Outline

- Issues with RNNs
- Comparison with Transformers

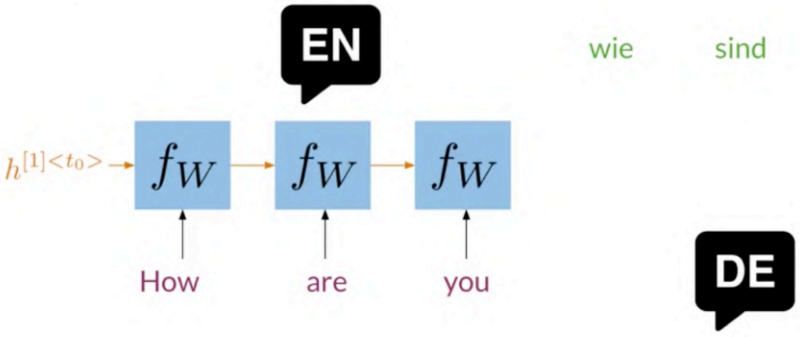
Transformer models are purely attention based models.

Neural Machine Translation

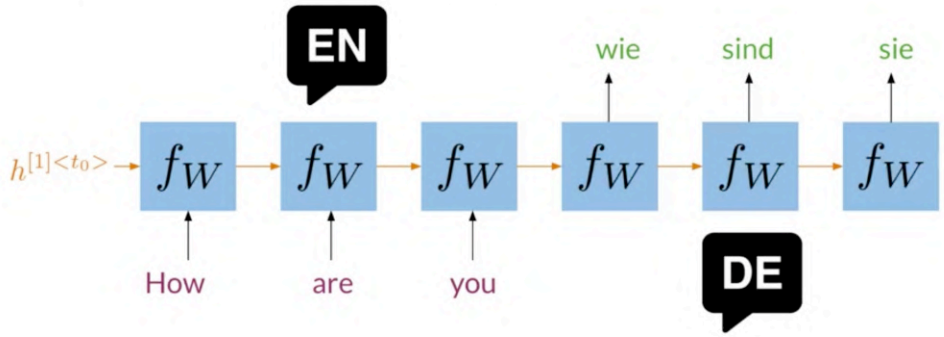
In neural machine translation, you use a neural network architecture to translate from one language to another. In this example, we're going to translate from English to German.

Neural Machine Translation

Using an RNN, you have to take sequential steps to encode your input, and you start from the beginning of your input making computations at every step until you reach the end.

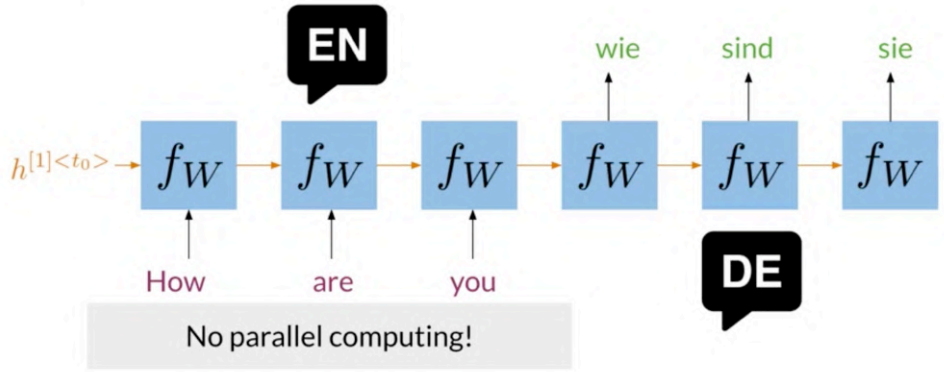


Neural Machine Translation



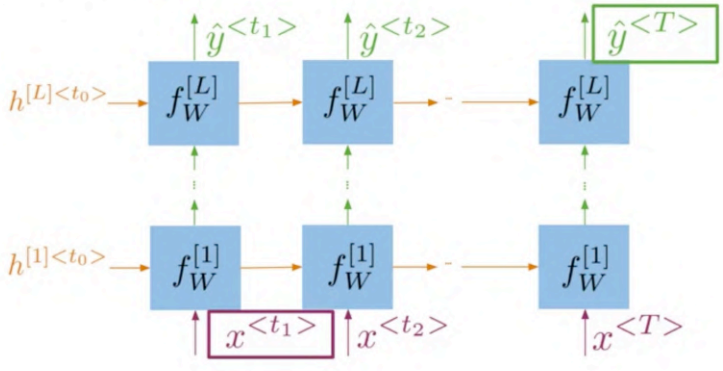
At that point, you decode the information following a similar sequential procedure.

Neural Machine Translation



As you can see here, you have to go through every word in your inputs starting with the first word followed by the second word, one after another. In sequential matcher in order to start the translation, that is done in a sequential way too. For that reason, there is not much room for parallel computations here. The more words you have in the input sequence, the more time it will take to process that sentence.

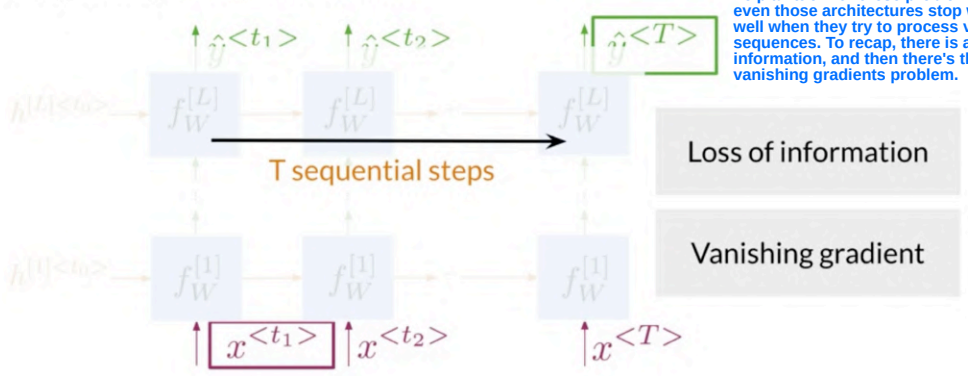
Seq2Seq Architectures



Take a look at a more general sequence to sequence architecture. In this case, to propagate information from your first word to the last output, you have to go through T sequential steps. Where T is an integer that stands for the number of time-steps that your model will go through to process the inputs of one example sentence. If let's say for instance, you are inputting a sentence that consist of five words, then the model will take five times steps to encode the sentence, and in this example, T equals five.

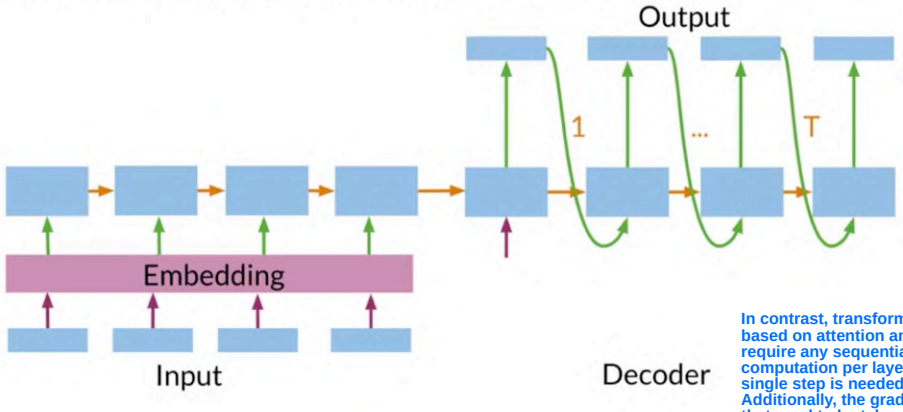
As you may recall from earlier in the specialization, with large sequences, the information tends to get lost within the network, and vanishing gradients problems arise related to the length of your input sequences. LSTMs and GRUs help a little with these problems. But even those architectures stop working well when they try to process very long sequences. To recap, there is a loss of information, and then there's the vanishing gradients problem.

Seq2Seq Architectures



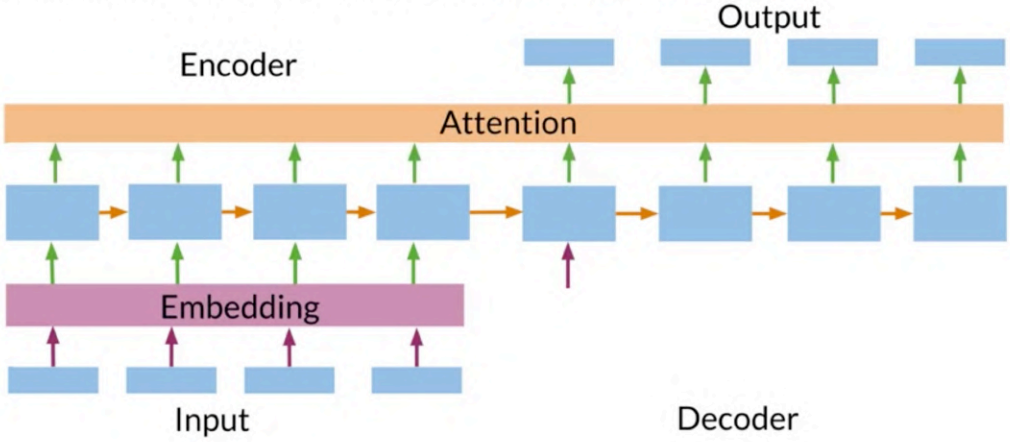
Now transformers are models specially designed to tackle some of the problems that you just saw. For instance, let's recall the conventional encoder-decoder architecture which is based on RNNs and used to compute T sequential steps.

RNNs vs Transformer: Encoder-Decoder

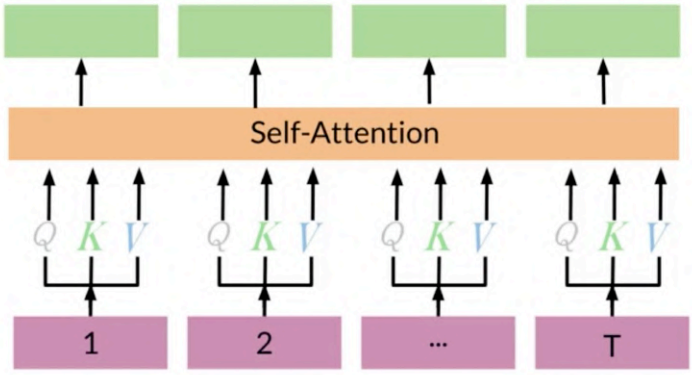


In contrast, transformers are based on attention and don't require any sequential computation per layer, only one single step is needed. Additionally, the gradient steps that need to be taken from the last output to the first input in a transformer is just one. For RNNs, the number of steps is equal to T. Finally, transformers don't suffer from vanishing gradients problems that are related to the length of the sequences.

