Transformer-based Encoder-Decoder Models

The *transformer-based* encoder-decoder model was introduced by Vaswani et al. in the famous Attention is all you need paper and is today the *de-facto* standard encoder-decoder architecture in natural language processing (NLP).

Recently, there has been a lot of research on different *pre-training* objectives for transformer-based encoder-decoder models, *e.g.* T5, Bart, Pegasus, ProphetNet, Marge, *etc*..., but the model architecture has stayed largely the same.

The goal of the blog post is to give an **in-detail** explanation of **how** the transformer-based encoder-decoder architecture models *sequence-to-sequence* problems. We will focus on the mathematical model defined by the architecture and how the model can be used in inference. Along the way, we will give some background on sequence-to-sequence models in NLP and break down the *transformer-based* encoder-decoder architecture into its **encoder** and **decoder** part. We provide many illustrations and establish the link between the theory of *transformer-based* encoder-decoder models and their practical usage in 🤗 Transformers for inference. Note that this blog post does *not* explain how such models can be trained - this will be the topic of a future blog post.

Transformer-based encoder-decoder models are the result of years of research on *representation learning* and *model architectures*. This notebook provides a short summary of the history of neural encoder-decoder models. For more context, the reader is advised to read this awesome blog post by Sebastion Ruder. Additionally, a basic understanding of the *self-attention architecture* is recommended. The following blog post by Jay Alammar serves as a good refresher on the original Transformer model here.

At the time of writing this notebook, 🤗Transformers comprises the encoder-decoder models *T5*, *Bart*, *MarianMT*, and *Pegasus*, which are summarized in the docs under model summaries.

The notebook is divided into four parts:

- **Background** - *A short history of neural encoder-decoder models is given with a focus on on RNN-based models.*
- **Encoder-Decoder** - *The transformer-based encoder-decoder model is presented and it is explained how the model is used for inference.*
- **Encoder** - *The encoder part of the model is explained in detail.*
- **Decoder** - *The decoder part of the model is explained in detail.*

Each part builds upon the previous part, but can also be read on its own.

Background

Tasks in natural language generation (NLG), a subfield of NLP, are best expressed as sequence-to-sequence problems. Such tasks can be defined as finding a model that maps a sequence of input words to a sequence of target words. Some classic examples are *summarization* and *translation*. In the following, we assume that each word is encoded into a vector representation. n input words can then be represented as a sequence of n input vectors:

$$\mathbf{X}_{1:n} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}.$$

Consequently, sequence-to-sequence problems can be solved by finding a mapping f from an input sequence of n vectors $\mathbf{X}_{1:n}$ to a sequence of m target vectors $\mathbf{Y}_{1:m}$, whereas the number of target vectors m is unknown apriori and depends on the input sequence:

$$f : \mathbf{X}_{1:n} \to \mathbf{Y}_{1:m}.$$

Sutskever et al. (2014) noted that deep neural networks (DNN)s, "*despite their flexibility and power can only define a mapping whose inputs and targets can be sensibly encoded with vectors of fixed dimensionality.*"[1]

Using a DNN model [2] to solve sequence-to-sequence problems would therefore mean that the number of target vectors m has to be known *apriori* and would have to be independent of the input $\mathbf{X}_{1:n}$. This is suboptimal because, for tasks in NLG, the number of target words usually depends on the input $\mathbf{X}_{1:n}$ and not just on the input length n. *E.g.* An article of 1000 words can be summarized to both 200 words and 100 words depending on its content.

In 2014, Cho et al. and Sutskever et al. proposed to use an encoder-decoder model purely based on recurrent neural networks (RNNs) for *sequence-to-sequence* tasks. In contrast to DNNS, RNNs are capable of modeling a mapping to a variable number of target vectors. Let's dive a bit deeper into the functioning of RNN-based encoder-decoder models.

During inference, the encoder RNN encodes an input sequence $\mathbf{X}_{1:n}$ by successively updating its *hidden state*[3]. After having processed the last input vector $\mathbf{x}_n$, the encoder's hidden state defines the input *encoding* $\mathbf{c}$. Thus, the encoder defines the mapping:

$$f_{\theta_{enc}} : \mathbf{X}_{1:n} \to \mathbf{c}.$$

Then, the decoder's hidden state is initialized with the input encoding and during inference, the decoder RNN is used to auto-regressively generate the target sequence. Let's explain.

Mathematically, the decoder defines the probability distribution of a target sequence $\mathbf{Y}_{1:m}$ given the hidden state $\mathbf{c}$:

$$p_{\theta_{dec}}(\mathbf{Y}_{1:m} | \mathbf{c}).$$

By Bayes' rule the distribution can be decomposed into conditional distributions of single target vectors as follows:

$$p_{\theta_{dec}}(\mathbf{Y}_{1:m}|\mathbf{c}) = \prod_{i=1}^{m} p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \mathbf{c}).$$

Thus, if the architecture can model the conditional distribution of the next target vector, given all previous target vectors:

$$p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \mathbf{c}), \forall i \in \{1, \ldots, m\},$$

then it can model the distribution of any target vector sequence given the hidden state $\mathbf{c}$ by simply multiplying all conditional probabilities.

So how does the RNN-based decoder architecture model $p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \mathbf{c})$?

In computational terms, the model sequentially maps the previous inner hidden state $\mathbf{c}_{i-1}$ and the previous target vector $\mathbf{y}_i$ to the current inner hidden state $\mathbf{c}_i$ and a *logit vector* $\mathbf{l}_i$ (shown in dark red below):

$$f_{\theta_{\text{dec}}}(\mathbf{y}_{i-1}, \mathbf{c}_{i-1}) \to \mathbf{l}_i, \mathbf{c}_i.$$

$\mathbf{c}_0$ is thereby defined as $\mathbf{c}$ being the output hidden state of the RNN-based encoder. Subsequently, the *softmax* operation is used to transform the logit vector $\mathbf{l}_i$ to a conditional probablity distribution of the next target vector:

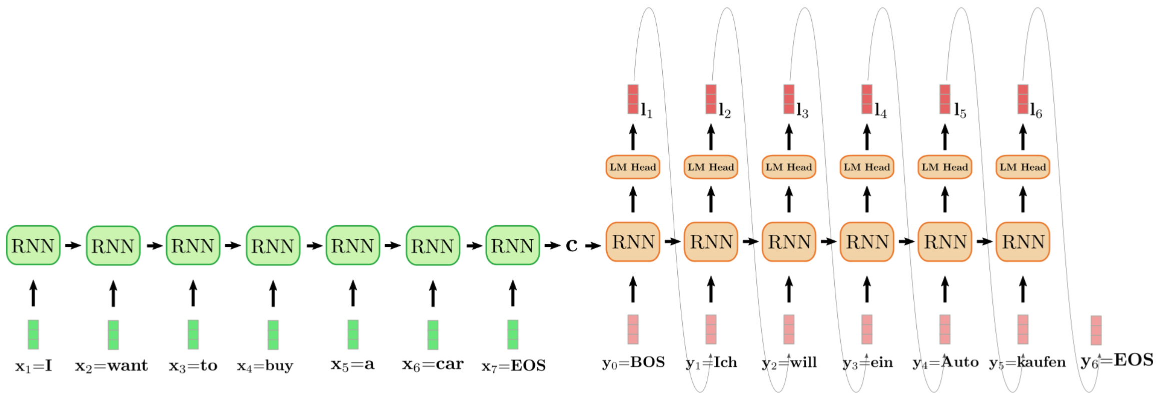
$$p(\mathbf{y}_i|\mathbf{l}_i) = \textbf{Softmax}(\mathbf{l}_i), \text{ with } \mathbf{l}_i = f_{\theta_{\text{dec}}}(\mathbf{y}_{i-1}, \mathbf{c}_{\text{prev}}).$$

For more detail on the logit vector and the resulting probability distribution, please see footnote [4]. From the above equation, we can see that the distribution of the current target vector $\mathbf{y}_i$ is directly conditioned on the previous target vector $\mathbf{y}_{i-1}$ and the previous hidden state $\mathbf{c}_{i-1}$. Because the previous hidden state $\mathbf{c}_{i-1}$ depends on all previous target vectors $\mathbf{y}_0, \ldots, \mathbf{y}_{i-2}$, it can be stated that the RNN-based decoder *implicitly* (*e.g. indirectly*) models the conditional distribution $p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \mathbf{c})$.

