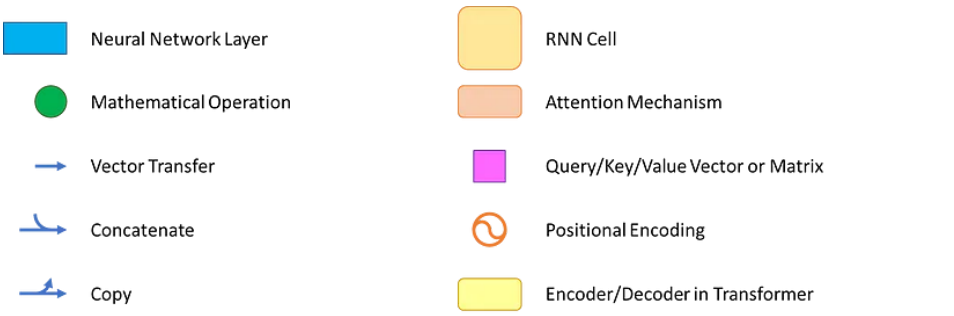
Natural Language Processing (NLP) is a challenging problem in deep learning since computers don't understand what to do with raw words. To use computer power, we need to convert words to vectors before feeding them into a model. The resulting vectors are called **word embeddings**.

Those embeddings can be used to solve the desired task, such as sentiment classification, text generation, name entity recognition, or machine translation. They are processed in a clever way such that the performance of the model for some tasks becomes on par with that of human capabilities.

So, how to process word embeddings, I hear you ask? How do you build a natural model for this kind of data?

Let's first try to get comfortable with the notation we'll be using.



The notation we'll be using in this story | Image by author

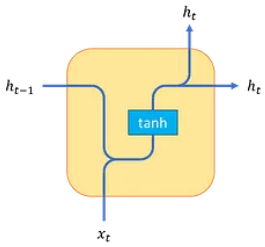
Recurrent Neural Networks (RNN)

You don't start thinking from scratch for each incoming word when you read this sentence. You maintain the information of the previous words to understand the word you're currently looking at. Based on this behavior, Recurrent Neural Network (RNN) was born into existence.

In this part of the story, we will focus on RNN cells and their improvements. Later, we will see how the cells are arranged together.

Vanilla RNN

The vanilla RNN consists of repeated cells, each receiving an input embedding x_t sequentially and remembering the past sequence through the hidden state h_{t-1}. The hidden state is updated to h_t and is forwarded to the next cell, or — depending on the task — can be used to output a prediction. The diagram below shows the inner workings of an RNN cell.



$$h_t = \tanh(W \cdot [h_{t-1}, x_t] + b)$$

Vanilla RNN cell | Image by author

Advantages:

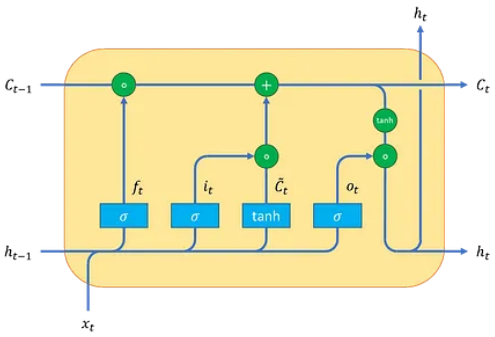
- Account for order and previous inputs in a meaningful way.

Disadvantages:

- Each step's prediction depends on the previous predictions, so it's hard to parallelize RNN operations.
- Processing long sequences can yield exploding or vanishing gradients.

Long Short-term Memory (LSTM)

One way to mitigate the exploding or vanishing gradients issue is to use gated RNNs, which are devised to retain information selectively and are capable of learning long-term dependencies. There are two popular varieties of gated RNNs: Long Short-term Memory (LSTM) and Gated Recurrent Unit (GRU).



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$C_t = f_t \circ C_{t-1} + i_t \circ \tilde{C}_t$$
$$h_t = o_t \circ \tanh(C_t)$$

LSTM cell | Image by author

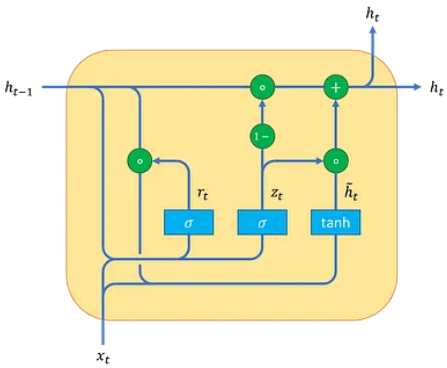
To avoid the long-term dependency problem, LSTM is equipped with a cell state C_t which is kind of like a freeway, so it's very easy for information to simply flow through it unchanged.