RNNs vs Transformer: Encoder-Decoder

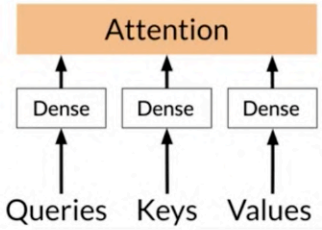


RNNs vs Transformer: Multi-headed attention



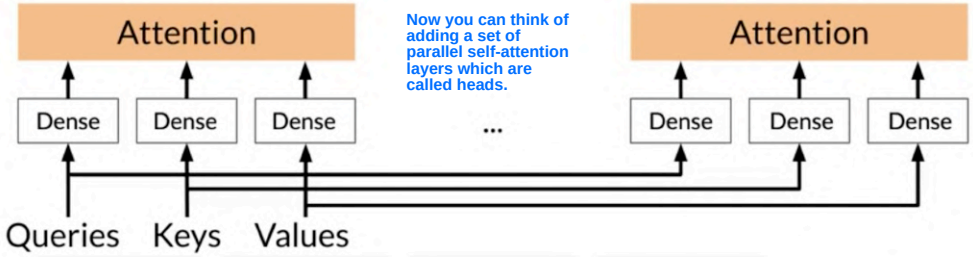
Transformer differs from sequence to sequence by using multi-head attention layers instead of recurrent layers. To understand the intuition of multi-head attention, let's first recall self-attention. Self-attention uses directly its query, key, and value obtained from every sequential inputs. Finally, it's out puts a same length sequential output for each input.

RNNs vs Transformer: Multi-headed attention



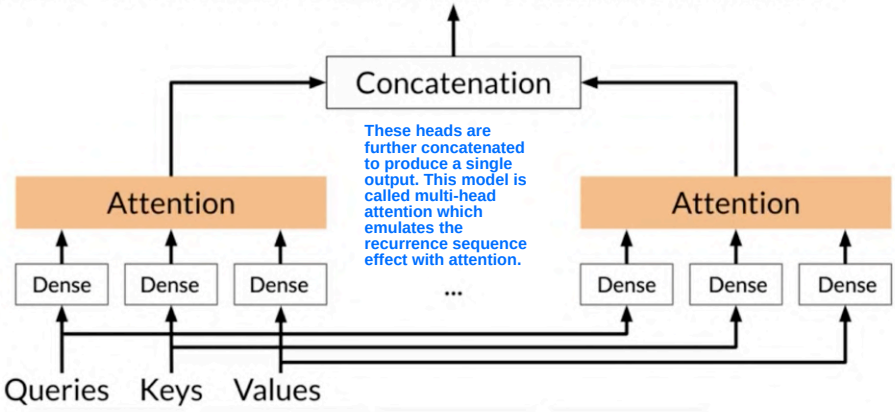
Self-attention uses directly its query, key, and value obtained from every sequential inputs. Finally, it's out puts a same length sequential output for each input. Self-attention can be understood as an intention model that incorporates a dense layer for every input, queries, keys, and values.

RNNs vs Transformer: Multi-headed attention



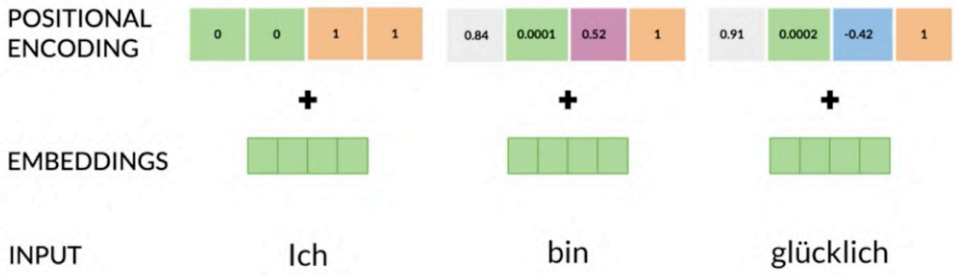
Now you can think of adding a set of parallel self-attention layers which are called heads.

RNNs vs Transformer: Multi-headed attention



These heads are further concatenated to produce a single output. This model is called multi-head attention which emulates the recurrence sequence effect with attention.

RNNs vs Transformer: Positional Encoding



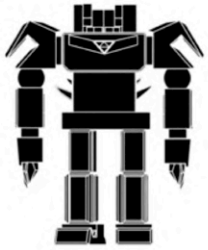
Transformers also incorporates a positional encoding stage which encodes each inputs position in the sequence since the words order and position is very important for any language. For instance, let's suppose you want to translate from German the phrase ich bin glucklich.

Now to capture the sequential information, the transformers use a positional encoding to retain the positional information of the input sequence. The positional encoding out puts values to be added to the embeddings. That's where every input word that is given to the model you have some of the information about it's order and the position. In this case, a positional coding vector for each word I-C-H B-I-N, and glucklich will have some information which will tell us about their respective positions.

Unlike the recurrent layer, the multi-head attention layer computes the outputs of each inputs in the sequence independently then it allows us to parallelize the computation. But it fails to model the sequential information for a given sequence. That is why you need to incorporate the positional encoding stage into the transformer model.

Summary

- In RNNs parallel computing is difficult to implement
- For long sequences in RNNs there is loss of information
- In RNNs there is the problem of vanishing gradient
- Transformers help with all of the above

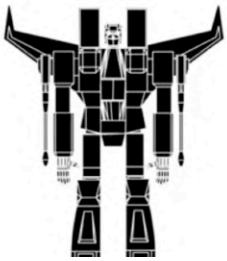


In summary, RNNs have some problems that's come from their sequential structure. With RNNs, it is hard to fully exploit the advantages of parallel computing. For long sequences, important information might get lost within the network, and vanishing gradients problems arise. But fortunately recent research has found ways to solve for the shortcomings of RNNs by using transformers. Transformers are a great alternative to RNNs that help overcome these problems in NLP, and in many fields that processed sequence data. Now you understand why RNNs can be slow and can have problems with big contexts. These are the cases where transformer can help.

Outline

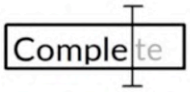
- Transformers applications in NLP
- Some Transformers
- Introduction to T5

Transformer is one of the most versatile deep learning models. It is successfully applied to a number of tasks, both in NLP and beyond. Let me show you a few examples. First, I'll mention the most popular applications of Transformers in NLP. Then you will learn what are the states of the art Transformer models, including the so-called Text-to-Text Transfer Transformer, T5 in shorthand. Finally, you'll see how useful and versatile T5 is.



Transformer NLP applications

Text summarization



Auto-Complete

Named entity recognition (NER)

Question answering (Q&A)

Translation

Chat-bots

Other NLP tasks

- Sentiment Analysis
- Market Intelligence
- Text Classification
- Character Recognition
- Spell Checking

State of the Art Transformers

Radford, A., et al. (2018)
Open AI

GPT-2: Generative Pre-training for Transformer

GPT-2, which stands for Generative Pre-training for Transformer, is a Transformer created by open AI with pre training. It is so good at generating text that the news magazine, The Economist, had a reporter asked the GPT-2 model questions, as if they were interviewing a person. And they published the interview at the end of 2019.

State of the Art Transformers

Radford, A., et al. (2018) Open AI	GPT-2: Generative Pre-training for Transformer
Devlin, J., et al. (2018) Google AI Language	BERT:Bidirectional Encoder Representations from Transformers

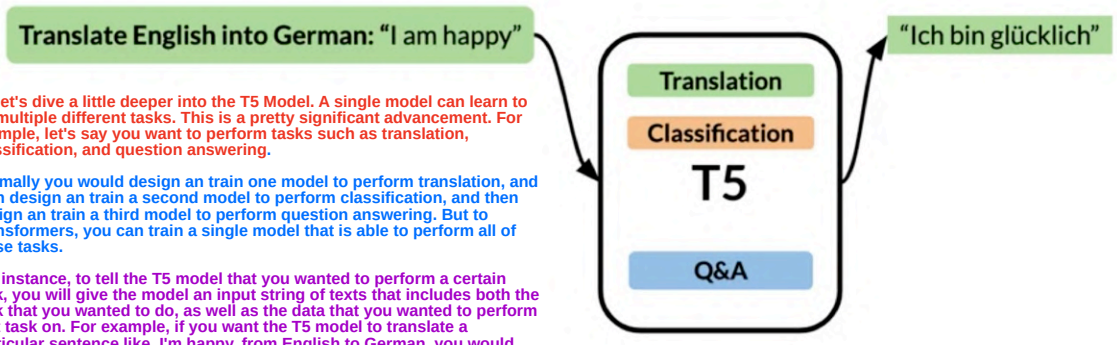
BERT, which stands for Bidirectional Encoder Representations from Transformers, and which was created by the Google AI Language team, is another famous transformer used for learning text representations.

State of the Art Transformers

Radford, A., et al. (2018) Open AI	GPT-2: Generative Pre-training for Transformer
Devlin, J., et al. (2018) Google AI Language	BERT:Bidirectional Encoder Representations from Transformers
Colin, R., et al. (2019) Google	T5: Text-to-text transfer transformer

T5, which stands for Text-to-Text Transfer Transformer, and was also created by Google. As a multi task Transformer, this can do question answering among the lots of different tasks.

T5: Text-To-Text Transfer Transformer

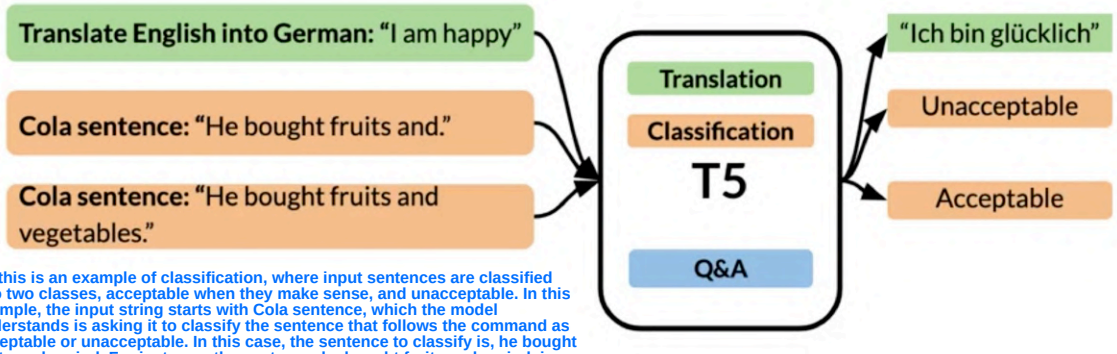


So let's dive a little deeper into the T5 Model. A single model can learn to do multiple different tasks. This is a pretty significant advancement. For example, let's say you want to perform tasks such as translation, classification, and question answering.

Normally you would design an train one model to perform translation, and then design an train a second model to perform classification, and then design an train a third model to perform question answering. But to Transformers, you can train a single model that is able to perform all of these tasks.