The space of possible target vector sequences $\mathbf{Y}_{1:m}$ is prohibitively large so that at inference, one has to rely on decoding methods [5] that efficiently sample high probability target vector sequences from $p_{\theta_{dec}}(\mathbf{Y}_{1:m}|\mathbf{c})$.

Given such a decoding method, during inference, the next input vector $\mathbf{y}_i$ can then be sampled from $p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \mathbf{c})$ and is consequently appended to the input sequence so that the decoder RNN then models $p_{\theta_{\text{dec}}}(\mathbf{y}_{i+1}|\mathbf{Y}_{0:i}, \mathbf{c})$ to sample the next input vector $\mathbf{y}_{i+1}$ and so on in *auto-regressive* fashion.

An important feature of RNN-based encoder-decoder models is the definition of *special* vectors, such as the EOS and BOS vector. The EOS vector often represents the final input vector $\mathbf{x}_n$ to "cue" the encoder that the input sequence has ended and also defines the end of the target sequence. As soon as the EOS is sampled from a logit vector, the generation is complete. The BOS vector represents the input vector $\mathbf{y}_0$ fed to the decoder RNN at the very first decoding step. To output the first logit $\mathbf{l}_1$, an input is required and since no input has been generated at the first step a special BOS input vector is fed to the decoder RNN. Ok - quite complicated! Let's illustrate and walk through an example.



The unfolded RNN encoder is colored in green and the unfolded RNN decoder is colored in red.

The English sentence "I want to buy a car", represented by $\mathbf{x}_1 = \text{I}$, $\mathbf{x}_2 = \text{want}$, $\mathbf{x}_3 = \text{to}$, $\mathbf{x}_4 = \text{buy}$, $\mathbf{x}_5 = \text{a}$, $\mathbf{x}_6 = \text{car}$ and $\mathbf{x}_7 = \text{EOS}$ is translated into German: "Ich will ein Auto kaufen" defined as $\mathbf{y}_0 = \text{BOS}$, $\mathbf{y}_1 = \text{Ich}$, $\mathbf{y}_2 = \text{will}$, $\mathbf{y}_3 = \text{ein}$, $\mathbf{y}_4 = \text{Auto}, \mathbf{y}_5 = \text{kaufen}$ and $\mathbf{y}_6 = \text{EOS}$. To begin with, the input vector $\mathbf{x}_1 = \text{I}$ is processed by the encoder RNN and updates its hidden state. Note that because we are only interested in the final encoder's hidden state $\mathbf{c}$, we can disregard the RNN encoder's target vector. The encoder RNN then processes the rest of the input sentence want, to, buy, a, car, EOS in the same fashion, updating its hidden state at each step until the vector $\mathbf{x}_7 = EOS$ is reached [6]. In the illustration above the horizontal arrow connecting the unfolded encoder RNN represents the sequential updates of the hidden state. The final hidden state of the encoder RNN, represented by $\mathbf{c}$ then completely defines the *encoding* of the input sequence and is used as the initial hidden state of the decoder RNN. This can be seen as *conditioning* the decoder RNN on the encoded input.

To generate the first target vector, the decoder is fed the BOS vector, illustrated as $\mathbf{y}_0$ in the design above. The target vector of the RNN is then further mapped to the logit vector $\mathbf{l}_1$ by means of the *LM Head* feed-forward layer to define the conditional distribution of the first target vector as explained above:

$$p_{\theta_{dec}}(\mathbf{y}|\text{BOS}, \mathbf{c}).$$

The word Ich is sampled (shown by the grey arrow, connecting $\mathbf{l}_1$ and $\mathbf{y}_1$) and consequently the second target vector can be sampled:

$$\text{will} \sim p_{\theta_{dec}}(\mathbf{y}|\text{BOS}, \text{Ich}, \mathbf{c}).$$

And so on until at step $i = 6$, the EOS vector is sampled from $\mathbf{l}_6$ and the decoding is finished. The resulting target sequence amounts to $\mathbf{Y}_{1:6} = \{\mathbf{y}_1, \ldots, \mathbf{y}_6\}$, which is "Ich will ein Auto kaufen" in our example above.

To sum it up, an RNN-based encoder-decoder model, represented by $f_{\theta_{\text{enc}}}$ and $p_{\theta_{\text{dec}}}$ defines the distribution $p(\mathbf{Y}_{1:m}|\mathbf{X}_{1:n})$ by factorization:

$$p_{\theta_{\text{enc}},\theta_{\text{dec}}}(\mathbf{Y}_{1:m}|\mathbf{X}_{1:n}) = \prod_{i=1}^{m} p_{\theta_{\text{enc}},\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \mathbf{X}_{1:n}) = \prod_{i=1}^{m} p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \mathbf{c}), \text{ with } \mathbf{c} = f_{\theta_{enc}}($$

During inference, efficient decoding methods can auto-regressively generate the target sequence $\mathbf{Y}_{1:m}$.

The RNN-based encoder-decoder model took the NLG community by storm. In 2016, Google announced to fully replace its heavily feature engineered translation service by a single RNN-based encoder-decoder model (see here).

Nevertheless, RNN-based encoder-decoder models have two pitfalls. First, RNNs suffer from the vanishing gradient problem, making it very difficult to capture long-range dependencies, *cf.* Hochreiter et al. (2001). Second, the inherent recurrent architecture of RNNs prevents efficient parallelization when encoding, *cf.* Vaswani et al. (2017).

Encoder-Decoder

In 2017, Vaswani et al. introduced the **Transformer** and thereby gave birth to *transformer-based* encoder-decoder models.

Analogous to RNN-based encoder-decoder models, transformer-based encoder-decoder models consist of an encoder and a decoder which are both stacks of *residual attention blocks*. The key innovation of transformer-based encoder-decoder models is that such residual attention blocks can process an input sequence $\mathbf{X}_{1:n}$ of variable length n without exhibiting a recurrent structure. Not relying on a recurrent structure allows transformer-based encoder-decoders to be highly parallelizable, which makes the model orders of magnitude more computationally efficient than RNN-based encoder-decoder models on modern hardware.

As a reminder, to solve a *sequence-to-sequence* problem, we need to find a mapping of an input sequence $\mathbf{X}_{1:n}$ to an output sequence $\mathbf{Y}_{1:m}$ of variable length m. Let's see how transformer-based encoder-decoder models are used to find such a mapping.

