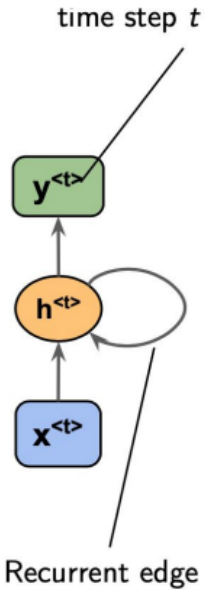
Chapter 08 – Transformers

Author: Gianmarco Scarano

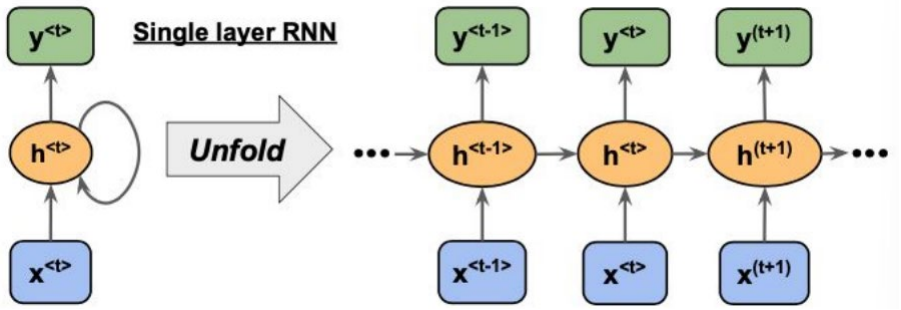
gianmarcoscarano@gmail.com

1. Recurrent Neural Networks

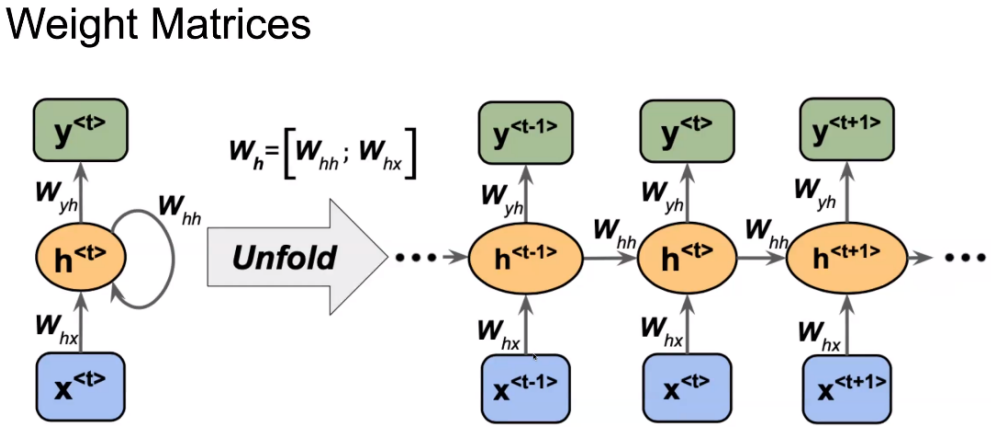
The general idea behind Recurrent Neural Networks (**RNN**) is that it is a neural network that is specialized for processing a sequence of values $\{x_1, \dots, x_T\}$. The model is able to generalize across them thanks to the parameter sharing (right image). Always in the right image, we can see that we have an input and an output. The information flows from input to output but in $h^{<t>}$ there is a recurrent edge which takes the output of the hidden layer and sends it back to its input. So the hidden layer is a combination of the input and the previous state of hidden data. So an output y depends on the direct input + all the previous inputs in the hidden layer.



We say that this was a single layer RNN, but now we can't use this sort of architecture (with a loop) since it would be bad for the back-propagation algohirthm. That's why we use h^{t-1} linked to h^t, linked to h^{t+1}, etc.



As for the Weight Matrices, we have the following scheme:



W_{hx} which represents the grey arrow from x to h. Then we have W_{yh} which represents the arrow from h and y and then we have W_{hh} which are the arrows going from h to h. Clearly now, at time-step t, the output does not only depend on the input, but depends also on h^t which is what the network has seen up to that point.

1.1 Vanilla RNN

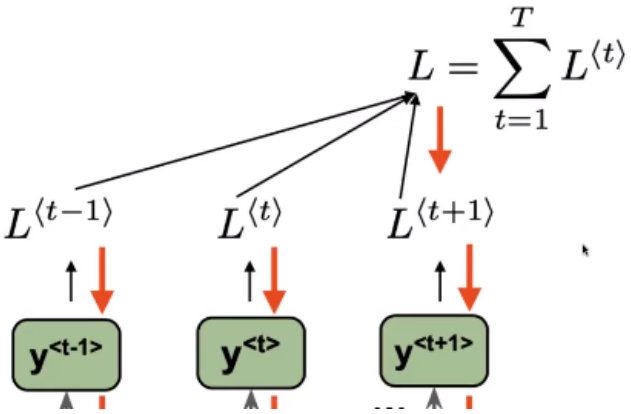
$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

$$h_t = g_\theta(h_{t-1}, x_t)$$
$$\downarrow$$
$$h_t = \tanh(W_{hh}\, h_{t-1} + W_{xh} x_t)$$

$$o_t = softmax(W_{ho} h_t)$$

This is the default RNN that comes with PyTorch. A possible instantiation of a hidden layer is computed as follows (left image). Very simply, we compute hidden state at time t. It is equal to the Hyperbolic Tangent of W_{hh} applied to the previous hidden state, as we showed before. Then we add the weights of our input at time step t.