To selectively retain information, LSTM also has three gates:

1. forget gate → looks at h_{t-1} and x_t, and outputs a vector f_t consisting of numbers between 0 and 1 that tells what information we're going to throw away from the cell state C_{t-1}.
2. input gate → similar to forget gate, but this time the output i_t is used to decide what new information we're going to store in the cell state based on a dummy cell state $\acute{c}_t$.
3. output gate → similar to forget gate, but the output o_t is used to filter the updated cell state C_t to become a new hidden state h_t.

Gated Recurrent Unit (GRU)

LSTM is pretty complex. GRU offers similar performance to LSTM with less complexity (fewer weights). It merges the cell state and the hidden state. It also combines the forget and input gates into a single "update gate".



$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$
$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$
$$\tilde{h}_t = \tanh(W_h \cdot [r_t \circ h_{t-1}, x_t] + b_h)$$
$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t$$

GRU cell | Image by author

To explain further, there are two gates in GRU:

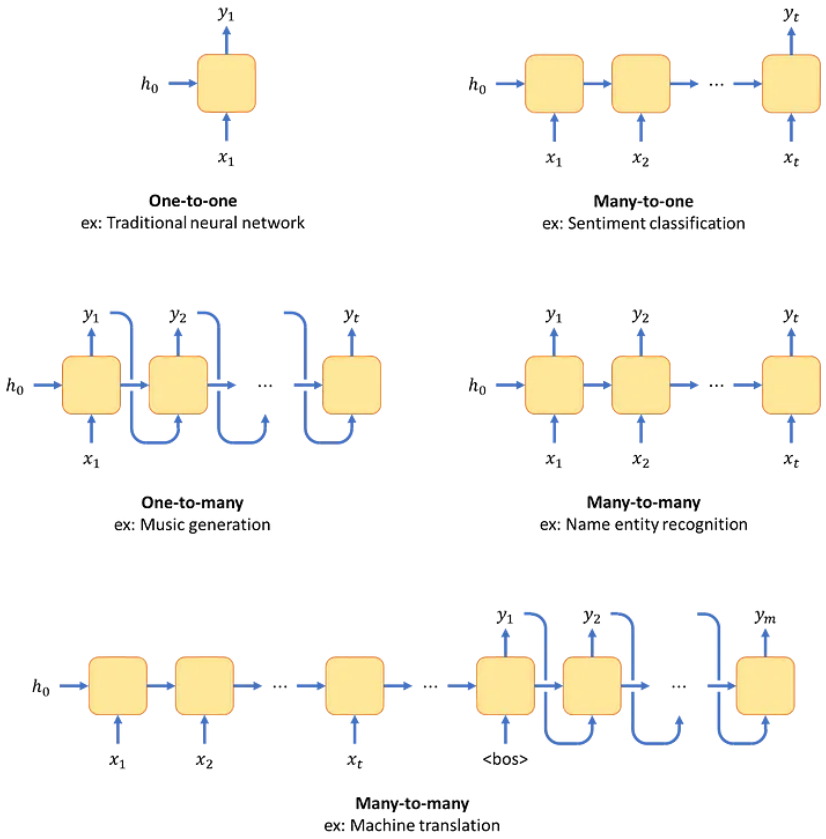
1. reset gate → looks at h_{t-1} and x_t, and outputs a vector r_t consisting of numbers between 0 and 1 that decides how much of the past information h_{t-1} is needed to neglect.
2. update gate → determines what information we're going to store in the new hidden state h_t or throw away based on r_t.

