idea of similarity between two vectors. Like Joy and Despair are exact opposites so its product would give -1, while tree is no way related to Joy hence dot product will be 0. We apply it to each pair of words/elements from our sequences to get their similarity.



Q, K and V are matrices representing sequences/sentences after embedding

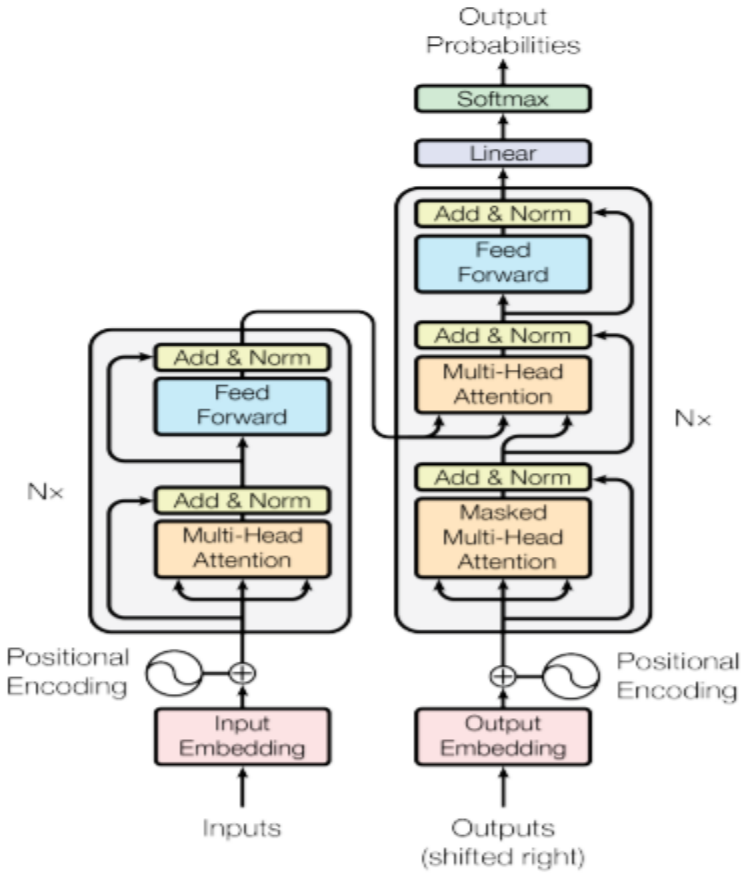


Figure 1: The Transformer - model architecture.

Q is my context sequence B & K, V are sequences that convey the information, that we want to work with. we divide by small scale- d_k which is the dimension of each element from each word to the key, it makes the model more stable.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

So the new matrix QK^T , gives us how every element of Q is related to each element of K.
Next I apply softmax.Softmax is an operation that takes as inputs as vectors and as outputs, we get a vector of the same dimension that each element will be between zero and one.
The second thing is that we keep the relations between each element of the initial vector. So if Element two was lower, that element three eight will be the same after the softmax.
And the third one is that the sum of all elements from the output of the Softmax should be equal to one.**So this softmax is necessary in order to get valid weights**.

Attention Layer

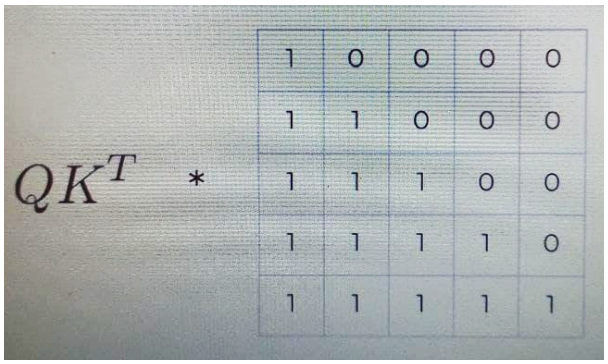
Self Attention: Beginning of encoding and decoding layers. We recompose a sequence to know how each element is related to others ,grabbing global information about sentence/sequence

Q=K=V

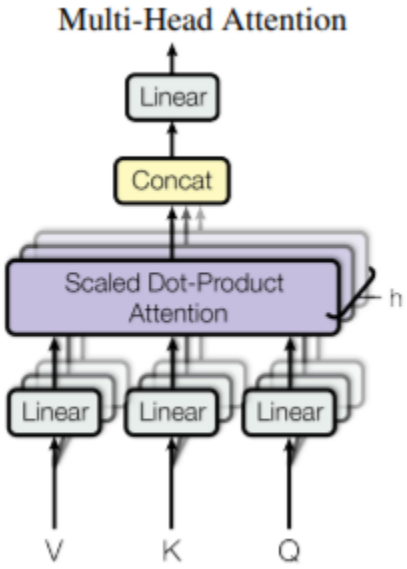
Encoder-Decoder- Attention: we have an internal sequence in the decoder Q and a context from the encoder(K=V) . Our new sequence is a combination of information from our text guided by the relation decoder sequence-encoder output

K=V

Look Ahead Mask: during training we feed a whole output sentence to the decoder , but to predict the word n we must not look at words after n. We change the attention matrix.



Multi-Head-Attention-Layer



Linear Projections: Applying our attention mechanism to multiple layered subspaces.

“Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.” - Paper

Mathematically : one big linear function, and then a splitting allows subspace to compose with full original sublayer. Splitting and then applying a linear function restricts the possibilities.