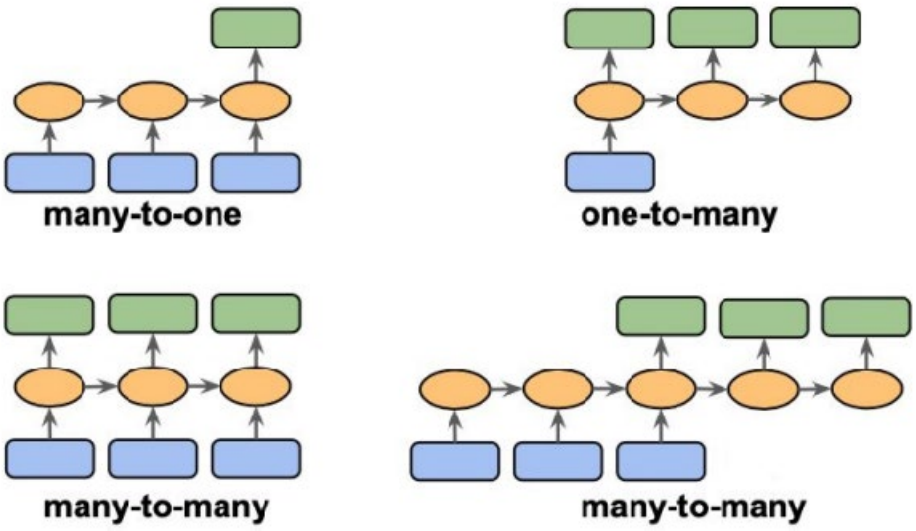
As for backpropagation, we call it backpropagation through time (right image), since we have a loss for each output at different time steps and we compute the derivative with respect to the sum of these losses.



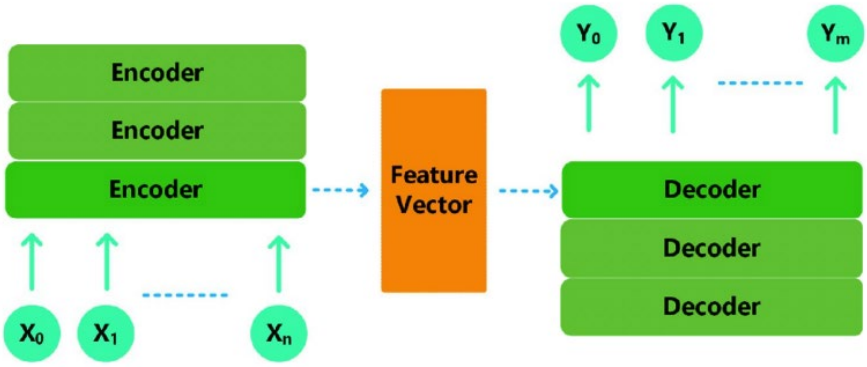
2. Sequence-to-Sequence (Seq2Seq)

Seq2Seq are a category of models where we want to go from a sequence of objects to another sequence of objects (input/output can be also different).
Example of translation: I am Italian -> Io sono Italiano.

We have different kinds of Seq2Seq models, as we can see here:



In general, we talk about the Encoder-Decoder paradigm, where we have an input that we encode into a feature vector and we have a decoder which decodes the output starting from the feature vector.

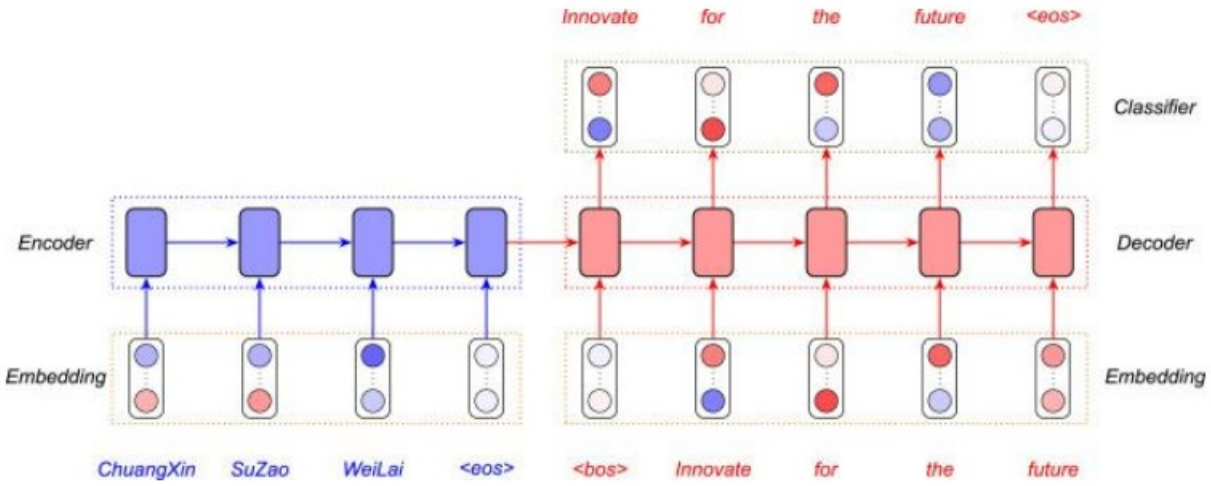


The DL version (upper image), simply stacks multiple encoder-decoder parts, where the feature vector is a linear combination of the Encoders.