RNN Architectures

Now you've understood how an RNN cell works, you can chain copies of it sequentially as follows. Here, y_t is the output probability of predictions and is expressed as

$$y_t = g(Wh_t + b),$$

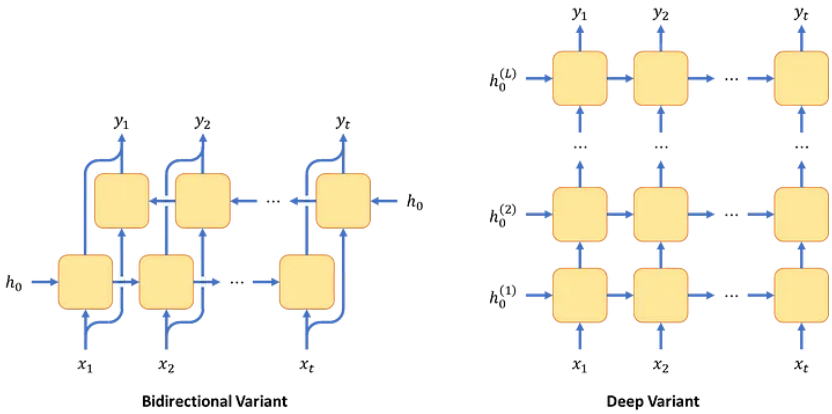
for some weight W, bias b, and activation function g. As you can see, the architecture and the activation function depend on the task.



RNN architectures depending on the task | Image by author

In particular, the last example of machine translation above is what we call a **seq2seq** model, where the input sequence $x_1, x_2, ..., x_t$ is translated to the output sequence $y_1, y_2, ..., y_m$. The group of the first t RNNs is called the **encoder** and the group of the last m RNNs is called the **decoder**.

RNN cells can also be flipped or vertically stacked as follows. Such architectures are usually composed of just two or three layers. This may improve the model performance at the expense of heavier calculations.



RNN variants, such architectures are usually composed of just two or three layers | Image by author

Attention

Seq2seq was constrained to using the representation at the very end of the encoder to be interpreted by the decoder. The decoder sees only one representation of the input sequence from the last encoder RNN cell. However, different parts of the input sequence can be more useful at each generation step of the output sequence. This is the idea of attention.

Advantages:

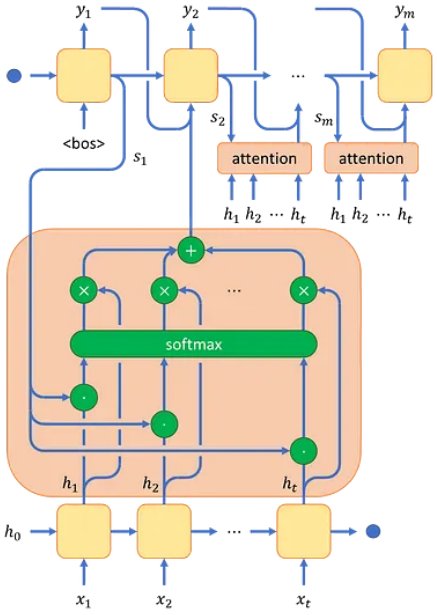
- Account for the appropriate encoded representations regardless of the position of input tokens.

Disadvantages:

- Another computing step involves learning weights.

Seq2seq with Attention

Each decoder token will look at every encoder token and decides which tokens need more attention through the means of their hidden states.



$$H = [h_1 \quad h_2 \quad \cdots \quad h_t]^T$$

$$\text{Attention}(s_i, H) = \text{softmax}\left(\frac{s_i H^T}{\sqrt{d_h}}\right) \cdot H$$

Working diagram of seq2seq with attention | Image by author

There are three steps to incorporate attention in seq2seq:

1. A scalar **attention score** is calculated from a pair of decoder and encoder hidden states (s_i, h_j) which represents the "relevance" of encoder token j for decoder token i.
2. All attention scores are passed through softmax to yield **attention weights** which form a probability distribution of the relevance of decoder and encoder token pairs.
3. The **weighted sum** of encoder hidden states with attention weights is calculated and fed into the next decoder cell.

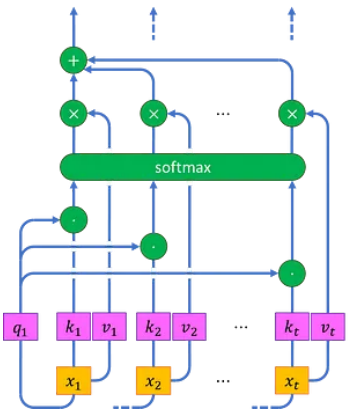
In our case, the score function is the scaled dot product

$$\text{score}(s_i, h_j) = \frac{s_i^T h_j}{\sqrt{d_h}},$$

where d_h is the dimension of h_j (and s_i).