Similar to RNN-based encoder-decoder models, the transformer-based encoder-decoder models define a conditional distribution of target vectors $\mathbf{Y}_{1:n}$ given an input sequence $\mathbf{X}_{1:n}$:

$$p_{\theta_{\text{enc}},\theta_{\text{dec}}}(\mathbf{Y}_{1:m}|\mathbf{X}_{1:n}).$$

The transformer-based encoder part encodes the input sequence $\mathbf{X}_{1:n}$ to a *sequence* of *hidden states* $\overline{\mathbf{X}}_{1:n}$, thus defining the mapping:

$$f_{\theta_{\text{enc}}} : \mathbf{X}_{1:n} \to \overline{\mathbf{X}}_{1:n}.$$

The transformer-based decoder part then models the conditional probability distribution of the target vector sequence $\mathbf{Y}_{1:n}$ given the sequence of encoded hidden states $\overline{\mathbf{X}}_{1:n}$:

$$p_{\theta_{dec}}(\mathbf{Y}_{1:n}|\overline{\mathbf{X}}_{1:n}).$$

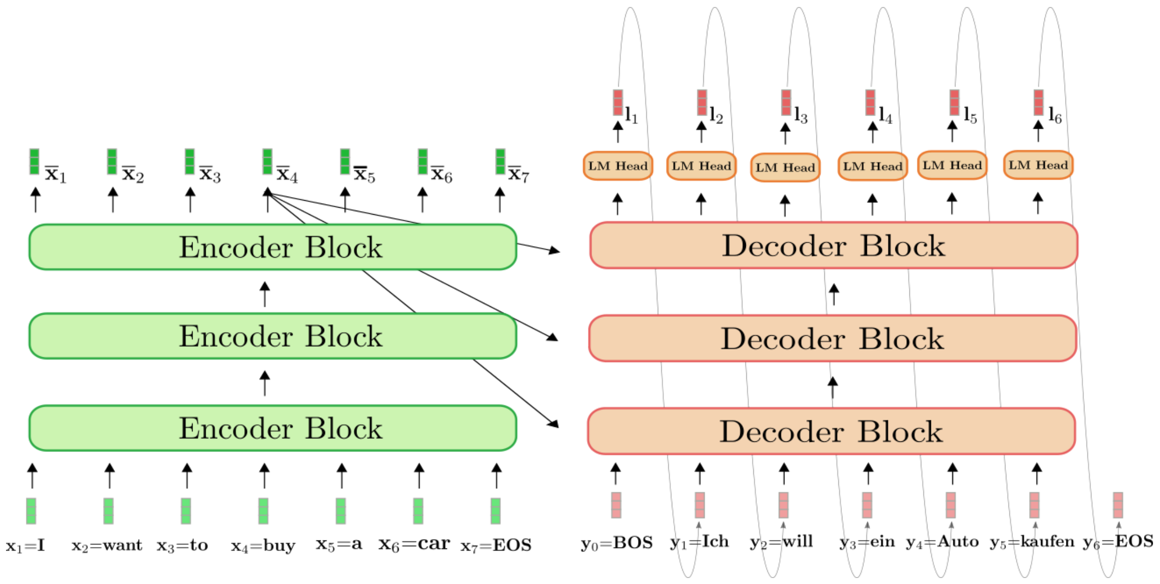
By Bayes' rule, this distribution can be factorized to a product of conditional probability distribution of the target vector $\mathbf{y}_i$ given the encoded hidden states $\overline{\mathbf{X}}_{1:n}$ and all previous target vectors $\mathbf{Y}_{0:i-1}$:

$$p_{\theta_{dec}}(\mathbf{Y}_{1:n}|\overline{\mathbf{X}}_{1:n}) = \prod_{i=1}^{n} p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \overline{\mathbf{X}}_{1:n}).$$

The transformer-based decoder hereby maps the sequence of encoded hidden states $\overline{\mathbf{X}}_{1:n}$ and all previous target vectors $\mathbf{Y}_{0:i-1}$ to the *logit* vector $\mathbf{l}_i$. The logit vector $\mathbf{l}_i$ is then processed by the *softmax* operation to define the conditional distribution $p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \overline{\mathbf{X}}_{1:n})$, just as it is done for RNN-based decoders. However, in contrast to RNN-based decoders, the distribution of the target vector $\mathbf{y}_i$ is *explicitly* (or directly) conditioned on all previous target vectors $\mathbf{y}_0, \ldots, \mathbf{y}_{i-1}$ as we will see later in more detail. The 0th target vector $\mathbf{y}_0$ is hereby represented by a special "begin-of-sentence" BOS vector.

Having defined the conditional distribution $p_{\theta_{\text{dec}}}(\mathbf{y}_i|\mathbf{Y}_{0:i-1}, \overline{\mathbf{X}}_{1:n})$, we can now *auto-regressively* generate the output and thus define a mapping of an input sequence $\mathbf{X}_{1:n}$ to an output sequence $\mathbf{Y}_{1:m}$ at inference.

Let's visualize the complete process of *auto-regressive* generation of *transformer-based* encoder-decoder models.



The transformer-based encoder is colored in green and the transformer-based decoder is colored in red. As in the previous section, we show how the English sentence "I want to buy a car", represented by $\mathbf{x}_1 = \text{I}$, $\mathbf{x}_2 = \text{want}$, $\mathbf{x}_3 = \text{to}$, $\mathbf{x}_4 = \text{buy}$, $\mathbf{x}_5 = \text{a}$, $\mathbf{x}_6 = \text{car}$, and

$\mathbf{x}_7 = \text{EOS}$ is translated into German: "Ich will ein Auto kaufen" defined as $\mathbf{y}_0 = \text{BOS}$, $\mathbf{y}_1 = \text{Ich}$, $\mathbf{y}_2 = \text{will}$, $\mathbf{y}_3 = \text{ein}$, $\mathbf{y}_4 = \text{Auto}, \mathbf{y}_5 = \text{kaufen}$, and $\mathbf{y}_6 = \text{EOS}$.

To begin with, the encoder processes the complete input sequence $\mathbf{X}_{1:7}$ = "I want to buy a car" (represented by the light green vectors) to a contextualized encoded sequence $\overline{\mathbf{X}}_{1:7}$. *E.g.* $\overline{\mathbf{x}}_4$ defines an encoding that depends not only on the input $\mathbf{x}_4$ = "buy", but also on all other words "I", "want", "to", "a", "car" and "EOS", *i.e.* the context.

Next, the input encoding $\overline{\mathbf{X}}_{1:7}$ together with the BOS vector, *i.e.* $\mathbf{y}_0$, is fed to the decoder. The decoder processes the inputs $\overline{\mathbf{X}}_{1:7}$ and $\mathbf{y}_0$ to the first logit $\mathbf{l}_1$ (shown in darker red) to define the conditional distribution of the first target vector $\mathbf{y}_1$:

$$p_{\theta_{enc,dec}}(\mathbf{y}|\mathbf{y}_0, \mathbf{X}_{1:7}) = p_{\theta_{enc,dec}}(\mathbf{y}|\text{BOS}, \text{I want to buy a car EOS}) = p_{\theta_{dec}}(\mathbf{y}|\text{BOS}, \overline{\mathbf{X}}_{1:7}).$$

Next, the first target vector $\mathbf{y}_1$ = Ich is sampled from the distribution (represented by the grey arrows) and can now be fed to the decoder again. The decoder now processes both $\mathbf{y}_0$ = "BOS" and $\mathbf{y}_1$ = "Ich" to define the conditional distribution of the second target vector $\mathbf{y}_2$:

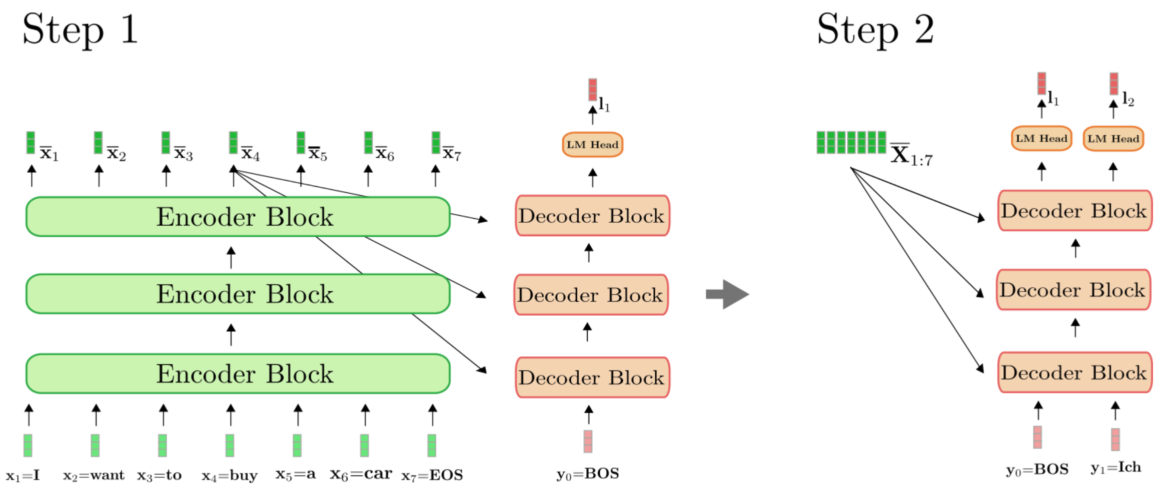
$$p_{\theta_{dec}}(\mathbf{y}|\text{BOS Ich}, \overline{\mathbf{X}}_{1:7}).$$

We can sample again and produce the target vector $\mathbf{y}_2$ = "will". We continue in auto-regressive fashion until at step 6 the EOS vector is sampled from the conditional distribution:

$$\text{EOS} \sim p_{\theta_{dec}}(\mathbf{y}|\text{BOS Ich will ein Auto kaufen}, \overline{\mathbf{X}}_{1:7}).$$

And so on in auto-regressive fashion.

It is important to understand that the encoder is only used in the first forward pass to map $\mathbf{X}_{1:n}$ to $\overline{\mathbf{X}}_{1:n}$. As of the second forward pass, the decoder can directly make use of the previously calculated encoding $\overline{\mathbf{X}}_{1:n}$. For clarity, let's illustrate the first and the second forward pass for our example above.



Step 1
Step 2
Encoder Block
Encoder Block
Encoder Block
Decoder Block
Decoder Block
Decoder Block
LM Head
$\mathbf{l}_1$
$\mathbf{l}_2$
$\overline{\mathbf{x}}_1$ $\overline{\mathbf{x}}_2$ $\overline{\mathbf{x}}_3$ $\overline{\mathbf{x}}_4$ $\overline{\mathbf{x}}_5$ $\overline{\mathbf{x}}_6$ $\overline{\mathbf{x}}_7$
$\overline{\mathbf{X}}_{1:7}$
x_1=I x_2=want x_3=to x_4=buy x_5=a x_6=car x_7=EOS
y_0=BOS
y_1=Ich

As can be seen, only in step $i = 1$ do we have to encode "I want to buy a car EOS" to $\overline{\mathbf{X}}_{1:7}$. At step $i = 2$, the contextualized encodings of "I want to buy a car EOS" are simply reused by the decoder.

In 🤗Transformers, this auto-regressive generation is done under-the-hood when calling the `.generate()` method. Let's use one of our translation models to see this in action.

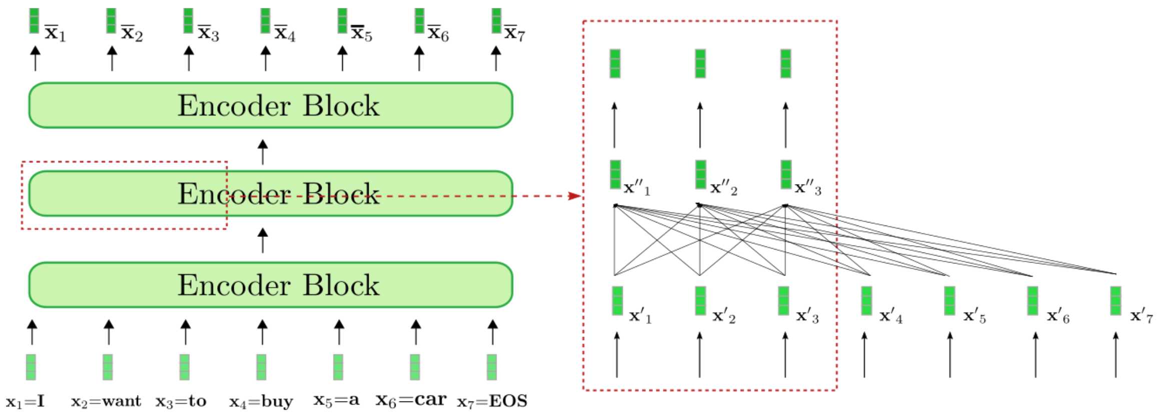
Encoder

As mentioned in the previous section, the *transformer-based* encoder maps the input sequence to a contextualized encoding sequence:

$$f_{\theta_{\text{enc}}} : \mathbf{X}_{1:n} \to \overline{\mathbf{X}}_{1:n}.$$

Taking a closer look at the architecture, the transformer-based encoder is a stack of residual *encoder blocks*. Each encoder block consists of a **bi-directional** self-attention layer, followed by two feed-forward layers. For simplicity, we disregard the normalization layers in this notebook. Also, we will not further discuss the role of the two feed-forward layers, but simply see it as a final vector-to-vector mapping required in each encoder block ^{1}. The bi-directional self-attention layer puts each input vector $\mathbf{x'}_j, \forall j \in \{1, \ldots, n\}$ into relation with all input vectors $\mathbf{x'}_1, \ldots, \mathbf{x'}_n$ and by doing so transforms the input vector $\mathbf{x'}_j$ to a more "refined" contextual representation of itself, defined as $\mathbf{x''}_j$. Thereby, the first encoder block transforms each input vector of the input sequence $\mathbf{X}_{1:n}$ (shown in light green below) from a *context-independent* vector representation to a *context-dependent* vector representation, and the following encoder blocks further refine this contextual representation until the last encoder block outputs the final contextual encoding $\overline{\mathbf{X}}_{1:n}$ (shown in darker green below).

Let's visualize how the encoder processes the input sequence "I want to buy a car EOS" to a contextualized encoding sequence. Similar to RNN-based encoders, transformer-based encoders also add a special "end-of-sequence" input vector to the input sequence to hint to the model that the input vector sequence is finished ^{2}.