3. Autoregressive models

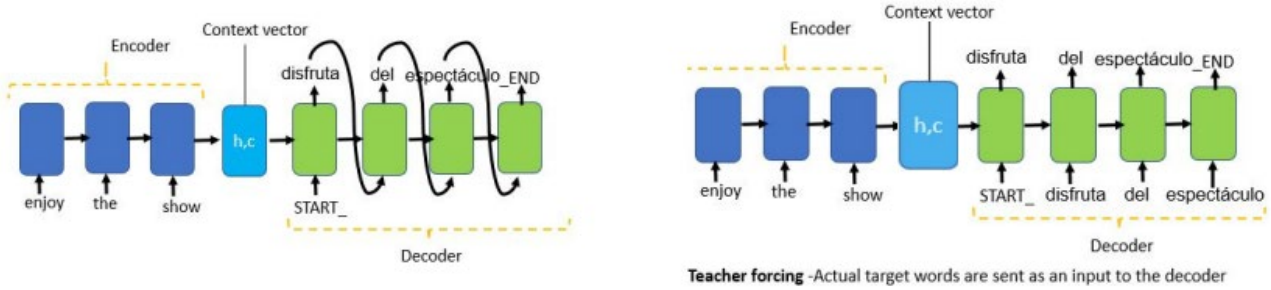
A very important Seq2Seq models are represented by the autoregressive models. These models take both the hidden state up to time t, but also the generated output (y) at $t-1$.

Simply, we can look at this through the Encoder-Decoder part on the image on top where if we are at $t = 2$, the model takes the hidden state at time step 2 plus the output y_1. In terms of language translation, this is really helpful because as we output translations, we also take into account the latest output + all the previous ones.



3.1 Teaching Forcing

In Teaching forcing, instead of inputting always the previous output (during the encoding), we discard the previous output and we feed the stage with the actual ground truth with probability p.



Teacher forcing -Actual target words are sent as an input to the decoder

3.2 A brief comment on LLMs

In the field of Natural Language Processing (NLP), modern large language models aim to understand the meaning of a word by examining the words that surround it in a given context. One such simple language model is Word2Vec, which was pioneering in showing how to create word representations using neural networks. This can be accomplished through two distinct tasks: Continuous Bag of Words (CBOW) and Skip-Gram.

In CBOW, we are provided with the context and aim to predict the missing word, or we can reverse this process with Skip-Gram. A notable example of Word2Vec's capabilities is that it allows us to capture semantic relationships, like (man + uncle) - aunt = woman. The fundamental idea is to transition from an N-dimensional vector, representing tokens or words using one-hot encoding, to a much smaller D-dimensional vector, where D is significantly smaller than N. Typically, N might be on the order of 10^7, while D is around 10^2.

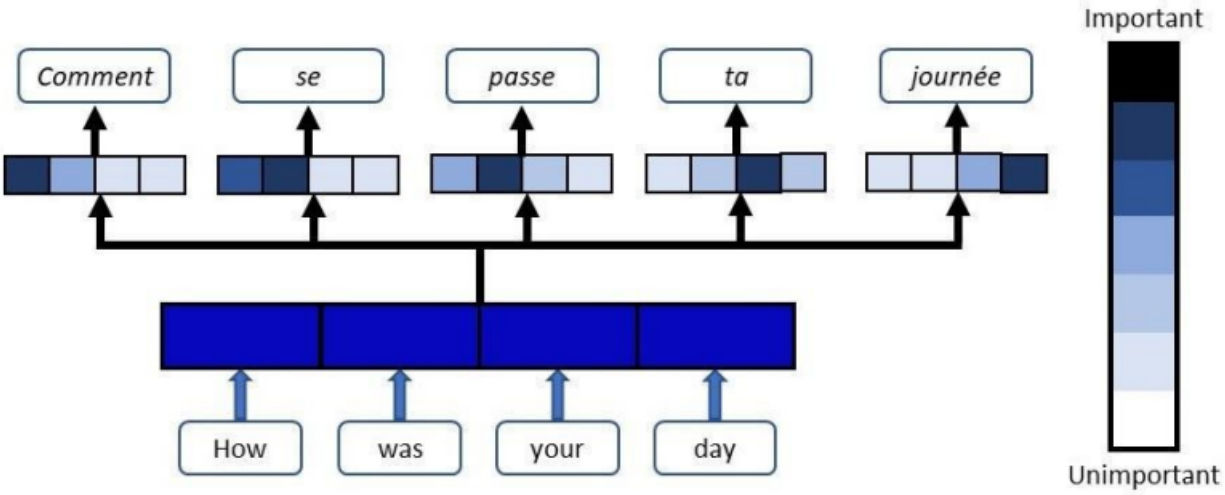
However, a significant limitation of Word2Vec is that it assigns a single vector to each word, regardless of the context in which that word appears. This poses a challenge for capturing the true meaning of a word, as meaning can vary based on context. To address this issue, ElMo introduced a method for generating word embeddings that depend on the context. It utilizes Word2Vec representations and passes them through an LSTM (Long Short-Term Memory) architecture to create a vector representation for each specific meaning of a word based on its context.
Forward and backward language models are like two language detectives working together to understand a word's context.