Self-attention

Let's discard seq2seq entirely and focus on attention only. A popular kind of attention is self-attention where instead of finding the relevance from the decoder to encoder tokens, it finds relevance from each token in a set of tokens to all other tokens in the same set.



$$X = [x_1 \quad x_2 \quad \cdots \quad x_t]^T$$

$$Q = [q_1 \quad q_2 \quad \cdots \quad q_t]^T = XW_Q$$

$$K = [k_1 \quad k_2 \quad \cdots \quad k_t]^T = XW_K$$

$$V = [v_1 \quad v_2 \quad \cdots \quad v_t]^T = XW_V$$

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

Working diagram of self-attention | Image by author

Self-attention creates a weighted representation that is based on the similarity between pairs of input tokens using the attention function, which delivers rich representations of the input sequence that are aware of the relationships between its elements.

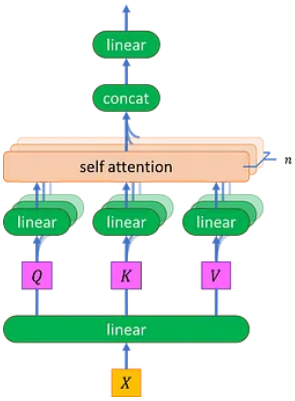
There are three main differences between vanilla attention and self-attention:

1. Since there is no decoder token in self-attention, a "query" vector q_i — that relates linearly to input embedding x_i — takes its place.
2. The attention score is calculated from the pair (q_i, k_j), where k_j is a "key" vector with the same dimension as q_i and also relates linearly to x_j.
3. Instead of multiplying k_j again with attention weights as in vanilla attention, self-attention multiplies a new "value" vector v_j with attention weights. Note that v_j may have a different dimension than k_j, and again, relates linearly to x_j.

Multi-head Attention

Attention can be run several times in parallel to produce what we call multi-head attention. The independent attention outputs are then concatenated and linearly transformed into the expected dimension.

A motivation behind multiple attention heads is they allow for attending to parts of the input sequence differently (e.g. longer-term dependencies versus shorter-term dependencies) and hence it may improve the performance of single attention.



$$X = [x_1 \quad x_2 \quad \cdots \quad x_t]^T$$

$$Q = XW_Q$$

$$K = XW_K$$

$$V = XW_V$$

$$\text{head}_i = \text{Attention}\left(QW_Q^{(i)}, KW_K^{(i)}, VW_V^{(i)}\right)$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_n)W_O$$

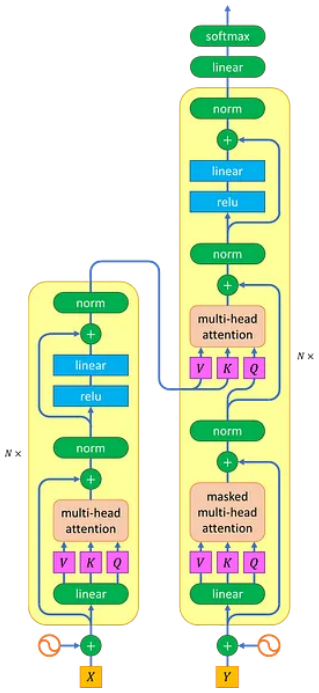
Working diagram of multi-head attention with n attention mechanisms | Image by author

Since things will get crazier in the next section of the story, in the diagram above, we vectorize everything so that the input and output of the multi-head attention are represented by a matrix with a single arrow.

The self-attention may also be swapped out with any kind of attention or implemented with any modifications as long as the dimensions of its variables are appropriate.

Transformer

Given input embeddings X and output embeddings Y, generally speaking, a transformer is built using N encoders stacked after another linked to N decoders also stacked after another. No recurrence or convolution, *attention is all you need* in each encoder and decoder.