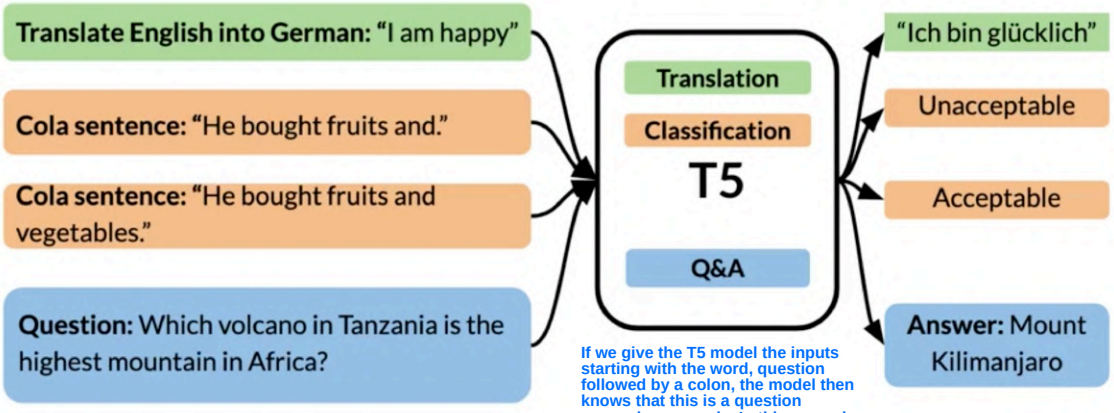
For instance, to tell the T5 model that you wanted to perform a certain task, you will give the model an input string of texts that includes both the task that you wanted to do, as well as the data that you wanted to perform that task on. For example, if you want the T5 model to translate a particular sentence like, I'm happy, from English to German, you would give the input string translate English into German, I'm happy. And the model would be able to output the sentence, ich bin glücklich, which is the translation of, I'm happy into German. So as you can see, we started with, I'm happy, and we got the German equivalent.

T5: Text-To-Text Transfer Transformer



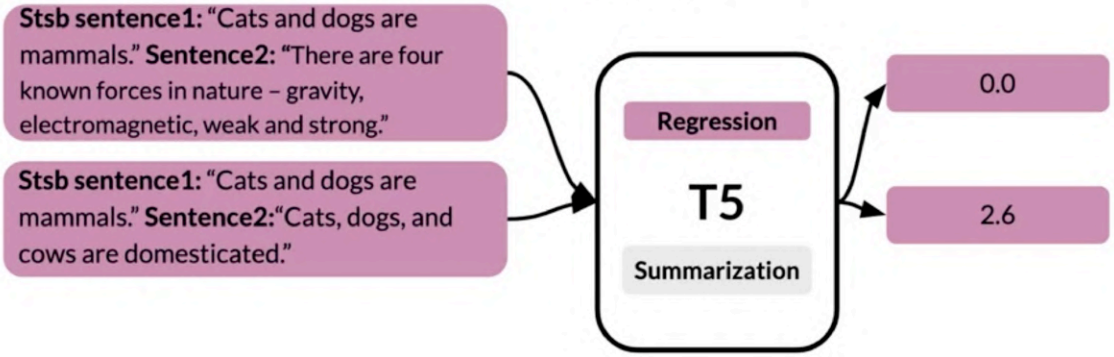
So this is an example of classification, where input sentences are classified into two classes, acceptable when they make sense, and unacceptable. In this example, the input string starts with Cola sentence, which the model understands is asking it to classify the sentence that follows the command as acceptable or unacceptable. In this case, the sentence to classify is, he bought fruits and period. For instance, the sentence, he bought fruits and period, is incomplete. And then it's classified as unacceptable. Meanwhile, if we give the T5 model this input, Cola sentence, he bought fruits and vegetables. The model classifies, he bought fruits and vegetables, as an acceptable sentence.

T5: Text-To-Text Transfer Transformer



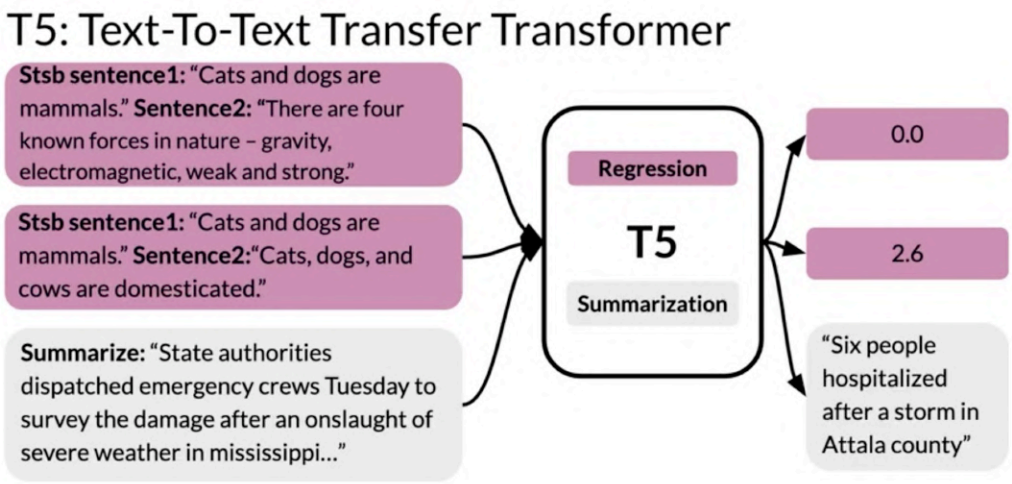
If we give the T5 model the inputs starting with the word, question followed by a colon, the model then knows that this is a question answering example. In this example, the question is, which volcano in Tanzania is the highest mountain in Africa? And your T5 will ask for the answer to that question, which is Mount Kilimanjaro. And remember, that's all of these tasks are done by the same model with no modification other than the input sentences.

T5: Text-To-Text Transfer Transformer



T5 also performs tasks of regression and summarization. Recall that's a regression model is one that's outputs a continuous numeric value. Here you can see an example of regression, which outputs the similarity between two sentences. The start of the input string is Stsb, which indicates to the model that it should perform a similarity measurement between two sentences. The two sentences are denoted by the word sentence one and sentence two. The range of possible outputs for this model is any numeric value ranging from zero to five. Where zero indicates that the sentence is are not similar at all, and five indicates that the sentences are very similar.

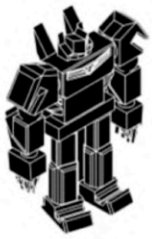
Let's consider this example. When comparing sentence one, cats and dogs are mammals, with the sentence two, there are four known forces in nature, gravity, electromagnetic, weak and strong. The resulting similarity level is zero, indicating that the sentences are not similar.



Summary

and details on an onslaught of severe weather in Mississippi, which is summarized just as, six people hospitalized after a storm in Attala County.

- Transformers are suitable for a wide range of NLP applications
- GPT-2, BERT and T5 are the cutting-edge Transformers
- T5 is a powerful multi-task transformer



Outline

- Introducing attention (Translation example)
- Mathematics behind Attention

The main operation in transformer is the dot product attention. In this section, I'll remind you how it works before we move to more complex attention variants. First I'll introduce the concept of attention using an example of a translation task. Then you'll see some details about the math behind attention.

Introducing attention - Translation example

- A query (German word) looks for similar keys (English words).

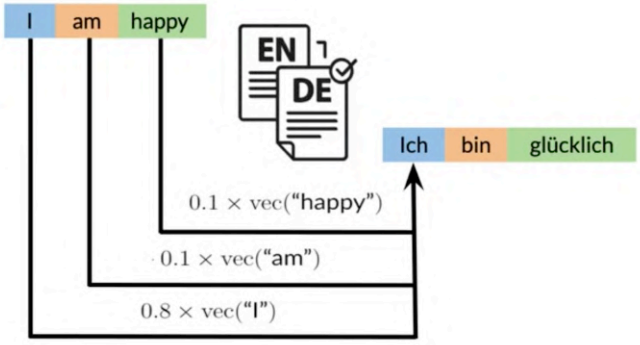
To understand what's attention, does think of the following translation task. You need to translate an English sentence to German. You've seen in previous notes how to make good word embeddings, so that's vectors for English and German words that mean the same or similar.

For instance, let's consider the example sentence. I am happy which is translated into the German equivalent. So when you have a German word you want to look at the English sentence and find all places with similar words. This is what attention does for you.

In an attention layer, the German word vectors are called queries. Because they initiate the look up, a query is matched against all keys and each guy gets probability that it's a match for the query. For instance, to translate the sentence, I'm happy to German, you can compare the word I from English with the following word I-c-h over here, Ich.

Introducing attention - Translation example

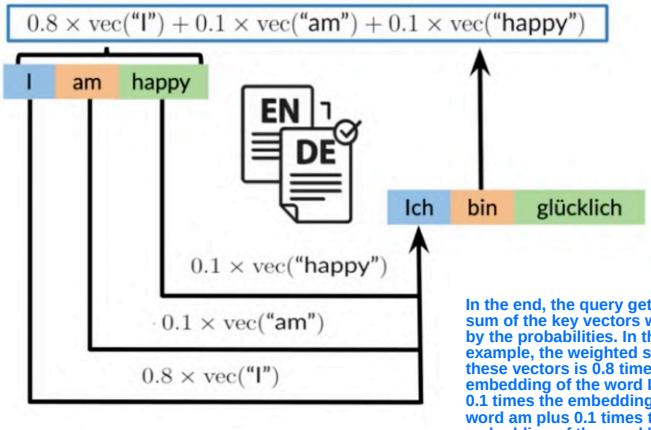
- A query (German word) looks for similar keys (English words).
- Each key has a probability of being a match for the query.



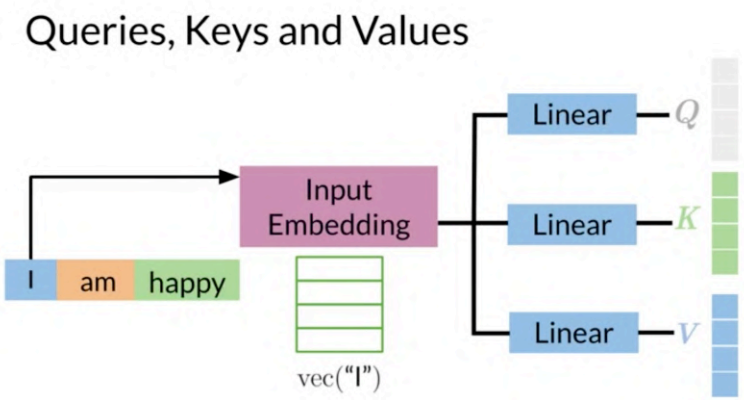
Then you can compare the word am with the word Ich and finally you can compare the word happy with the word Ich. These queries by keys matrix is often called attention weights and it says how much each key is similar to each query. For example, the vector of the word Ich maybe similar to the vector for I and get zero points its probability, while it's not very similar to am or happy, but still a little bit and get 0.1 probability for both.