The forward model reads the words that come before the target word, while the backward model looks at the words that come after it. They each create clues about what the word means based on their observations.

When you add the information from both models together, it's like combining the clues from both detectives. This sum of clues gives you a special code (embedding) that represents the word's meaning in its specific context. It's like having a complete picture of the word's role in the sentence.

4. Attention mechanism

The Attention mechanism can be appreciated through this scheme down here:



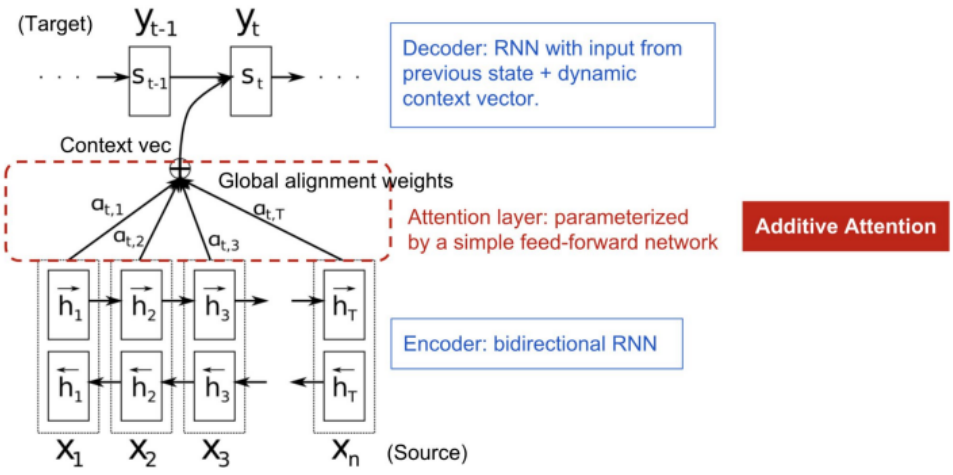
Basically, for each output, we want to associate a vector which has the same dimensions as the input vector (We call it attention vector and it has 4 elements).

For each element of the attention vector (through different output words) we have a score which measures the importance of the corresponding word for the output.

In this case, for each output we have a classifier (so we have 5 different classifiers) which measures the probability that the first input is important, the second input is important etc. for that specific output token.

Let's remember that each output has a vector associated with it. Each output is α_t and it's the t-attention vector and each dimension come from each hidden state of the Encoder (RNN).

Very primitive form of attention.



Looking deeply in the way Attention works, we want to understand how each classifier gives a certain score to a certain word (in the example above). Let's have a look at a general understanding of this concept:

We have a sequence $x = [x_1, x_2, \dots, x_n]$ and outputs $y = [y_1, y_2, \dots, y_n]$ and an encoder which is an RNN with a Forward Hidden State $h_{\rightarrow}$. The decoder then, has the following hidden states:

$$s_t = f(s_{t-1}, y_{t-1}, c)$$

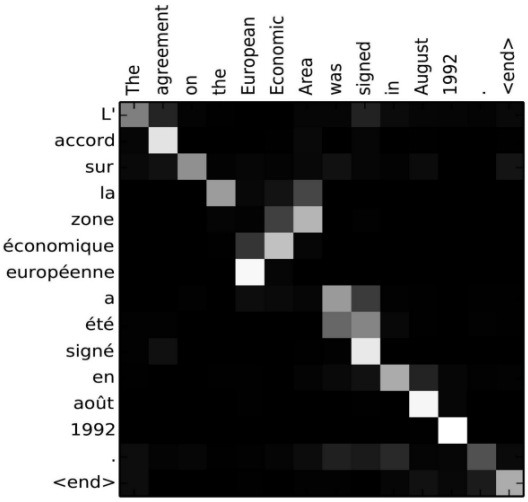
s_{t-1} = Previous hidden state | y_{t-1} = Previous output

There is c, which is the context at time t computed as a sum of hidden states of the input sequence weighted by the alignment scores (**attention**). The alignment scores are computed as a multi-class classification (SoftMax) of the previous state and each corresponding hidden state.

Now the alignment scores are learnable parameters, therefore the scores are learnt through the back-propagation mechanism.

In this case (on the right) we have a matrix of alignment scores which shows the correlation between source and target words.

If we take “signed” as target word, the model will look more into ‘signé, été, a’, which are more or less all past-tense verbs.