$$P_{i,2j} = \sin\left(i/10000^{2j/d_x}\right)$$
$$P_{i,2j+1} = \cos\left(i/10000^{2j/d_x}\right)$$
$$X = [x_1 \quad x_2 \quad \cdots \quad x_t]^T + [P_{ij}]$$
$$Y = [<\text{bos}> \quad y_1 \quad \cdots \quad y_m]^T + [P_{ij}]$$

$$\text{For } r = 1,2,\ldots,N$$
$$X := \text{LayerNorm}\left(X + \text{MultiHead}(XW_Q, XW_K, XW_V)\right)$$
$$X := \text{LayerNorm}\left(X + \max\left\{0, XW_1^{(r)} + b_1^{(r)}\right\}W_2^{(r)} + b_2^{(r)}\right)$$

$$\text{For } r = 1,2,\ldots,N$$
$$Y := \text{LayerNorm}\left(Y + \text{MaskedMultiHead}(YW_Q, YW_K, YW_V)\right)$$
$$Y := \text{LayerNorm}\left(Y + \text{MultiHead}(YW_Q, XW_K, XW_V)\right)$$
$$Y := \text{LayerNorm}\left(Y + \max\left\{0, YW_3^{(r)} + b_3^{(r)}\right\}W_4^{(r)} + b_4^{(r)}\right)$$

$$\text{proba} = \text{softmax}(YW_O)$$

Working diagram of transformer with N encoders and decoders | Image by author

Advantages:

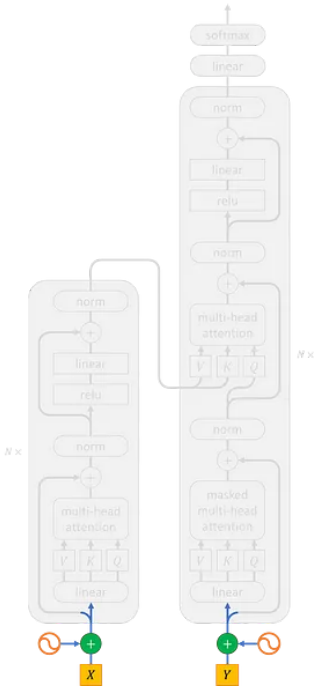
- better representation of input tokens where the token representation is based on specific neighboring tokens using self-attention.
- attends (in parallel) to all input tokens, as opposed to being limited by memory issues from sequential processing (RNNs).

Disadvantages:

- computationally intensive.
- requires large amounts of data (mitigated using pre-trained models).

Let's dive in to see how the transformer works!

Step 1. Adding Positional Encoding to Word Embeddings



Adding positional encoding to word embeddings | Image by author

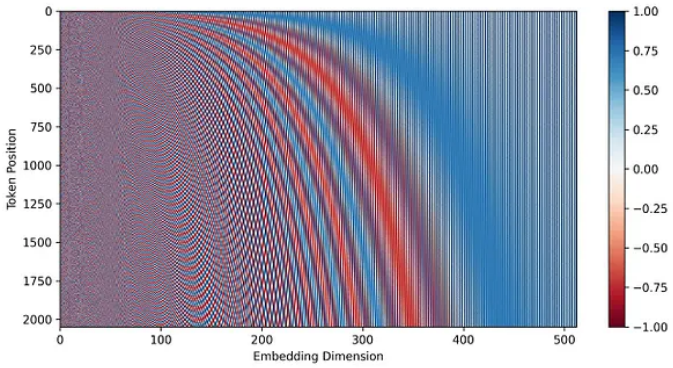
Since the transformer contains no recurrence and no convolution, for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence.

Therefore, we have to let the model know the positions of the tokens explicitly by the means of "positional encoding":

$$P_{i,2j} = \sin\left(i/10000^{2j/d_x}\right)$$
$$P_{i,2j+1} = \cos\left(i/10000^{2j/d_x}\right),$$

where i is the position of the token (token#0, token#1, and so on), j is the column number of the encoding, and d_x is the dimension of the encoding (which is the same as the dimension of the input embeddings X).