Introducing attention - Translation example

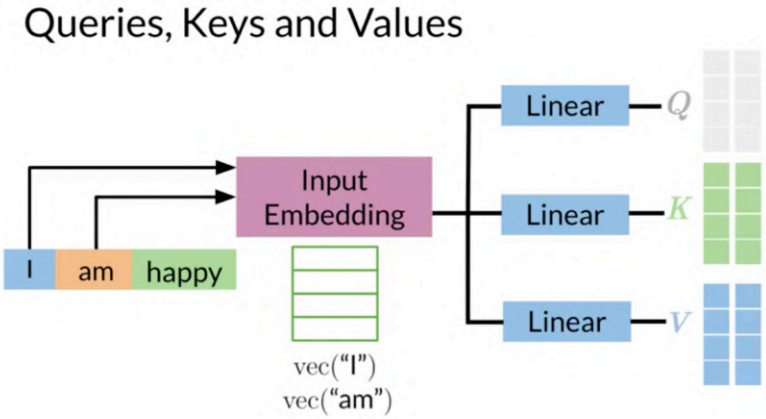
- A query (German word) looks for similar keys (English words).
- Each key has a probability of being a match for the query.
- Return sum of the keys weighted by their probabilities.



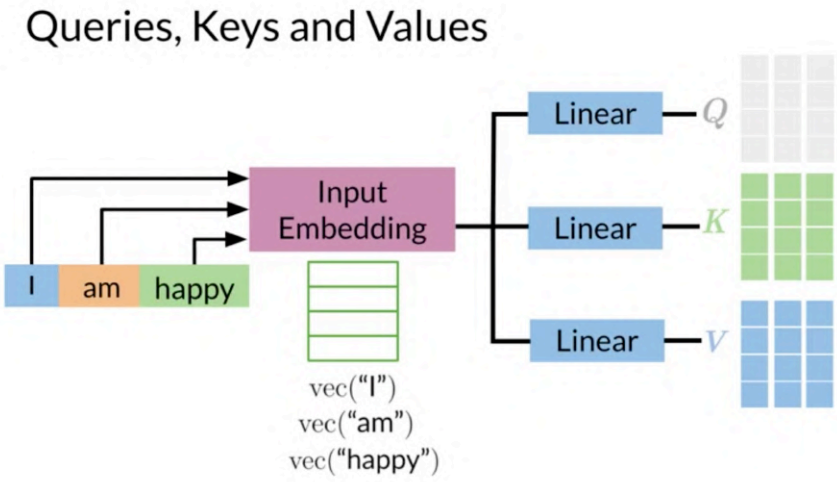
In the end, the query gets the sum of the key vectors waited by the probabilities. In the example, the weighted sum of these vectors is 0.8 times the embedding of the word I plus 0.1 times the embedding of the word am plus 0.1 times the embedding of the word happy.



Before you dive into the concept of attention, you need to first define three main matrices, capital Q, which stands for queries, capital K which stands for Keys and Capital V, which stands for values. So understand how to create these matrices. Let's consider the phrase in English, I am happy. First, the word I is embedded, to obtain a vector representation that holds continuous values which is unique for every single word as seen earlier in the course. By feeding three distinct linear layers, you get three different vectors for queries, keys and values.



Then you can do the same for the word am to output a second vector.



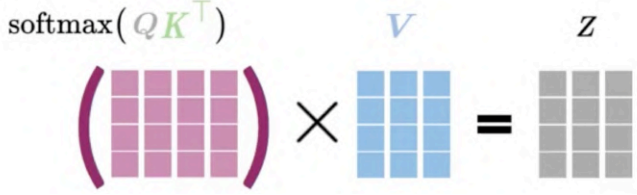
And finally the word happy to get a third vector and form the queries, keys and values matrix.

Concept of attention

From both the capital Q matrix and the capital K matrix and attention model calculates weights or scores representing the relative importance of the keys for a specific query. These attention weights can be understood as alignments course as they come from a dos product.

Additionally, to turn these weights into probabilities, a softmax function is required.

Finally, multiplying these probabilities with the values, you will then get a weighted sequence, which is the attention results itself.

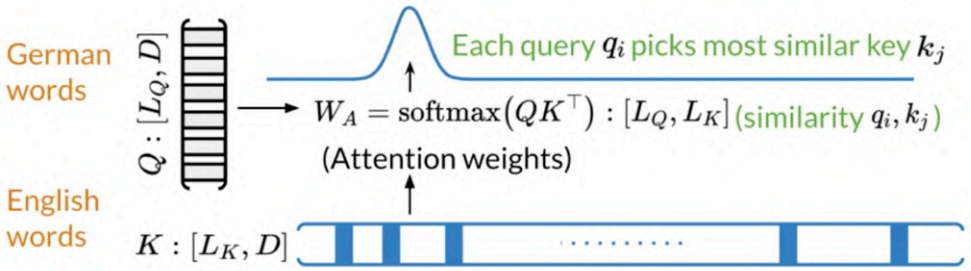


Attention math

The input to attention are queries, keys and values. Often values are the same as keys. Queries are vectors of dimension T and there is a number LQ of them. Keys are vectors of the same dimension D and there is a number LK of them. Think of keys as the embeddings of English words. And the queries as the embeddings of German words. Then D is the dimensionality of word embeddings. LQ, the length of the English sentence and LQ, number of words of the German senses. So queries are a tensor Q of shape LQ by D and keys have a shape LK by D. As I said before, a query Q will assign each key a probability that the key K is a match for Q.

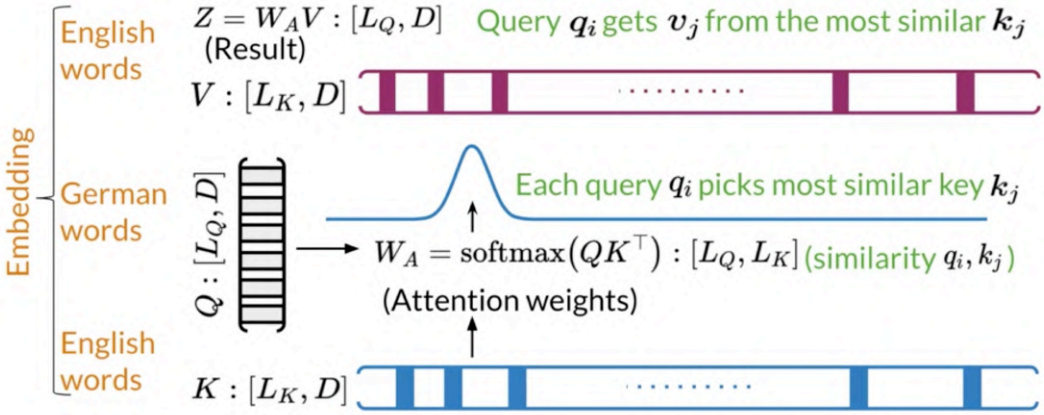
You measure the similarity by taking those products of vectors. So Q and K are similar if Q dot K is large. However, the similarity numbers do not add up to one, so they cannot be used as probabilities.

To make them so and to make attention more focused on the best matching keys, use the softmax. So you compute the matrix of query key probabilities, often called the attention weights, just as softmax of between Q&K transpose. This matrix has shaped LQ by LK.



Each query keeper gets a probability in the final step you take the values, which is another matrix of the same shape as keys. And often the same as keys, and we want to get a weighted sum, weighting each value vi by the probability that the key ki matches the query. This can be computed very efficiently just as matrix multiplication, we multiply attention weights W sub S by the values V, and that's it. And attention mechanism calculates the dynamic or alignment weights representing the relative importance of the inputs in this sequence. Which are the keys for that particular outputs, which is the query. Multiplying the dynamic weights or the alignments course with the input sequence the values will then weight the sequence. A single context vector can then be calculated using the sum of weighted vectors.

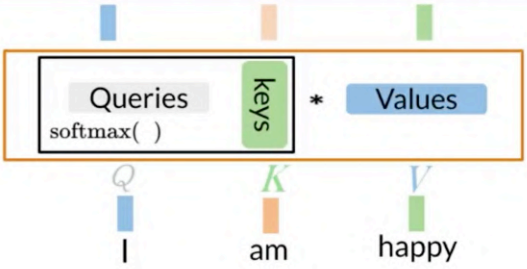
Attention math



Attention formula

$$Z = \text{attention}(Q, K, V) = \text{softmax}(QK^\top)V = W_A V$$

(Result)



Matrix Z is our final results, which is equal to the attention function of matrices Q, K and V. Which equals the softmax of the dot product between Q and K transpose times V. And in turn it is equal to W sub A times V. So the whole attention operation can be written in this simple equation which is the softmax of the dot product between QK transpose times V.

It is important and very simple. In practice, this means that you can compute attention by performing two matrix multiplications. Q times K transpose and then by V and once after softmax operation.

Modern hardware such as GPUs and CPUs are very fast as matrix multiplication, which makes attention very fast and such a great operation to use.

Summary

- Dot-product Attention is essential for Transformer
- The input to Attention are queries, keys, and values
- A softmax function makes attention more focused on best keys
- GPUs and TPUs is advisable for matrix multiplications

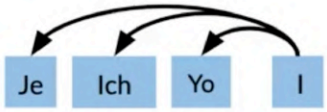
Outline

- Ways of Attention
- Overview of Causal Attention
- Math behind causal attention

In this section, we'll see how to make attention work for predicting one symbol after another, which we call the causal attention.

Three ways of attention

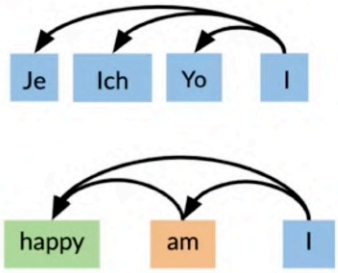
- **Encoder/decoder attention**: One sentence (decoder) looks at another one (encoder)



One is the encoder decoder attention. So when one sentence, for example German, attends to another one such as English. You've already used this kind of attention in the translation model.

Three ways of attention

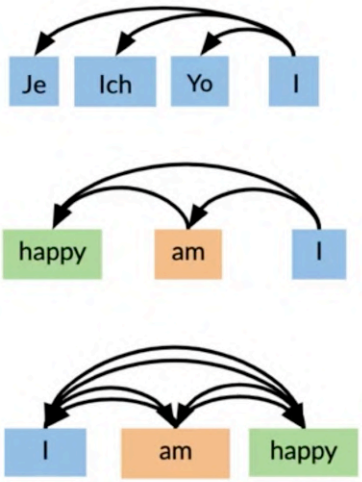
- **Encoder/decoder attention**: One sentence (decoder) looks at another one (encoder)
- **Causal (self) attention**: In one sentence, words look at previous words (used for generation)



The next is causal attention, where in the same sentence, words attend to words in the past. This kind of attention can be used for generating text, for example summaries.