$\overline{\mathbf{x}}_1$ $\overline{\mathbf{x}}_2$ $\overline{\mathbf{x}}_3$ $\overline{\mathbf{x}}_4$ $\overline{\mathbf{x}}_5$ $\overline{\mathbf{x}}_6$ $\overline{\mathbf{x}}_7$
Encoder Block
Encoder Block
Encoder Block
$\mathbf{x}_1$=I $\mathbf{x}_2$=want $\mathbf{x}_3$=to $\mathbf{x}_4$=buy $\mathbf{x}_5$=a $\mathbf{x}_6$=car $\mathbf{x}_7$=EOS
$\mathbf{x''}_1$ $\mathbf{x''}_2$ $\mathbf{x''}_3$
$\mathbf{x'}_1$ $\mathbf{x'}_2$ $\mathbf{x'}_3$ $\mathbf{x'}_4$ $\mathbf{x'}_5$ $\mathbf{x'}_6$ $\mathbf{x'}_7$

Our exemplary *transformer-based* encoder is composed of three encoder blocks, whereas the second encoder block is shown in more detail in the red box on the right for the first three input vectors $\mathbf{x}_1, \mathbf{x}_2 and \mathbf{x}_3$. The bi-directional self-attention mechanism is illustrated by the fully-connected graph in the lower part of the red box and the two feed-forward layers are shown in the upper part of the red box. As stated before, we will focus only on the bi-directional self-attention mechanism.

As can be seen each output vector of the self-attention layer $\mathbf{x''}_i, \forall i \in \{1, \ldots, 7\}$ depends *directly* on *all* input vectors $\mathbf{x'}_1, \ldots, \mathbf{x'}_7$. This means, *e.g.* that the input vector representation of the word "want", *i.e.* $\mathbf{x'}_2$, is put into direct relation with the word "buy", *i.e.* $\mathbf{x'}_4$, but also with the word "I",*i.e.* $\mathbf{x'}_1$. The output vector representation of "want", *i.e.* $\mathbf{x''}_2$, thus represents a more refined contextual representation for the word "want".

Let's take a deeper look at how bi-directional self-attention works. Each input vector $\mathbf{x'}_i$ of an input sequence $\mathbf{X'}_{1:n}$ of an encoder block is projected to a key vector $\mathbf{k}_i$, value vector $\mathbf{v}_i$ and query vector $\mathbf{q}_i$ (shown in orange, blue, and purple respectively below) through three trainable weight matrices $\mathbf{W}_q, \mathbf{W}_v, \mathbf{W}_k$:

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x'}_i,$$

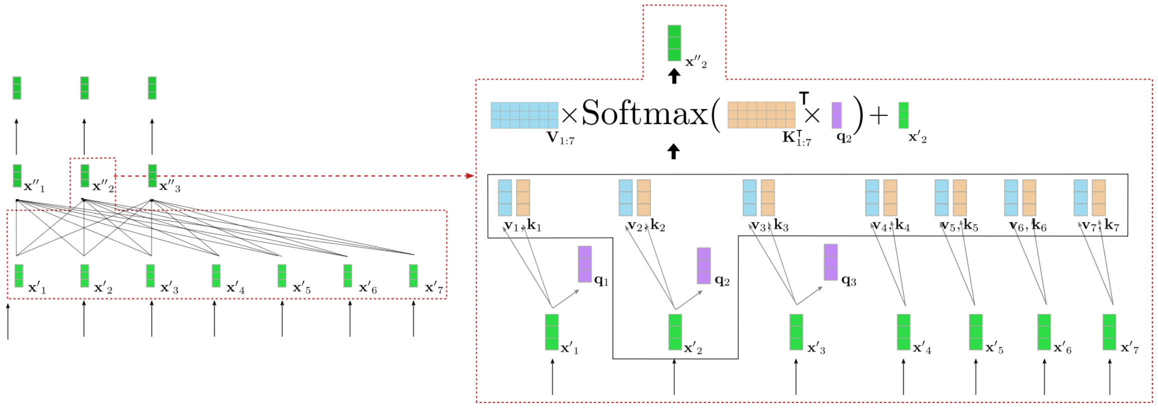
$$\mathbf{v}_i = \mathbf{W}_v \mathbf{x'}_i,$$

$$\mathbf{k}_i = \mathbf{W}_k \mathbf{x'}_i,$$

$$\forall i \in \{1, \ldots n\}.$$

Note, that the **same** weight matrices are applied to each input vector $\mathbf{x}_i, \forall i \in \{i, \ldots, n\}$. After projecting each input vector $\mathbf{x}_i$ to a query, key, and value vector, each query vector $\mathbf{q}_j, \forall j \in \{1, \ldots, n\}$ is compared to all key vectors $\mathbf{k}_1, \ldots, \mathbf{k}_n$. The more similar one of the key vectors $\mathbf{k}_1, \ldots \mathbf{k}_n$ is to a query vector $\mathbf{q}_j$, the more important is the corresponding value vector $\mathbf{v}_j$ for the output vector $\mathbf{x''}_j$. More specifically, an output vector $\mathbf{x''}_j$ is defined as the weighted sum of all value vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ plus the input vector $\mathbf{x'}_j$. Thereby, the weights are proportional to the cosine similarity between $\mathbf{q}_j$ and the respective key vectors $\mathbf{k}_1, \ldots, \mathbf{k}_n$, which is mathematically expressed by $\textbf{Softmax}(\mathbf{K}_{1:n}^\intercal \mathbf{q}_j)$ as illustrated in the equation below. For a complete description of the self-attention layer, the reader is advised to take a look at this blog post or the original paper.

Alright, this sounds quite complicated. Let's illustrate the bi-directional self-attention layer for one of the query vectors of our example above. For simplicity, it is assumed that our exemplary *transformer-based* decoder uses only a single attention head `config.num_heads = 1` and that no normalization is applied.



On the left, the previously illustrated second encoder block is shown again and on the right, an in detail visualization of the bi-directional self-attention mechanism is given for the second input vector $\mathbf{x'}_2$ that corresponds to the input word "want". At first all input vectors $\mathbf{x'}_1, \ldots, \mathbf{x'}_7$ are projected to their respective query vectors $\mathbf{q}_1, \ldots, \mathbf{q}_7$ (only the first three query vectors are shown in purple above), value vectors $\mathbf{v}_1, \ldots, \mathbf{v}_7$ (shown in blue), and key vectors $\mathbf{k}_1, \ldots, \mathbf{k}_7$ (shown in orange). The query vector $\mathbf{q}_2$ is then multiplied by the transpose of all key vectors, *i.e.* $\mathbf{K}_{1:7}^{\intercal}$ followed by the softmax operation to yield the *self-attention weights*. The self-attention weights are finally multiplied by the respective value vectors and the input vector $\mathbf{x'}_2$ is added to output the "refined" representation of the word "want", *i.e.* $\mathbf{x''}_2$ (shown in dark green on the right). The whole equation is illustrated in the upper part of the box on the right. The multiplication of $\mathbf{K}_{1:7}^{\intercal}$ and $\mathbf{q}_2$ thereby makes it possible to compare the vector representation of "want" to all other input vector representations "I", "to", "buy", "a", "car", "EOS" so that the self-attention weights mirror the importance each of the other input vector representations $\mathbf{x'}_j \text{, with } j \ne 2$ for the refined representation $\mathbf{x''}_2$ of the word "want".