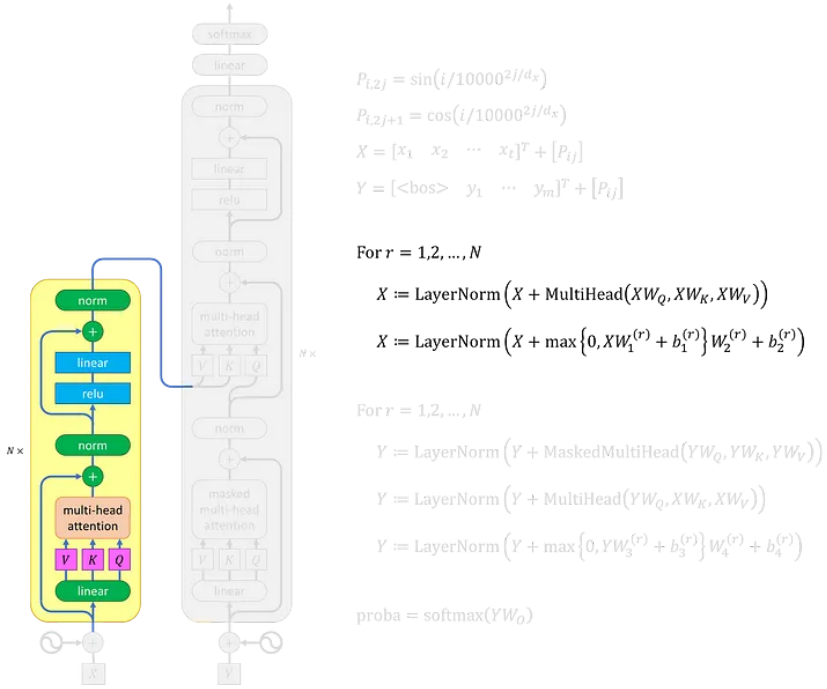
For the first 2048 tokens with encoding dimension 512, you can visualize the positional encoding matrix P as follows.



Positional encoding for the first 2048 tokens with encoding dimension 512 | Image by author

As in the seq2seq model, the output embeddings Y is shifted right and the first token is the "beginning of sentence" <bos>. Positional encoding is then added to X and Y before being fed to the first encoder and decoder, respectively.

Step 2. Encoder: Multi-head Attention and Feed Forward



Encoders in transformer | Image by author

An encoder consists of two blocks:

1. The input embeddings X — after being added positional encoding P — is fed into **multi-head attention**. Residual connection is employed to the multi-head attention, which means we add up the word embeddings (or the output of the previous encoder) to the output of the attention. The result is

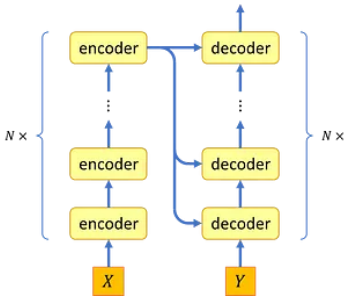
then normalized.

2. After being normalized, the information is passed through a **feed-forward** neural network, which consists of two layers with ReLU and linear activation function, respectively. Again, we employ residual connection and normalization on this block.

To know more about residual connection and layer normalization, you can head into this story:

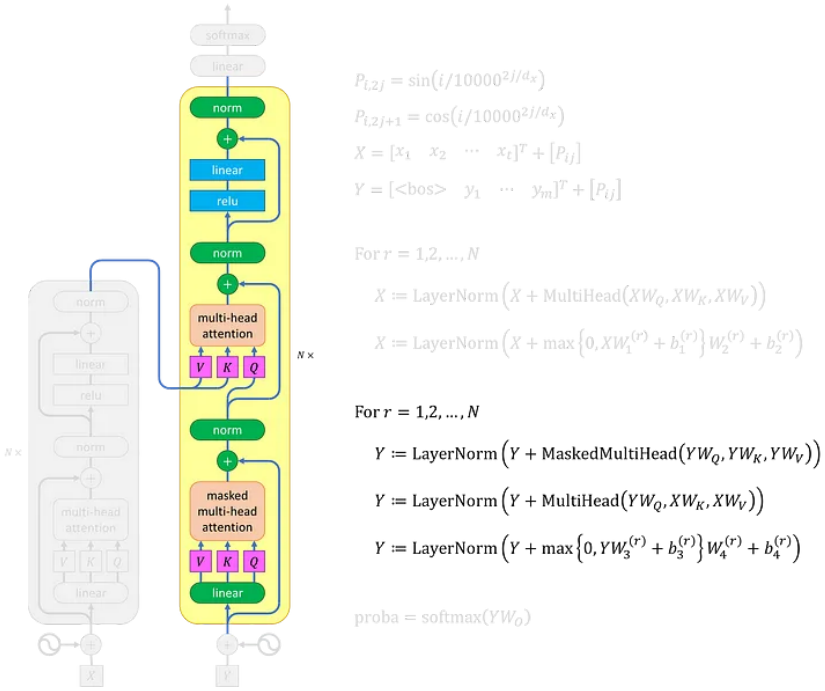
5 Popular CNN Architectures Clearly Explained and Visualized
And why the heck does Inception look like a trident?!
towardsdatascience.com

That is the end of the calculation inside an encoder. The output can be fed to another encoder or a specific part of every decoder after the last encoder.