5. Self-Attention

Self-Attention is an attention mechanism which relates different position of a single sequence in order to compute a representation of the same sequence. Namely, it’s a powerful mechanism that focuses on the words that are more relevant in order to explain the semantics of the whole sentence. So, it gives an importance to each word (importance to important tokens).

6. Transformers

Thanks to the paper “Attention is all you need”, it was possible to do seq2seq modeling without having to use Recurrent Networks units. Namely, we use the encoding of the input to decode the output. This leads us to the Transformer architecture.

Commenting it, the input goes through an Embedding layer (Which outputs an n-matrix where in each row there is a word embedding – like Word2Vec etc.) After that we add a positional encoder which tells us the position of each token in the whole sequence. Then we have N layers composed as follows: Multi-Head Attention, Normalization, Residual connection, Feed Forward (Projection) and Normalization.

Something really important to say is that our input is not threated directly as we start our encoding procedure, but will enter after a “Masked Multi-Head Attention” layer.

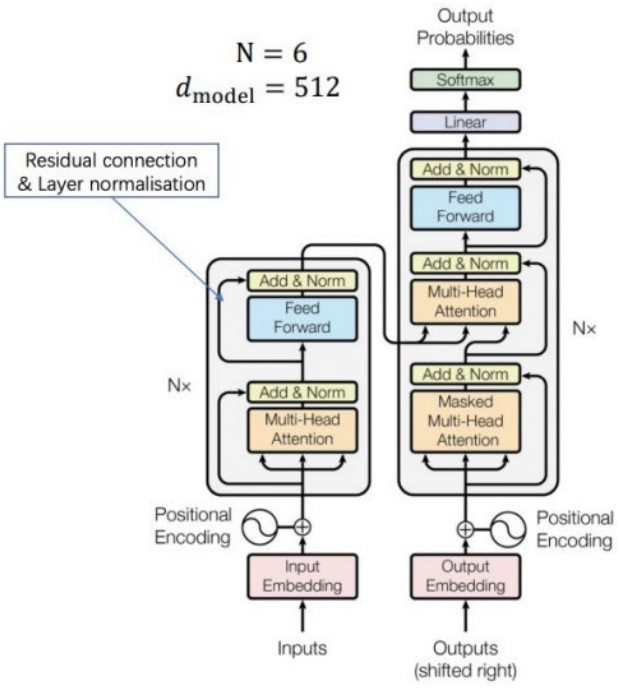


Figure 1: The Transformer - model architecture.

Figure in (Vaswani *et al.*, 2017)

If we take into consideration a translation task, we use some sort of autoregressive procedure, meaning that, during translation, we want to exploit only the knowledge that we have acquired up to a certain point.

So, if we are translating in Italian "Attention is all you need", we don't want to use "is all you need" to translate "Attention", because it could give away too much information, becoming expensive. So, we mask "is all you need" and then that is the input that is combined with the encoder.

6.1 Query, Key, Value

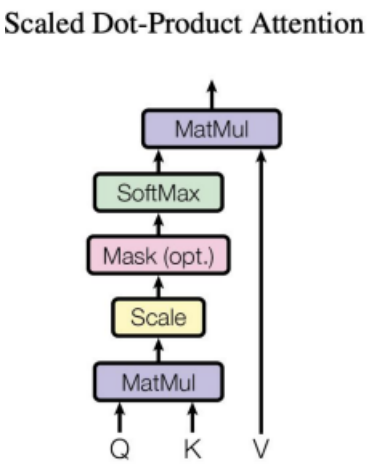
The self-attention layer has three main components: Query Q, Value V, and Key K.

Let's consider an example where we have 3 objects and we want to classify them through a multi-class classification. What attention does is that in order to classify one single object (a *Key*), it "seeks" help from some sort of "memory" through a *Query*. Then, the memory will return a specific *Value*. This value then is used to perform classification tasks.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{n}})\mathbf{V}$$

In the Transformer architecture, the encoded representation of the input is seen as a set of key-value pairs (K, V) both of dimension n (the input sequence length). So, we have Q which is the vector of tokens and K which is the vector of keys and V which is the vector of values.

Essentially, we compute the vector product (divided by a normalization factor $\sqrt{n}$) in order to penalize longer sequences and the we pass it through the SoftMax.



The SoftMax will tell us what are the most important values between the retrieved ones and in fact it's multiplied by V.

Overall, in general, we want to estimate the importance of some values that we have through a query in the memory (using Keys K to create that memory).

And so, we compute this vector product, which again, we normalize and later on we compute the SoftMax which will return a vector of probabilities which will be multiplied by V.

This is called "**Scaled Dot-Product Attention**".