To further understand the implications of the bi-directional self-attention layer, let's assume the following sentence is processed: "*The house is beautiful and well located in the middle of the city where it is easily accessible by public transport*". The word "it" refers to "house", which is 12 "positions away". In transformer-based encoders, the bi-directional self-attention layer performs a single mathematical operation to put the input vector of "house" into relation with the input vector of "it" (compare to the first illustration of this section). In contrast, in an RNN-based encoder, a word that is 12 "positions away", would require at least 12 mathematical operations meaning that in an RNN-based encoder a linear number of mathematical operations are required. This makes it much harder for an RNN-based encoder to model long-range contextual representations. Also, it becomes clear that a transformer-based encoder is much less prone to lose important information than an RNN-based encoder-decoder model because the sequence length of the encoding is kept the same, *i.e.* $\textbf{len}(\mathbf{X}_{1:n}) = \textbf{len}(\mathbf{\overline{X}}_{1:n}) = n$, while an RNN compresses the length from $\textbf{len}((\mathbf{X}_{1:n}) = n$ to just $\textbf{len}(\mathbf{c}) = 1$, which makes it very difficult for RNNs to effectively encode long-range dependencies between input words.

In addition to making long-range dependencies more easily learnable, we can see that the Transformer architecture is able to process text in parallel.Mathematically, this can easily be shown by writing the self-attention formula as a product of query, key, and value matrices:

$$\mathbf{X''}_{1:n} = \mathbf{V}_{1:n}\text{Softmax}(\mathbf{Q}_{1:n}^\intercal \mathbf{K}_{1:n}) + \mathbf{X'}_{1:n}.$$

The output $\mathbf{X''}_{1:n} = \mathbf{x''}_1, \ldots, \mathbf{x''}_n$ is computed via a series of matrix multiplications and a softmax operation, which can be parallelized effectively. Note, that in an RNN-based encoder model, the computation of the hidden state $\mathbf{c}$ has to be done sequentially: Compute hidden state of the first input vector $\mathbf{x}_1$, then compute the hidden state of the second input vector that depends on the hidden state of the first hidden vector, etc. The sequential nature of RNNs prevents effective parallelization and makes them much more inefficient compared to transformer-based encoder models on modern GPU hardware.

Great, now we should have a better understanding of a) how transformer-based encoder models effectively model long-range contextual representations and b) how they efficiently process long sequences of input vectors.

Now, let's code up a short example of the encoder part of our `MarianMT` encoder-decoder models to verify that the explained theory holds in practice.

---

^{1} An in-detail explanation of the role the feed-forward layers play in transformer-based models is out-of-scope for this notebook. It is argued in Yun et. al, (2017) that feed-forward layers are crucial to map each contextual vector $\mathbf{x'}_i$ individually to the desired output space, which the *self-attention* layer does not manage to do on its own. It should be noted here, that each output token $\mathbf{x'}$ is processed by the same feed-forward layer. For more detail, the reader is advised to read the paper.

^{2} However, the EOS input vector does not have to be appended to the input sequence, but has been shown to improve performance in many cases. In contrast to the *0th* BOS target vector of the transformer-based decoder is required as a starting input vector to predict a first target vector.

Decoder

As mentioned in the *Encoder-Decoder* section, the *transformer-based* decoder defines the conditional probability distribution of a target sequence given the contextualized encoding sequence:

$$p_{\theta_{dec}}(\mathbf{Y}_{1:m} | \mathbf{\overline{X}}_{1:n}),$$

which by Bayes' rule can be decomposed into a product of conditional distributions of the next target vector given the contextualized encoding sequence and all previous target vectors:

$$p_{\theta_{dec}}(\mathbf{Y}_{1:m} | \mathbf{\overline{X}}_{1:n}) = \prod_{i=1}^{m} p_{\theta_{dec}}(\mathbf{y}_i | \mathbf{Y}_{0:i-1}, \mathbf{\overline{X}}_{1:n}).$$

Let's first understand how the transformer-based decoder defines a probability distribution. The transformer-based decoder is a stack of *decoder blocks* followed by a dense layer, the "LM head". The stack of decoder blocks maps the contextualized encoding sequence $\overline{\mathbf{X}}_{1:n}$ and a target vector sequence prepended by the BOS vector and cut to the last target vector, *i.e.* $\mathbf{Y}_{0:i-1}$, to an encoded sequence of target vectors $\overline{\mathbf{Y}}_{0:i-1}$. Then, the "LM head" maps the encoded sequence of target vectors $\overline{\mathbf{Y}}_{0:i-1}$ to a sequence of logit vectors $\mathbf{L}_{1:n} = \mathbf{l}_1, \ldots, \mathbf{l}_n$, whereas the dimensionality of each logit vector $\mathbf{l}_i$ corresponds to the size of the vocabulary. This way, for each $i \in \{1, \ldots, n\}$ a probability distribution over the whole vocabulary can be obtained by applying a softmax operation on $\mathbf{l}_i$. These distributions define the conditional distribution:

$$p_{\theta_{dec}}(\mathbf{y}_i | \mathbf{Y}_{0:i-1}, \overline{\mathbf{X}}_{1:n}), \forall i \in \{1, \ldots, n\},$$

respectively. The "LM head" is often tied to the transpose of the word embedding matrix, *i.e.* $\mathbf{W}_{\text{emb}}^{\intercal} = \left[\mathbf{y}^1, \ldots, \mathbf{y}^{\text{vocab}}\right]^{\intercal}$ [1]. Intuitively this means that for all $i \in \{0, \ldots, n-1\}$ the "LM Head" layer compares the encoded output vector $\overline{\mathbf{y}}_i$ to all word embeddings in the vocabulary $\mathbf{y}^1, \ldots, \mathbf{y}^{\text{vocab}}$ so that the logit vector $\mathbf{l}_{i+1}$ represents the similarity scores between the encoded output vector and each word embedding. The softmax operation simply transformers the similarity scores to a probability distribution. For each $i \in \{1, \ldots, n\}$, the following equations hold:

$$p_{\theta_{dec}}(\mathbf{y} | \overline{\mathbf{X}}_{1:n}, \mathbf{Y}_{0:i-1})$$

$$= \text{Softmax}(f_{\theta_{\text{dec}}}(\overline{\mathbf{X}}_{1:n}, \mathbf{Y}_{0:i-1}))$$

$$= \text{Softmax}(\mathbf{W}_{\text{emb}}^{\intercal} \overline{\mathbf{y}}_{i-1})$$