POSITIONAL ENCODING

Why do we need it?

No convolution, no recurrence: the model has no clue about the order of the sentence. Permuting the first and second word of the sentence would make no difference.

In RNNs , CNNs the order of the words had an impact on the process.The fact that I use a small filter that goes through the whole sentence, it means that words that are close to each other in a sentence will appear in the same filter. So the order of the words has an importance and an inverting two words will change the outputs of our RNN just because two words might not appear anymore in the same filter.The fact that we feed the words one after another gives a real importance to the order of the words.

With transformers that's not the case anymore because we compute the attention mechanism in a global way and we give the same role to each word.
So, for instance, let's say we don't use a positional encoding and let's say we have an English sentence as input for the encoder we want to translate it into the Hindi sentence .The inputs and outputs of our decoder will be a different sentence. So let's imagine that we invert the first and the third word in our initial English sentence, in our self attention mechanism, we will just again invert the first and the third element of the sequence, because words that were previously related to the first elements are not related to the third element.
So when we recompose the sentence at the end of the attention mechanism, we can put it in the third element instead of the first.
We just keep the inversion and sentence for the first element instead of the third.
So basically at the very beginning in the inputs of our encoder, if we invert the first and the third position, it will just keep this inversion throughout the whole encoding phase.

So any inversion in the order of the words of the input sentence will have no impact, because in operation, we will just get the elements that are important for us with no regard to where they are in the sentence.

And that's a shame, because in our language, of course, the place of the words in a sentence has importance. So we would like to find a way for our transformer to have an idea of the positions of the words in a sentence.
Hence we add a positional encoding: Numbers after the embedding layer that help us keep track of the position of the words.

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

pos = index of word in sentence (between 0 and seq_len - 1)
i = one of the dimensions of the embedding (between 0 and d_model-1)

The idea is just to add something at the numbers that will be different for each position in the sentence and for each dimension of the embedding so that we lose the symmetry between each position. For each dimension of our embedding, which is represented by the number **i** , I will get a sign of cosine function with respect to the position in the sequence.
Hence I add positional encoding in order to give importance to the order of the words in our sequence, because the attention mechanism is very global and is symmetrical with respect to the positions of the words.

FEED FORWARD LAYERS: at the end of each encoding/decoding sublayers

- Composed of 2 linear transformations
- Applied to each position separately and identically
- Different for each sublayer : For each of the 8 encoding sublayers and 8 decoding sublayers, we apply a different feed forward. That is total 16 different feed forward sublayers that my model has to learn , that is 32 dense layers because each feed forward is made of two dense layers.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

It's just a standard dense layer or more precisely 2 dense layers that I used in order to allow our model to learn things by itself. so it should feed forward neural networks made up of two linear transformations or two dense layers.
x is the input to the feed forward. I multiply it by a matrix and later the bias then this max of zero and the result is what we call ReLU activation. It's just a function that removes all the negative values of our dense layer, and the result of that is used to compute the second dense layer. So we

have here a second matrix to apply and a second bias that we add, and the output of all this will have the same shape. So again, we will have a sequence of the same length and the same embedding dimension.

ADD & NORM residual connections: lets not forget about the information we previously had in each position. It helps learning during backward propagation. After each attention or feed forward sublayers we have an add & norm.

It means that the outputs of those sublayers are actually added to the inputs of the sublayers and then we just do a standard normalization. The two reasons i do this is

1) I have computed new things that give me new information about the data, but I don't want to forget the way it was before. For instance, in the self attention mechanism, after I compose a sentence with regards to the relations between each word, I still want to remember at least the bits, how the sentence was before. So it could be really useful to add the original sentence to the output of the self attention sublayer.

2) Residual Connections are very common in AI and they make learning easier.
3) It makes it easier for the back propagation phase to have access to the previous steps. For instance, we have access to the outputs of this attention sublayer ,so we have access to how we train those weights, hence we don't have to go through the whole dense layers right here in order to have access to how this output was computed. So that's also a good way to improve the learning phase.

DROPOUT: "We apply dropout to the output of each sub-layer, before it is added to the sub-layer input and normalized. In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks."
It shouts down some (10-20%) neurons during training to prevent overfitting.

The idea behind that is that by turning off certain neurons, we force AI to have good results, to compute good outputs with a smaller part of its capacities, and that helps keep some kind of a generality. Overfitting means the AI getting too close to the training set. So it will perform well in the training set but will perform poorly with the validation set.

LAST LINEAR: output of the decoder goes through a dense layer with a vocab_size unit and softmax, to get probabilities for each word. It's applied independently to each element of the sequence once again and the number of units. So the output dimension of this dense layer will be

the vocab_size of our target language for right from English to Hindi, the dimension of the output to the number of units of the dense layer will be the number of words that we have in our Hindi vocabulary. So after softmax we get actual probabilities for each word of our dictionary. For each element gets the word that has the higher probability and we get a sequence or a sentence that should be the output of a transformer.

DATASET

link: https://www.kaggle.com/aiswaryaramachandran/hindienglish-corpora
This corpus contains TED talks, news articles, Wikipedia articles, etc.

ACKNOWLEDGMENTS

[1] I built the Transformer using the following tutorial link
[2] Attention is all you need
[3] https://www.analyticsvidhya.com/blog/2020/08/a-simple-introduction-to-sequence-to-sequence-models/