Three ways of attention

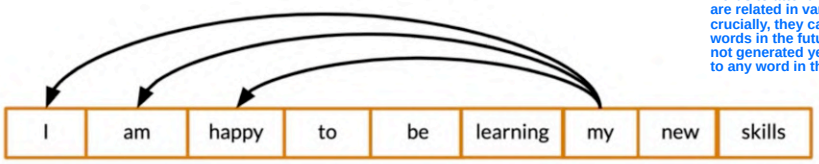
- **Encoder/decoder attention**: One sentence (decoder) looks at another one (encoder)
- **Causal (self) attention**: In one sentence, words look at previous words (used for generation)
- **Bi-directional self attention**: In one sentence, words look at both previous and future words



The final one is bi-directional self-attention, where words in the same sentence look both at previous and future words. This is used in models like BERTs and T5, that have a masking glass

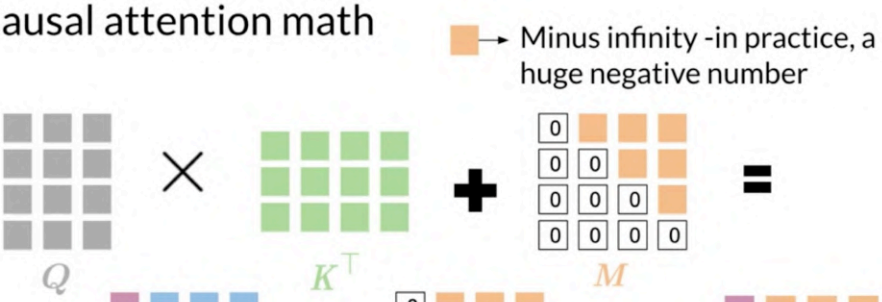
Causal attention

- Queries and keys are words from the same sentence
- Queries should only be allowed to look at words before



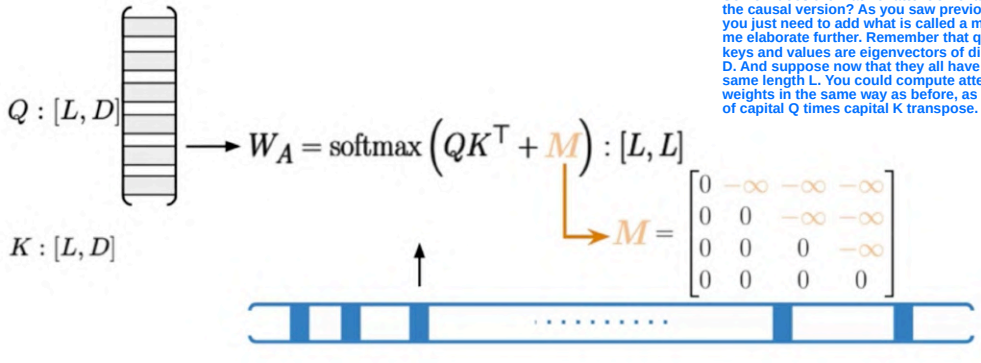
The final one is bi-directional self-attention, where words in the same sentence look both at previous and future words. This is used in models like BERTs and T5, that have a masking glass. You already saw the case when queries come from one sentence and keys from the other. In causal attention, queries and keys come from the same sentence. That is why it is often referred to as self-attention. It is used in language models where you are trying to generate sentences. For example, take the sentence, I'm happy to be learning my new skills. After generating the word my, the model may want to look at the vector for I to retrieve more information. In general, causal attention allows words to attend to other words that are related in various ways, but crucially, they cannot attend to words in the future since these were not generated yet, they can attend to any word in the past though.

Causal attention math



To understand the math behind causal attention, let's recall the dot-product attention, which requires the calculation of capital Q over here, times capital K transpose.And then this gives you your weights course, so Q K transpose. These weights are added to a constant matrix which is called mask, and it's a triangular matrix. It has 0s on the diagonal and below it, and minus infinity, or in practice just a huge negative number in all other places. So you get the new weights as Q times K transpose + M.

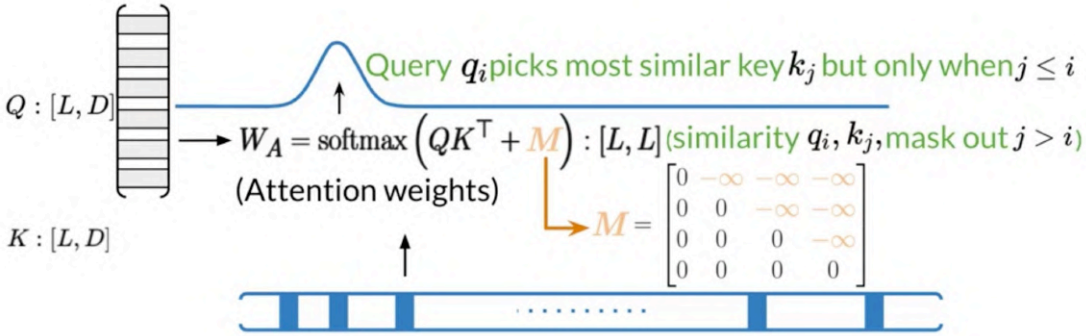
Causal attention math



So how does the math of attention change for the causal version? As you saw previously, you just need to add what is called a mask. Let me elaborate further. Remember that queries, keys and values are eigenvectors of dimension D. And suppose now that they all have the same length L. You could compute attention weights in the same way as before, as softmax of capital Q times capital K transpose.

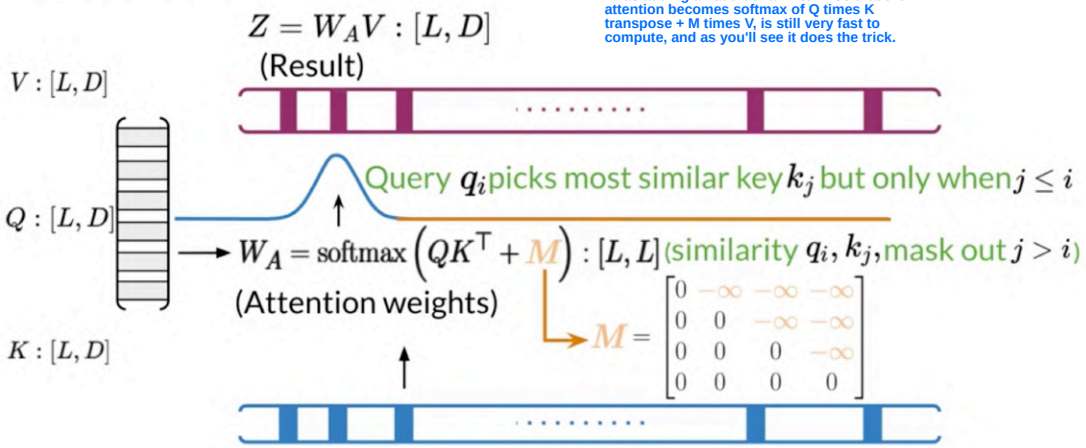
Causal attention math

But that way, you are allowing the model to attend to words in the future. To solve this issue, you add a mask by a sum of size L by L. So you compute softmax of Q times K transpose + M. When you add M to Q times K transpose, all values on the diagonal and below which correspond to queries attending words in the past are untouched. All other values become minus infinity. After a softmax, all minus infinities will become equal to 0, as exponents of negative infinity is equal to 0, so it prevents words from attending to the future.



Causal attention math

The final formula for causal attention is therefore very similar to the attention formula you've learned in the previous video. I was adding an additional M term so that the attention becomes softmax of Q times K transpose + M times V, is still very fast to compute, and as you'll see it does the trick.



Summary

- There are three main ways of Attention: Encoder/Decoder, Causal and Bi-directional type
- In causal attention, queries and keys come from the same sentence and queries search among words before only

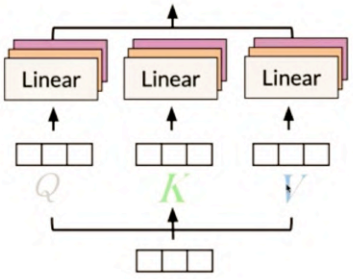
Outline

- Intuition of Multi-Head Attention
- Scaled dot-product and concatenation
- Multi-Head Attention formula

Multi-Head Attention

- Each head uses different linear transformations to represent words

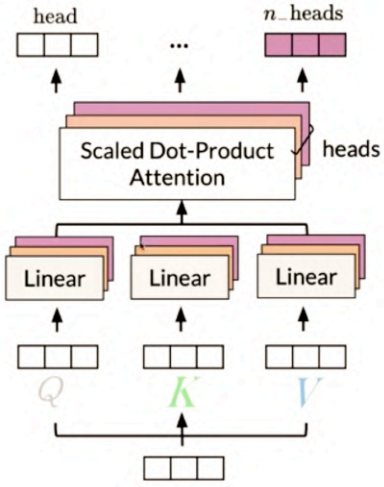
In a Multi-head attention, each head uses individually linear transformations to represent words. A single head works like this. For instance, here you have the word, I am happy. Then you first need to calculate the embedding and you get a vector. This can be linearly transformed to get the queries, matrix, the keys over here, and then the



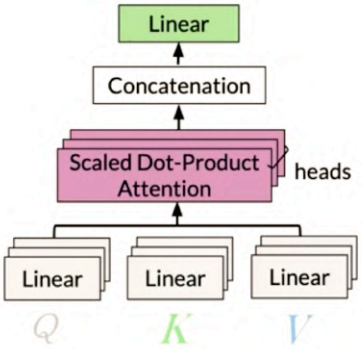
Multi-Head Attention

- Each head uses different linear transformations to represent words
- Different heads can learn different relationships between words

Also when you use a multi-head attention, a head can learn different relationships between words from another head. So over here you have N different heads, over here you can see that's how the heads are being concatenated, and each one of these will result in one head.



Multi-Head Attention - Overview



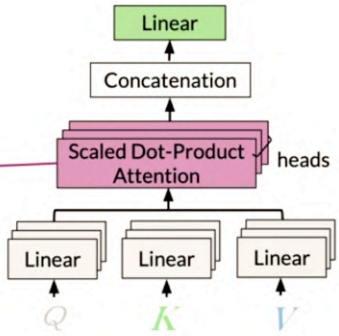
Let me show you how multi-headed attention achieves multiple lookups in parallel. The input to multi-head attention again is a triple Q, K, V. So query, key and value. To achieve the multiple lookups, you first use a fully-connected, dense linear layer on each query, key, and value. This layer will create the representations for parallel attention heads. Here, you split these vectors into number of heads and perform attention on them as each head was different. Then the result of the attention will be concatenated back together, and put through a final fully connected layer. The scale dot-product is the one used in the dot-product attention model except by the scale factor, one over square root of DK.