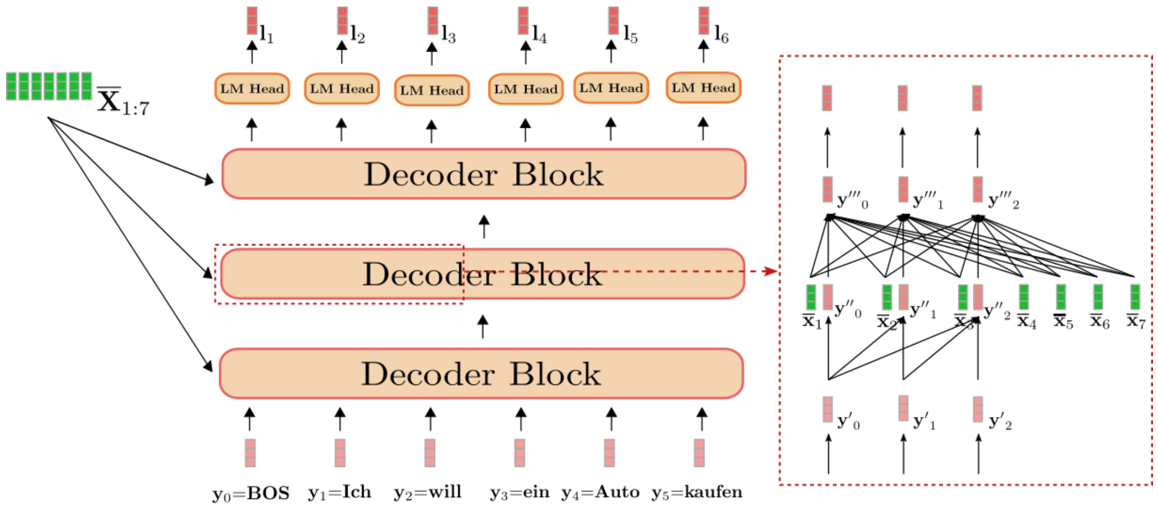
$$= \text{Softmax}(\mathbf{l}_i).$$

Putting it all together, in order to model the conditional distribution of a target vector sequence $\mathbf{Y}_{1:m}$, the target vectors $\mathbf{Y}_{1:m-1}$ prepended by the special BOS vector, *i.e.* $\mathbf{y}_0$, are first mapped together with the contextualized encoding sequence $\overline{\mathbf{X}}_{1:n}$ to the logit vector sequence $\mathbf{L}_{1:m}$. Consequently, each logit target vector $\mathbf{l}_i$ is transformed into a conditional probability distribution of the target vector $\mathbf{y}_i$ using the softmax operation. Finally, the conditional probabilities of all target vectors $\mathbf{y}_1, \ldots, \mathbf{y}_m$ multiplied together to yield the conditional probability of the complete target vector sequence:

$$p_{\theta_{dec}}(\mathbf{Y}_{1:m} | \overline{\mathbf{X}}_{1:n}) = \prod_{i=1}^{m} p_{\theta_{dec}}(\mathbf{y}_i | \mathbf{Y}_{0:i-1}, \overline{\mathbf{X}}_{1:n}).$$

In contrast to transformer-based encoders, in transformer-based decoders, the encoded output vector $\overline{\mathbf{y}}_i$ should be a good representation of the *next* target vector $\mathbf{y}_{i+1}$ and not of the input vector itself. Additionally, the encoded output vector $\overline{\mathbf{y}}_i$ should be conditioned on all contextualized encoding sequence $\overline{\mathbf{X}}_{1:n}$. To meet these requirements each decoder block consists of a **uni-directional** self-attention layer, followed by a **cross-attention** layer and two feed-forward layers [2]. The uni-directional self-attention layer puts each of its input vectors $\mathbf{y}'_j$ only into relation with all previous input vectors $\mathbf{y}'_i,$ with $i \leq q$ for all $j \in \{1, \ldots, n\}$ to model the probability distribution of the next target vectors. The cross-attention layer puts each of its input vectors $\mathbf{y}''_j$ into relation with all contextualized encoding vectors $\overline{\mathbf{X}}_{1:n}$ to condition the probability distribution of the next target vectors on the input of the encoder as well.

Alright, let's visualize the *transformer-based* decoder for our English to German translation example.



$\overline{\mathbf{X}}_{1:7}$
$\mathbf{l}_1$ $\mathbf{l}_2$ $\mathbf{l}_3$ $\mathbf{l}_4$ $\mathbf{l}_5$ $\mathbf{l}_6$
LM Head
Decoder Block
Decoder Block
Decoder Block
$\mathbf{y}_0$=BOS $\mathbf{y}_1$=Ich $\mathbf{y}_2$=will $\mathbf{y}_3$=ein $\mathbf{y}_4$=Auto $\mathbf{y}_5$=kaufen
$\mathbf{y}'''_0$ $\mathbf{y}'''_1$ $\mathbf{y}'''_2$
$\overline{\mathbf{x}}_1$ $\mathbf{y}''_0$ $\overline{\mathbf{x}}_2$ $\mathbf{y}''_1$ $\overline{\mathbf{x}}_3$ $\mathbf{y}''_2$ $\overline{\mathbf{x}}_4$ $\overline{\mathbf{x}}_5$ $\overline{\mathbf{x}}_6$ $\overline{\mathbf{x}}_7$
$\mathbf{y}'_0$ $\mathbf{y}'_1$ $\mathbf{y}'_2$

We can see that the decoder maps the input $\mathbf{Y}_{0:5}$ "BOS", "Ich", "will", "ein", "Auto", "kaufen" (shown in light red) together with the contextualized sequence of "I", "want", "to", "buy", "a", "car", "EOS", *i.e.* $\overline{\mathbf{X}}_{1:7}$ (shown in dark green) to the logit vectors $\mathbf{L}_{1:6}$ (shown in dark red).

Applying a softmax operation on each $\mathbf{l}_1, \mathbf{l}_2, \ldots, \mathbf{l}_5$ can thus define the conditional probability distributions:

$$p_{\theta_{dec}}(\mathbf{y}|\text{BOS}, \overline{\mathbf{X}}_{1:7}),$$

$$p_{\theta_{dec}}(\mathbf{y}|\text{BOS Ich}, \overline{\mathbf{X}}_{1:7}),$$

$$\ldots,$$

$$p_{\theta_{dec}}(\mathbf{y}|\text{BOS Ich will ein Auto kaufen}, \overline{\mathbf{X}}_{1:7}).$$

The overall conditional probability of:

$$p_{\theta_{dec}}(\text{Ich will ein Auto kaufen EOS}|\overline{\mathbf{X}}_{1:n})$$

can therefore be computed as the following product:

$$p_{\theta_{dec}}(\text{Ich}|\text{BOS}, \overline{\mathbf{X}}_{1:7}) \times \ldots \times p_{\theta_{dec}}(\text{EOS}|\text{BOS Ich will ein Auto kaufen}, \overline{\mathbf{X}}_{1:7}).$$

The red box on the right shows a decoder block for the first three target vectors $\mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2$. In the lower part, the uni-directional self-attention mechanism is illustrated and in the middle, the cross-attention mechanism is illustrated. Let's first focus on uni-directional self-attention.

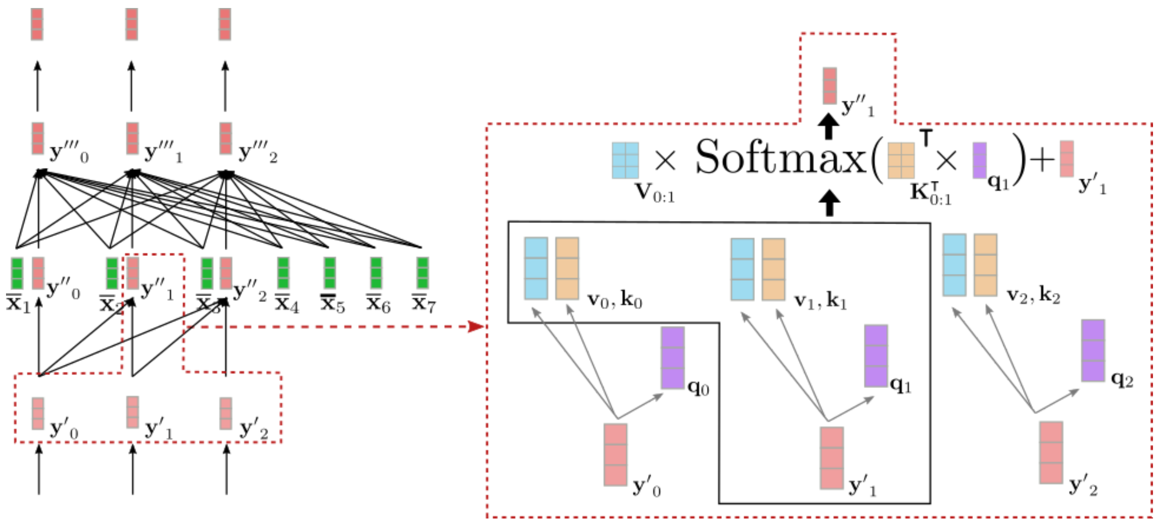
As in bi-directional self-attention, in uni-directional self-attention, the query vectors $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$ (shown in purple below), key vectors $\mathbf{k}_0, \ldots, \mathbf{k}_{m-1}$ (shown in orange below), and value vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{m-1}$ (shown in blue below) are projected from their respective input vectors $\mathbf{y}'_0, \ldots, \mathbf{y}_{m-1}$ (shown in light red below). However, in uni-directional self-attention, each query vector $\mathbf{q}_i$ is compared *only* to its respective key vector and all previous ones, namely $\mathbf{k}_0, \ldots, \mathbf{k}_i$ to yield the respective *attention weights*. This prevents an output vector $\mathbf{y}''_j$ (shown in dark red below) to include any information about the following input vector $\mathbf{y}_i, \text{ with } i > 1$ for all $j \in \{0, \ldots, m-1\}$. As is the case in bi-directional self-attention, the attention weights are then multiplied by their respective value vectors and summed together.