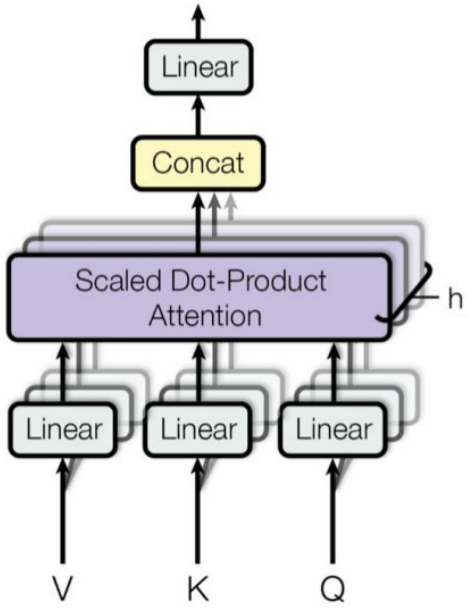
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}	$x_{11} * v_1$	$\sum = z_1$
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}	$x_{13} * v_3$	

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action		k_1	v_1	$q_2 . k_1$	$q_2 . k_1 / 8$	x_{21}	$x_{21} * v_1$	
gets	q_2	k_2	v_2	$q_2 . k_2$	$q_2 . k_2 / 8$	x_{22}	$x_{22} * v_2$	z_2
results		k_3	v_3	$q_2 . k_3$	$q_2 . k_3 / 8$	x_{23}	$x_{23} * v_3$	

As we can see, we compute Z_1, Z_2 which are respectively the self-attention vector for the first word of the input sequence "Action gets results".

N.B. (score / 8) should be (score / **3**), since $n = 3$.

7. Multi-head Self-Attention



Rather than only computing the attention once, the multi-head attention mechanism computes the attention multiple times in parallel. Essentially, you repeat the input (creating H copies of V, K, Q and H copies of the architecture) and then we compute them in parallel (facilitating training for larger models) and in the end, we just concatenate them.

This is nice because it allows each "head" to capture different aspects of the input instead of having a single "view" of the input sequence.

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\text{head}_1; \ldots; \text{head}_h]\mathbf{W}^O$$
$$\text{where head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V)$$

where $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$, and $\mathbf{W}^O$ are parameter matrices to be learned.

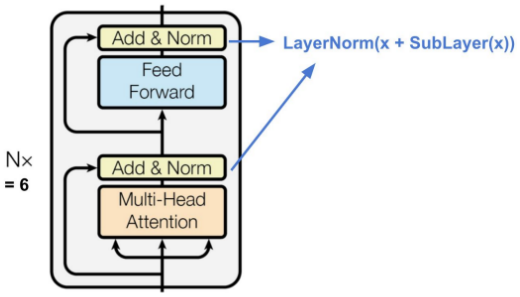
8. Transformer Architecture

8.1 Encoder

The Transformer Encoder can be easily explicated as follows:

The encoder generates an attention-based representation with capability to locate a specific piece of information from a potentially infinitely-large context.

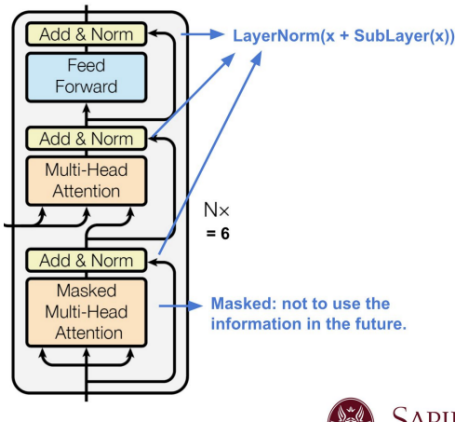
- A stack of N=6 identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.
- Each sub-layer adopts a residual connection and a layer normalization.
- All the sub-layers output data of the same dimension d_{model}=512.