Multi-Head Attention - Scaled dot product

DK is the key inquiry dimension. It's normalization prevents the gradients from the function to be extremely small when large values of D sub K are used.

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Query and key dimension



Multi-Head Attention - Concatenation

- Input(Q, K, V): [batch, length, d_model]

512, 1024

Now, let me show you how this intuition looks in practice.

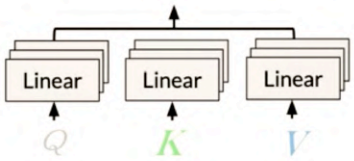
The inputs Q, K, V are all of shape, batch size, length by D model. Usually D model is 512 or 1024. Sometimes less or more but of this order.

Multi-Head Attention - Concatenation

- Input(Q, K, V): `[batch, length, d_model]`
- Linear layer: `[batch, length, n_heads * d_head]`

4, 6, 16, ...

The linear layers acts on the final dimension, and change it to N head by D head. N heads is the number of parallel attention heads, often four, eight or even 16 or more.

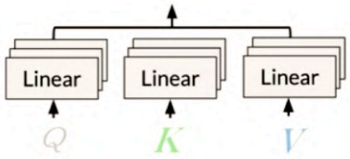


Multi-Head Attention - Concatenation

- Input(Q, K, V): `[batch, length, d_model]`
- Linear layer: `[batch, length, n_heads * d_head]`

4, 6, 16, ... 64, 128

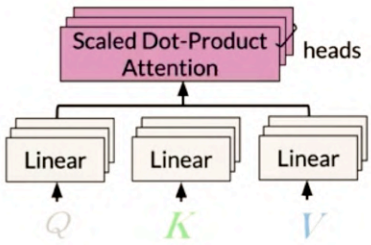
While D head is the dimensionality of the vector for each head, very often 64 or 128. So the linear layer creates a set of N head vectors, each of size D head.



Multi-Head Attention - Concatenation

- Input(Q, K, V): `[batch, length, d_model]`
- Linear layer: `[batch, length, n_heads * d_head]`
- Transpose: `[batch, n_heads, length, d_head]`
- Apply attention treating `n_heads` like batch

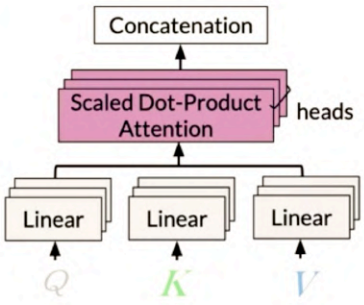
To apply attention on all heads in parallel, you transpose the N heads parts just after the batch dimension so it can be treated in the same way as batch. Separate heads just don't interact with each other. After this transposition, you apply the dot-product attention you already know in the same way as before.



Multi-Head Attention - Concatenation

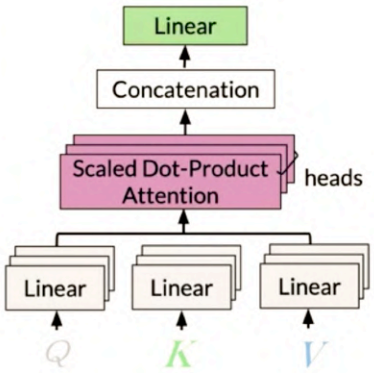
- Input(Q, K, V): `[batch, length, d_model]`
- Linear layer: `[batch, length, n_heads * d_head]`
- Transpose: `[batch, n_heads, length, d_head]`
- Apply attention treating `n_heads` like batch
- Result shape: `[batch, n_heads, length, d_head]`
- Transpose: `[batch, length, n_heads * d_head]`

Then the result is reshaped and concatenated, and transposed back to get a matrix in size of N heads times D head.

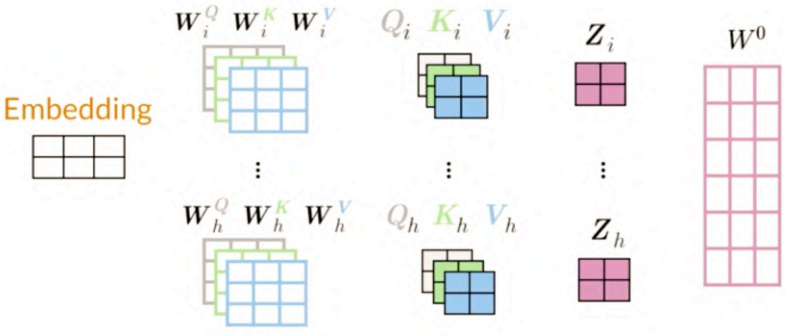


Multi-Head Attention - Concatenation

- Input(Q, K, V): `[batch, length, d_model]`
- Linear layer: `[batch, length, n_heads * d_head]`
- Transpose: `[batch, n_heads, length, d_head]`
- Apply attention treating `n_heads` like batch
- Result shape: `[batch, n_heads, length, d_head]`
- Transpose: `[batch, length, n_heads * d_head]`
- Linear layer into: `[batch size, length, d_model]`



Multi-Head Attention math



To understand the math behind multi-head attention, let's see this step-by-step explanation. As any conventional attention model, you first need to create the embedding of the input words.

Next, you calculate the queries, keys and value matrices, capital Q, capital K, and capital V. This is done by packing all the embeddings into a matrix for every head, from one to H heads.

Then you train weight matrices one per head to obtain new weighted queries, keys and values as capital W superscripts Q, capital W superscript K and capital W superscript V. Note that the subscripts I tells you that this corresponds to the ith head.

With these weighted queries, keys and value matrices, you calculate the results matrix capital Z subscript I from I equals one to H, where H is the number of heads.

Finally, by multiplying by the outputs weight, matrix capital W, superscript 0, you obtain the output layer Z.

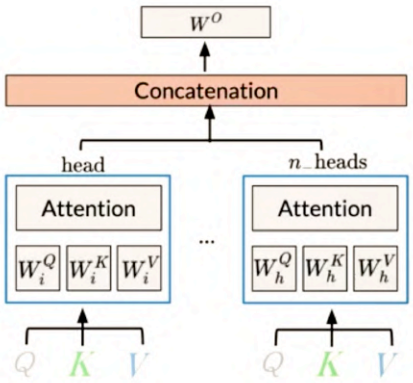
Multi-Head Attention Formula

Now I'll show you how the multi-head attention works. A multi-headed model jointly attends to information from different representations at different positions over the projected versions of queries, keys, and values. You have to apply the attention function in parallel, reaching different output values. These output values are then concatenated and weighted.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(h_1, \ldots, h_h)W^0$$

where $h_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

Each head h_i is the attention function of **Query, Key and Value** with trainable parameters (W_i^Q, W_i^K, W_i^V)



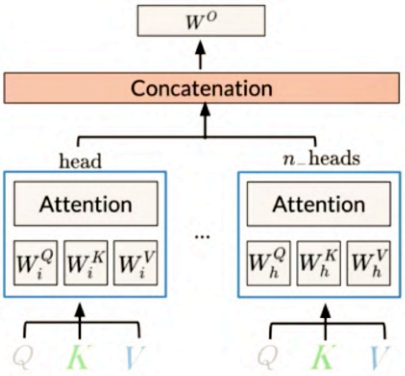
Multi-Head Attention Formula

Then the multi-head attention can be summarized in this formula. The multi-head of Q, K and V equals to the concatenation of H subscript one all the way to H subscript H times W superscript 0.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(h_1, \ldots, h_h)W^0$$

where $h_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

Each head h_i is the attention function of **Query, Key and Value** with trainable parameters (W_i^Q, W_i^K, W_i^V)



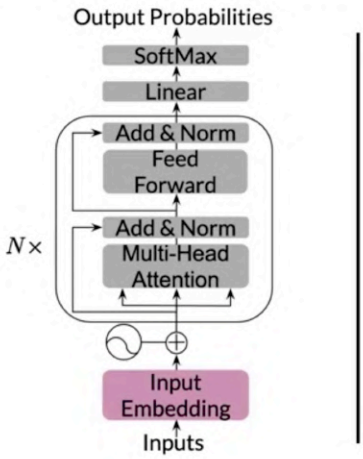
Summary

- Different heads can learn different relationship between words
- Scaled dot-product is adequate for Multi-Head Attention
- Multi-Headed models attend to information from different representations at different positions

Outline

- Overview of Transformer decoder
- Implementation (decoder and feed-forward block)

Transformer decoder



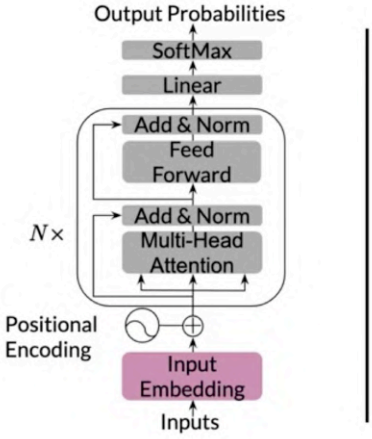
Overview

- input: sentence or paragraph
  - we predict the next word

Once you know attention, it's a fairly simple model as you'll see. In this section, you'll see the basic structure of a transformer decoder. I'll show you the definition of a transformer and how to implement the decoder and the feed forward blocks.

So on the left, you can see a picture of the transformer decoder. As input, it gets a tokenized sentence, a vector of integers as usual. The sentence gets embedded with word embeddings, which you know quite well by now.

Transformer decoder

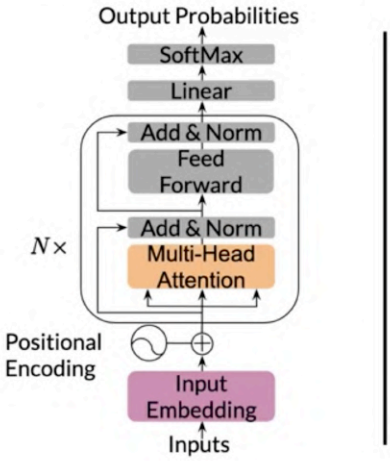


Overview

- input: sentence or paragraph
  - we predict the next word
- sentence gets embedded, add positional encoding
  - (vectors representing $\{0, 1, 2, \ldots, K\}$)

Then you add to these embeddings the information about positions. This information is nothing else than learned vectors representing 1, 2, 3 and so on, up to some maximum length that we'll put into the model. So the embedding of the first word will get added with the vector presenting one. The embedding of the second word with the vector representing two, and so on.