Transformer encoders and decoders | Image by author

Step 3. Decoder: (Masked) Multi-head Attention and Feed Forward



Decoders in transformer | Image by author

A decoder consists of three blocks:

1. The output embeddings Y — after being shifted right and added positional encoding P — is fed into **multi-head attention**. In the attention, we mask out (setting to $-\infty$) all values in the softmax input which correspond to connections to subsequent positions.

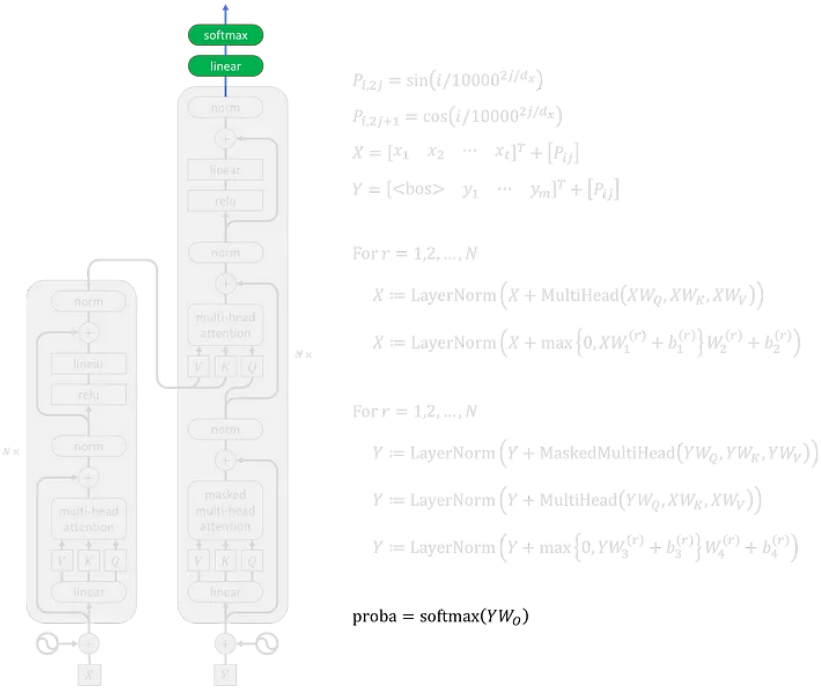
2. The information is then passed through another **multi-head attention** — now without masking — as the query vector. The key and value vectors come from the output of the last encoder. This allows every position in the decoder to attend to all positions in the input sequence.

3. The result is passed through a **feed-forward** neural network with activation functions just like the encoders have.

Note that all decoder blocks also employ residual connection and layer normalization. The masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position *i* can depend only on the known outputs at positions less than *i*.

That is the end of the calculation inside a decoder. The output can be fed to another decoder or a classifier after the last decoder.

Step 4. Classifier



Classifier in transformer | Image by author

This is the last step and it's pretty simple. We use learned linear transformation and softmax to convert the decoder output to predicted next-token probabilities.

The transformer has had great success in NLP. Many pre-trained models such as GPT-2, GPT-3, BERT, XLNet, and RoBERTa demonstrate the ability of transformers to perform a wide variety of NLP-related tasks such as machine translation, document summarization, document generation, named entity recognition, and video understanding.

Transformers have also been applied to image processing with competitive results compared to Convolutional Neural Networks.

Wrapping Up

Photo by Jeremy Thomas on Unsplash

NLP-related tasks are difficult. There have been numerous approaches to solving them. The most intuitive one is RNN, even though its operations are hard to parallelize and it suffers from exploding or vanishing gradients when exposed to long sequences. Worry not, LSTM and GRU are to the rescue!

There are many ways to arrange RNN cells, one of which is called seq2seq. In seq2seq, the decoder sees only one representation of the input sequence from the last encoder RNN cell. This motivates the use of attention, where different parts of the input sequence can be attended to differently at each generation step of the output sequence.

With the development of self-attention, the RNN cells can be discarded entirely. Bundles of self-attention called multi-head attention along with feed-forward neural networks form the transformer, building state-of-the-art NLP models such as GPT-3, BERT, and many more to tackle many NLP tasks with excellent performance.