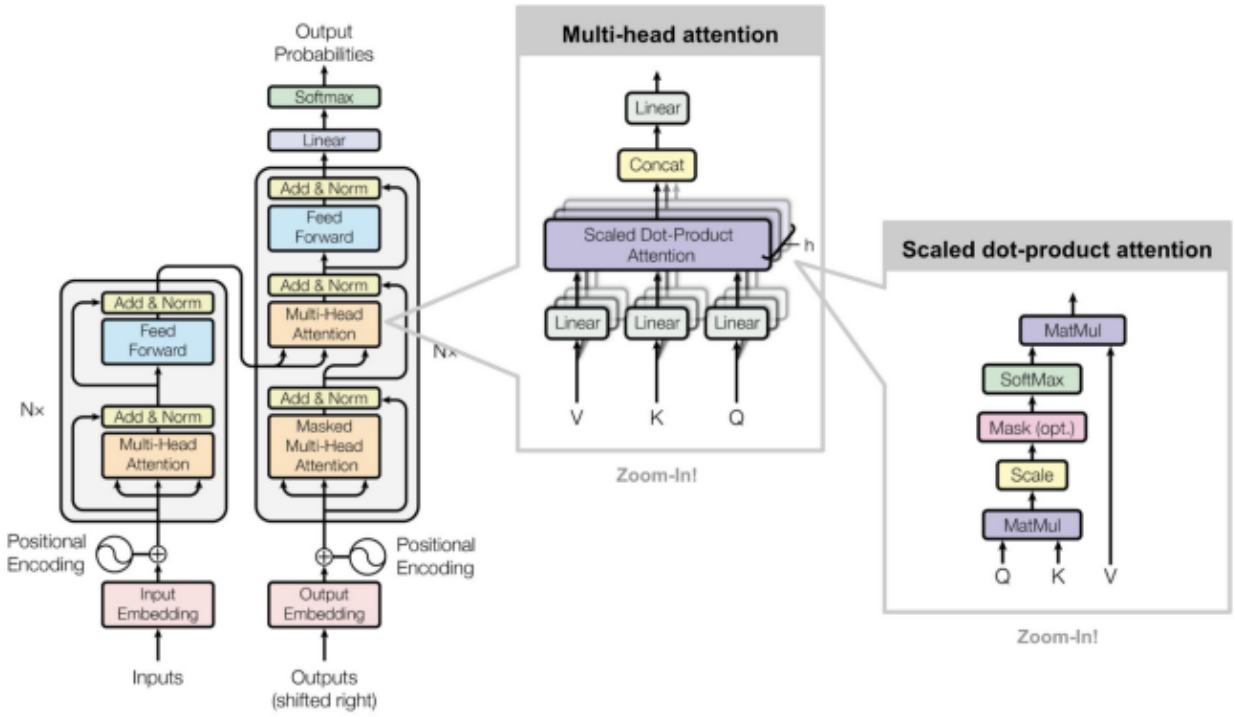
8.2 Decoder

The decoder is able to retrieval from the encoded representation.

- A stack of N = 6 identical layers
- Each layer has two sub-layers of multi-head attention mechanisms and one sub-layer of fully-connected feed-forward network.
- Similar to the encoder, each sub-layer adopts a residual connection and a layer normalization.
- The first multi-head attention sub-layer is modified to prevent positions from attending to subsequent positions, as we don't want to look into the future of the target sequence when predicting the current position.



We have to remember to mask our multi-head-attention. A mask is a binary vector (0,1) which sets to some neurons to 0 (as we do in Dropout).

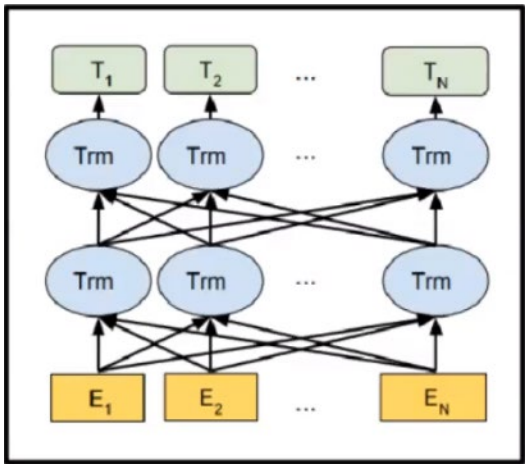


One thing that we might have missed is the positional encoding procedure:

The most straightforward one is to have an embedding for each position. Supposing that the maximum length is N, then we could have an embedding for position 1, an embedding for position 2 and so on and so forth. At the beginning we randomly initialize it and through backprop we make the model learn the best values.

In order to train this model, you use Cross-Entropy since it's basically a classification task.

9. BERT (Bidirectional Encoder Representations from Transformers)



BERT is a multi-layer bidirectional Transformer Encoder where we have m layers of Self-Attention and then we have the output T_1, T_2, T_3 etc. which are the scores (Z_1, Z_2, etc) we had before.

BERT is a language model and it's only the encoder of the Transformer, so we are not using the decoder at all. We are just encoding the sentence.

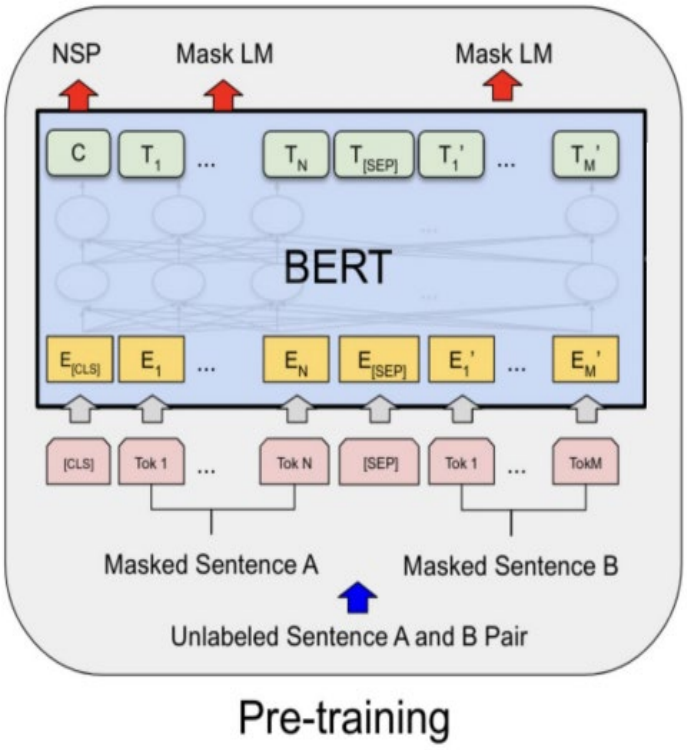
In the BERT paper, they adopted 3 proxy tasks which are:

• 80% of the time: Replace a word with the [MASK] token. Essentially the mask is used to mask some of the input. (Example: My dog is ~~hairy~~ [MASK]). This word is unknown to the Encoder, so that when it reads the input and generates the output, the correct word should have higher likelihood and the target class should be the ID of the word "hairy".