Transformer decoder

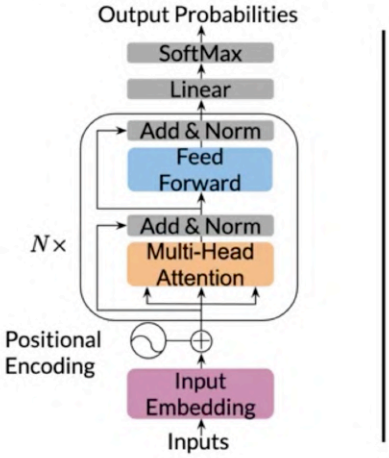


Overview

- input: sentence or paragraph
  - we predict the next word
- sentence gets embedded, add positional encoding
  - (vectors representing $\{0, 1, 2, \ldots, K\}$)
- multi-head attention looks at previous words

Now, this constitutes the inputs for the first multi headed attention layer.

Transformer decoder

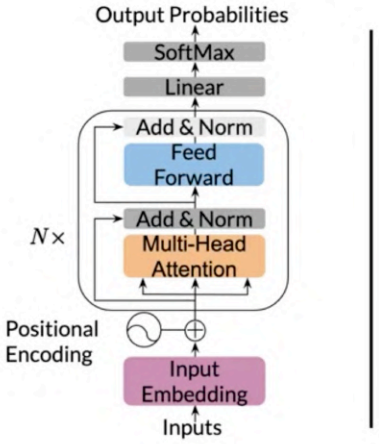


Overview

- input: sentence or paragraph
  - we predict the next word
- sentence gets embedded, add positional encoding
  - (vectors representing $\{0, 1, 2, \ldots, K\}$)
- multi-head attention looks at previous words
- feed-forward layer with ReLU
  - that's where most parameters are!

After the attention layer, you have a feed-forward layer which operates on each position independently.

Transformer decoder

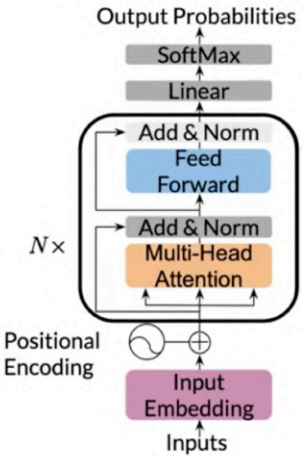


Overview

- input: sentence or paragraph
  - we predict the next word
- sentence gets embedded, add positional encoding
  - (vectors representing $\{0, 1, 2, \ldots, K\}$)
- multi-head attention looks at previous words
- feed-forward layer with ReLU
  - that's where most parameters are!
- residual connection with layer normalization

After each attention and feed-forward layer, you put a residual or skip connection. So just add the inputs of that layer to its output and then perform layer normalization.

Transformer decoder

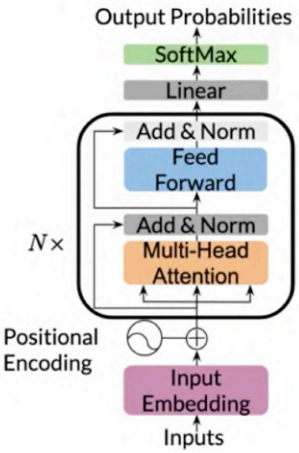


Overview

- input: sentence or paragraph
  - we predict the next word
- sentence gets embedded, add positional encoding
  - (vectors representing $\{0, 1, 2, \ldots, K\}$)
- multi-head attention looks at previous words
- feed-forward layer with ReLU
  - that's where most parameters are!
- residual connection with layer normalization
- repeat N times

The original model starts with N equals 6, but now transformers go up to 100 or even more.

Transformer decoder

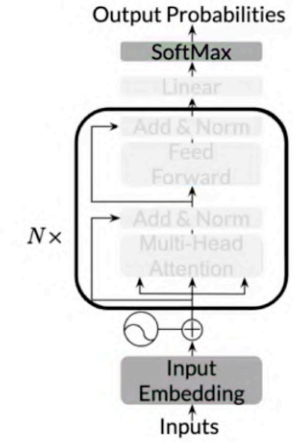


Overview

- input: sentence or paragraph
  - we predict the next word
- sentence gets embedded, add positional encoding
  - (vectors representing $\{0, 1, 2, \ldots, K\}$)
- multi-head attention looks at previous words
- feed-forward layer with ReLU
  - that's where most parameters are!
- residual connection with layer normalization
- repeat N times
- dense layer and softmax for output

And then you have a final dense layer for outputs and a softmax layer, and that's it.

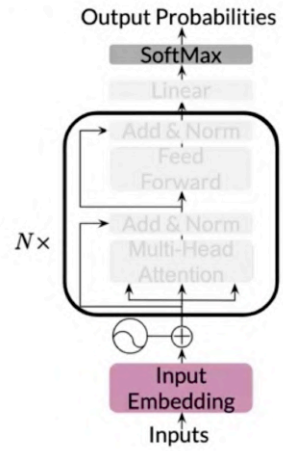
Transformer decoder



Explanation

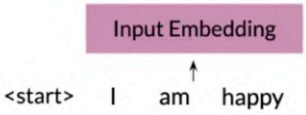
Here on the right, you'll see the core of the transformer model. It has three layers at the beginning.

Transformer decoder

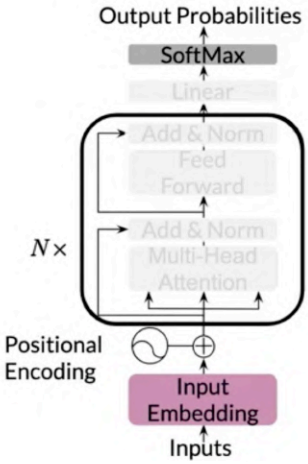


Explanation

Then the shift writes just introduces the start token, which your model will use to predict the next word. You have the embedding, which trains a word to vector embedding.

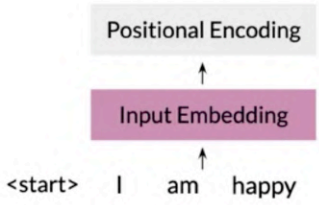


Transformer decoder

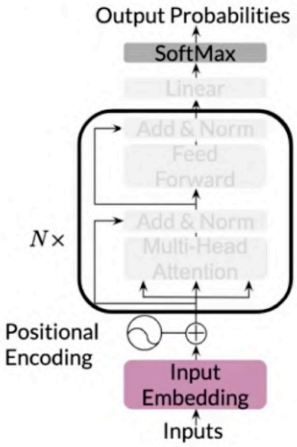


Explanation

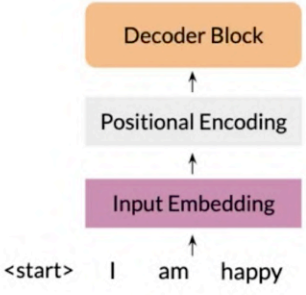
And positional encoding, which trains the vectors for one, two and so on as explained before. If the input to the model was a tensor of shape batch by length, then after the embedding layer it will be tensor of shape batch by length by D model where D model is the size of these embeddings. And they usually go to 512, 1024 and nowadays up to 10K or more.



Transformer decoder

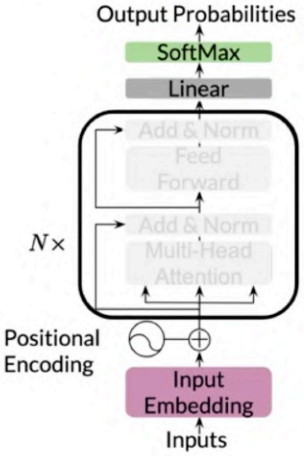


Explanation

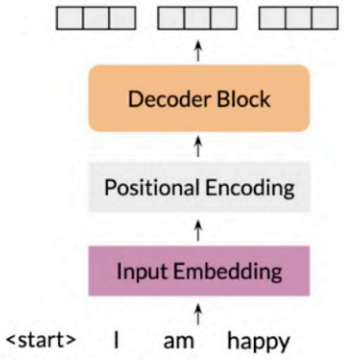


After these early layers, you'll get N decoder blocks.

Transformer decoder



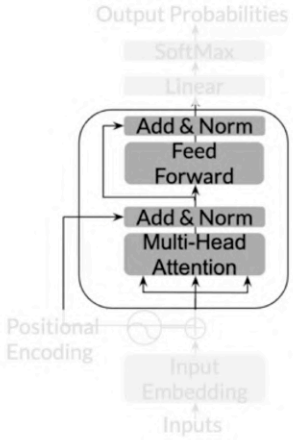
Explanation



Then a fully connected layer that's output sensors of shape batch by length, by vocab size.

Lastly a log softmax for cross entropy loss.

The Transformer decoder

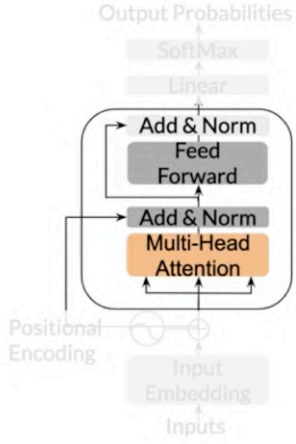


Decoder Block

Now let's see how the decoder block is built. It starts with these sets of vectors as an input sequence which are added to the corresponding positional coding vectors, producing the so-called positional input embedding.

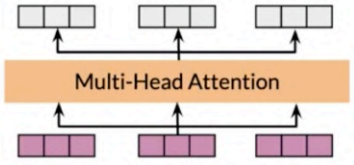
Positional input embedding

The Transformer decoder



Decoder Block

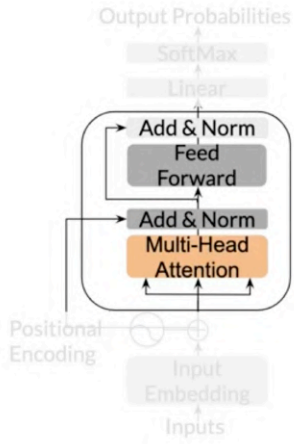
After embedding, the input sequence passes through a multi headed attention model. And while this model processes each word, each position in the input sequence, the attention itself searches other positions in the sequence to help identify relationships. Each of the words in the sequence is weighted.



Output Vector

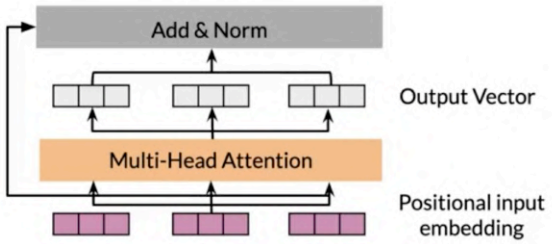
Positional input embedding

The Transformer decoder

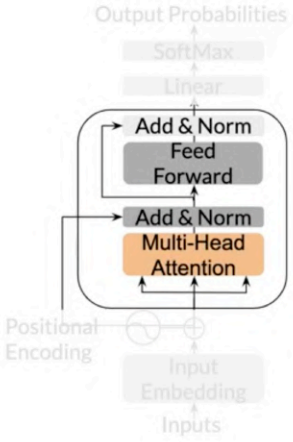


Then in each layer of attention, there is a residual connection around it, followed by a layer normalization step to speed up the training and significantly reduce the overall processing time.