- 10% of the time: Replace with a random word (making it more robust to mistakes)
- 10% of the time: Keep the word unchanged (making the model understand the meaning of the sentence).

We do this to make sure that the representation of “My dog is” is so good that we can directly recover the missing word, which could also have multiple variations. Meaning that we can have “My dog is [happy], [running], [barking]” each with a high probability (like: [happy, 87%], [running, 89%], [barking, 76%]). In this case the model will put “running” as the missing word.

9.1 Next Sequence Prediction



Pre-training

Let’s take this example:

Input = [CLS] the man went to [MASK] store [SEP]

He bought a gallon [MASK] milk

Label = IsNext

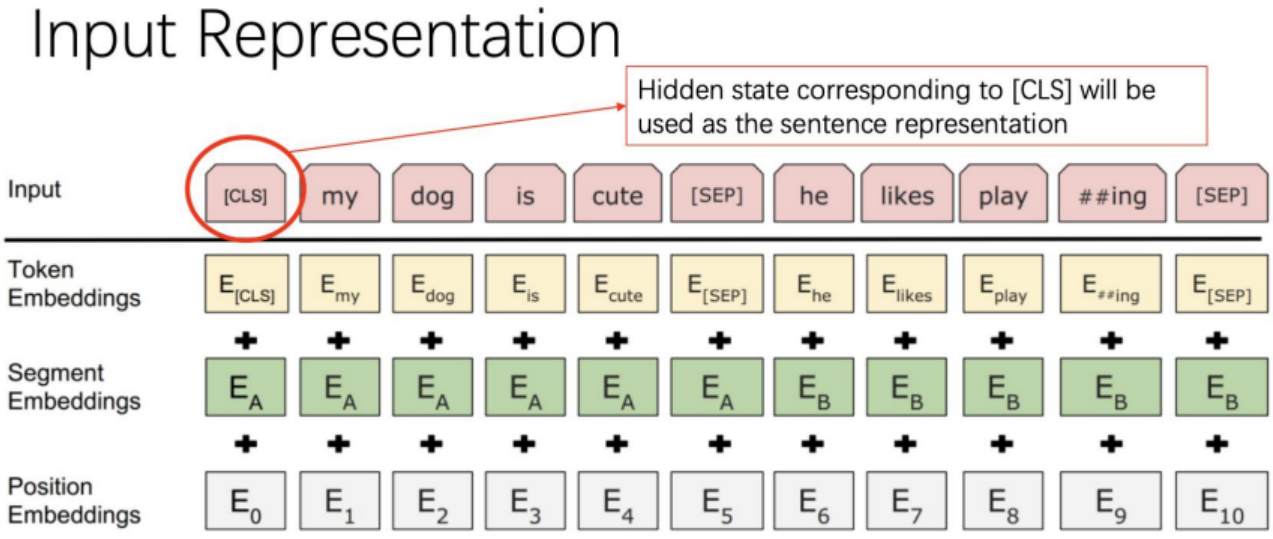
In this example right here, we see that we have a task which involves replacing [MASK] with the correct token, but we also have [CLS] (Class) and [SEP] (Separator) tokens.

We use the CLS token to build a classifier which will tell me whether the sequence of tokens before [SEP] can precede (in a paragraph) the sequence of tokens after [SEP] itself that and before the last one. If yes, then label = IsNext, otherwise label = NotNext

Input = [CLS] the man went to [MASK] store [SEP]

Penguin [MASK] are flight less birds

Label = NotNext



One important thing to specify here is that when we deal with multiple sentences, we insert a Segment Embeddings vector which specifies whether tokens belong to a certain sentence A or B, which will help the model being aware the position of the tokens. We can see this in the “Segment Embeddings” part.

9.2 Fine-Tuning

As we do with Transfer Learning between CNN architectures, we can do the same for these models here by fine-tuning. An example could be Question-Answer Pair where we give the question, I give a potential answer and then the output would be, for each token in the answer, 0 or 1 if that token is part of the answer to the question. This becomes a classification task at the level of tokens.

In NER (Named Entity Recognition) we do basically the same. We input the sequence and we want to tag each token with a possible entity.

Example: President Obama flew to San Francisco.

In this case "President Obama" will be tagged with the ID of the entity regarding Obama, while "San Francisco" will be tagged with the ID of the city.

In some cases, we want to classify the whole sentence (and we do this through using the [CLS] token as features for the classifier).

10. Visual Transformers

In ViT, we adapt the idea of Transformers to images.

At first, it could be difficult if we try to use the transformer directly on pixels instead of wordpieces. This is due to the fact that if we then compute the n squared matrix, it would be very very expensive (256x256 image -> 62.500 pixel -> 3.906.250.000 calculations).

In order to fix this, we use patches and linear embed them, so that we'll get our embeddings and then we can easily apply our transformer architecture.