Decoder Block

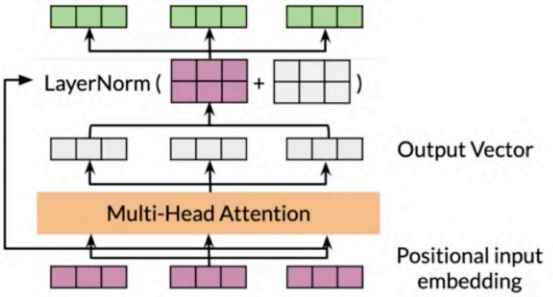


The Transformer decoder

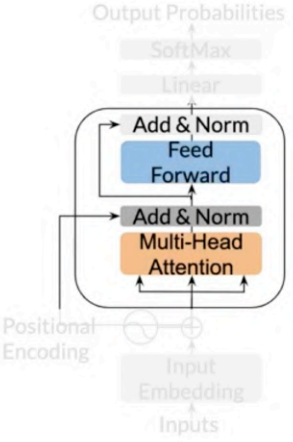


Then each word is passed through a feed-forward layer. That is, embeddings are fed into a neural network.

Decoder Block

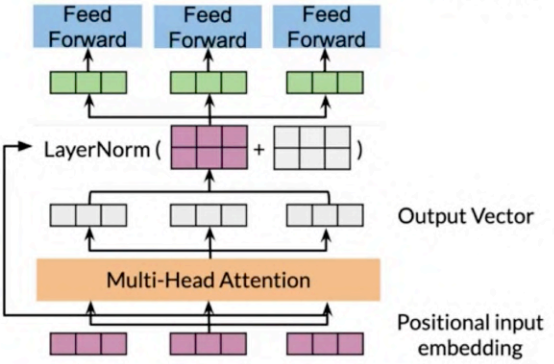


The Transformer decoder

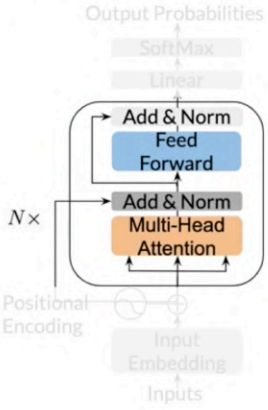


And then you have a drop out at the end as a form of regularization.

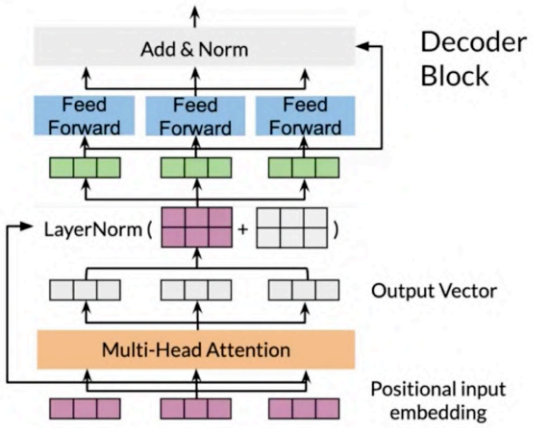
Decoder Block



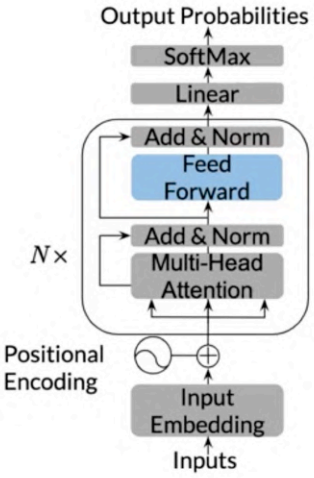
The Transformer decoder



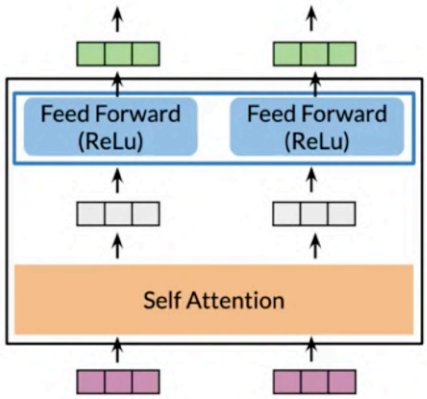
Next, a layer normalization step is repeated N times. Finally, the encoder layer output is obtained.



The Transformer decoder



Feed forward layer



After the attention mechanism and the normalization step, some nonlinear transformations are introduced by including fully connected feed-forward layers with simple but nonlinear ReLu activation functions for each input. And you have shared parameters for efficiency. The feed forward neural network output vectors will essentially replace the hidden states of the original RNN encoder.

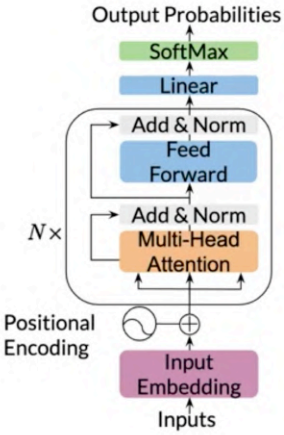
Summary

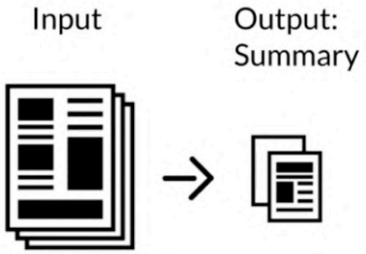
- Transformer decoder mainly consists of three layers
- Decoder and feed-forward blocks are the core of this model code
- It also includes a module to calculate the cross-entropy loss

Outline

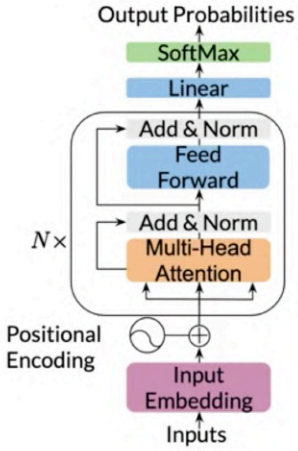
- Overview of Transformer summarizer
- Technical details for data processing
- Inference with a Language Model

Transformer for summarization





Technical details for data processing

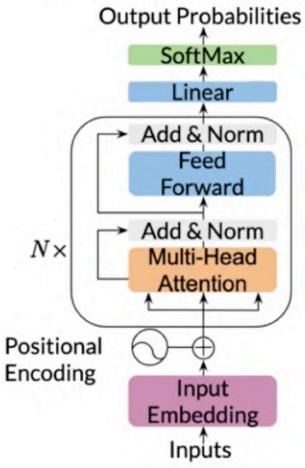


Model Input:

```
ARTICLE TEXT <EOS> SUMMARY <EOS> <pad> ...
```

First, take a look at the problem you will solve in this week's assignments. As input, you get whole news articles. As output, your model is expected to produce the summary of the articles. That is few sentences that's mentioned the most important ideas. To do this you will use the transformer model that I showed you in previous videos. But the one thing may immediately stand out to you. Transformer only takes text as inputs and predict the next word. For summarization, it turns out you just need to concatenate the inputs, in this case, the article, and put the summary after it.

Technical details for data processing



Model Input:

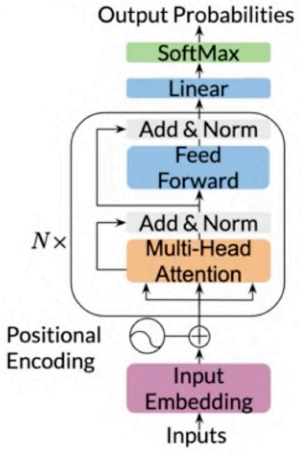
```
ARTICLE TEXT <EOS> SUMMARY <EOS> <pad> ...
```

Tokenized version:

```
[2,3,5,2,1,3,4,7,8,2,5,1,2,3,6,2,1,0,0]
```

Let me show you how. Here is an example of how to create inputs features for training the transformer from an article and its summary. The input for the model is a long text that starts with a news article, then comes the EOS tag, the summary, and then another EOS tag. As usual, the input is tokenized as a sequence of integers. Here 0 denotes padding, and 1 EOS. And all other numbers for the tokens for different words. When you're on the transformer on this input, it will predict the next word by looking at all the previous ones, but you do not want to have a huge loss in the model just because it's not able to predict the correct ones. And that's why you have to use a weighted loss.

Technical details for data processing



Model Input:

```
ARTICLE TEXT <EOS> SUMMARY <EOS> <pad> ...
```

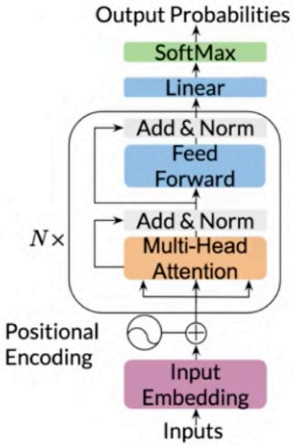
Tokenized version:

```
[2,3,5,2,1,3,4,7,8,2,5,1,2,3,6,2,1,0,0]
```

Loss weights: 0s until the first `<EOS>` and then 1 on the start of the summary.

Instead of averaging the loss for every word in the whole sequence, you weight the loss for the words within the article with 0s, and the ones within the summary with 1s. So the model only focuses on the summary. However, when there is little data for the summaries, it's actually helps to weight the article loss with nonzero numbers say 0.2 or 0.5 or even 1. That way the model is able to learn word relationships that are common in the news. You'll not have to do it for this week's assignment, but it's good that you have this in mind for your own applications.

Cost function



Cross entropy loss

$$J = -\frac{1}{m}\sum_{j}^{m}\sum_{i}^{K} y_j^i \log \hat{y}_j^i$$

j : over summary

i : bach elements

Another way to look this in is by looking at the cost function. Which sums the losses over the words J, within the summary for every example I in the batch. So the cost function is a cross entropy function that ignores the words from the article to be summarized.

Inference with a Language Model

Model input:

`[Article] <EOS> [Summary] <EOS>`

Inference:

- Provide: [Article] <EOS>
- Generate summary word-by-word
  - until the final <EOS>
- Pick the next word by random sampling
  - each time you get a different summary!

Now that you know how to construct the inputs and the model, you can train your transformer summarizer. Recall again, the transformers predict the next word, and your inputs is in news article.

At test or inference time, you will input the article with the EOS token to the model

And ask for the next word, it is the first word of the summary.

Then you will ask for the next word, and the next, and so on until you get the EOS token. When you run your transformer model, it generates a probability distribution over all possible words. You will sample from this distribution. So each time you run this process you'll get a different summary.

Summary

- For summarization, a weighted loss function is optimized
- Transformer Decoder summarizes predicting the next word using
- The transformer uses tokenized versions of the input