We can summarize uni-directional self-attention as follows:

$$\mathbf{y}''_i = \mathbf{V}_{0:i} \textbf{Softmax}(\mathbf{K}^\intercal_{0:i} \mathbf{q}_i) + \mathbf{y}'_i.$$

Note that the index range of the key and value vectors is $0 : i$ instead of $0 : m - 1$ which would be the range of the key vectors in bi-directional self-attention.

Let's illustrate uni-directional self-attention for the input vector $\mathbf{y}'_1$ for our example above.



As can be seen $\mathbf{y}''_1$ only depends on $\mathbf{y}'_0$ and $\mathbf{y}'_1$. Therefore, we put the vector representation of the word "Ich", *i.e.* $\mathbf{y}'_1$ only into relation with itself and the "BOS" target vector, *i.e.* $\mathbf{y}'_0$, but **not** with the vector representation of the word "will", *i.e.* $\mathbf{y}'_2$.

So why is it important that we use uni-directional self-attention in the decoder instead of bi-directional self-attention? As stated above, a transformer-based decoder defines a mapping from a sequence of input vector $\mathbf{Y}_{0:m-1}$ to the logits corresponding to the **next** decoder input vectors, namely $\mathbf{L}_{1:m}$. In our example, this means, *e.g.* that the input vector $\mathbf{y}_1$ = "Ich" is mapped to the logit vector $\mathbf{l}_2$, which is then used to predict the input vector $\mathbf{y}_2$. Thus, if $\mathbf{y}'_1$ would have access to the following input vectors $\mathbf{Y}'_{2:5}$, the decoder would simply copy the vector representation of "will", *i.e.* $\mathbf{y}'_2$, to be its output $\mathbf{y}''_1$. This would be forwarded to the last layer so that the encoded output vector $\overline{\mathbf{y}}_1$ would essentially just correspond to the vector representation $\mathbf{y}_2$.

This is obviously disadvantageous as the transformer-based decoder would never learn to predict the next word given all previous words, but just copy the target vector $\mathbf{y}_i$ through the network to $\overline{\mathbf{y}}_{i-1}$ for all $i \in \{1, \ldots, m\}$. In order to define a conditional distribution of the next target vector, the distribution cannot be conditioned on the next target vector itself. It does not make much sense to predict $\mathbf{y}_i$ from $p(\mathbf{y}|\mathbf{Y}_{0:i}, \overline{\mathbf{X}})$ because the distribution is conditioned on the target vector it is supposed to model. The uni-directional self-attention architecture, therefore, allows us to define a *causal* probability distribution, which is necessary to effectively model a conditional distribution of the next target vector.

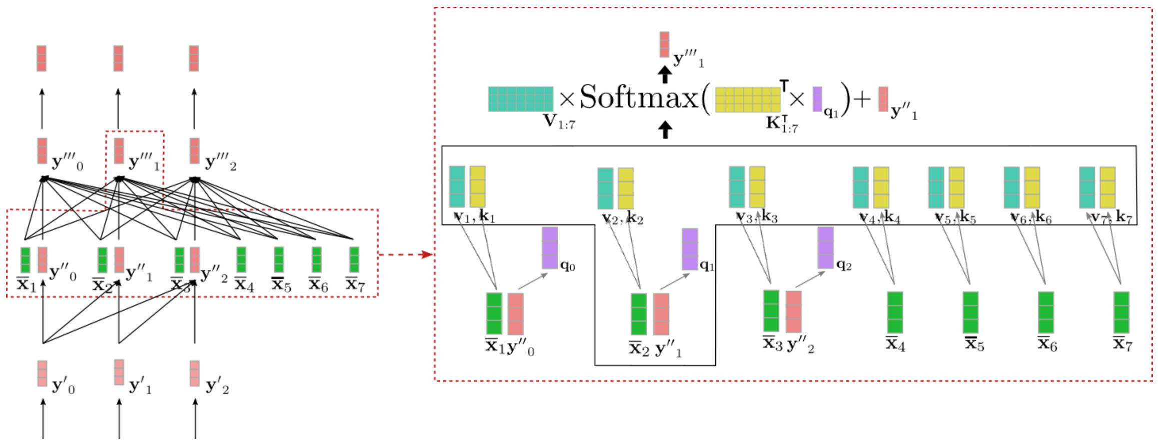
Great! Now we can move to the layer that connects the encoder and decoder - the *cross-attention* mechanism!

The cross-attention layer takes two vector sequences as inputs: the outputs of the uni-directional self-attention layer, *i.e.* $\mathbf{Y}''_{0:m-1}$ and the contextualized encoding vectors $\overline{\mathbf{X}}_{1:n}$. As in the self-attention layer, the query vectors $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$ are projections of the output vectors of the previous layer, *i.e.* $\mathbf{Y}''_{0:m-1}$. However, the key and value vectors $\mathbf{k}_0, \ldots, \mathbf{k}_{m-1}$ and $\mathbf{v}_0, \ldots, \mathbf{v}_{m-1}$ are projections of the contextualized encoding vectors $\overline{\mathbf{X}}_{1:n}$. Having defined key, value, and query vectors, a query vector $\mathbf{q}_i$ is then compared to *all* key vectors and the corresponding score is used to weight the respective value vectors, just as is the case for *bi-directional* self-attention to give the output vector $\mathbf{y}'''_i$ for all $i \in {0, \ldots, m-1}. Cross-attention can be summarized as follows:

$$\mathbf{y}'''_i = \mathbf{V}_{1:n}\mathbf{Softmax}(\mathbf{K}_{1:n}^\intercal \mathbf{q}_i) + \mathbf{y}''_i.$$

Note that the index range of the key and value vectors is $1 : n$ corresponding to the number of contextualized encoding vectors.

Let's visualize the cross-attention mechanism Let's for the input vector $\mathbf{y}''_1$ for our example above.


$\mathbf{y}'''_0$
$\mathbf{y}'''_1$
$\mathbf{y}'''_2$
$\overline{\mathbf{x}}_1$
$\mathbf{y}''_0$
$\overline{\mathbf{x}}_2$
$\mathbf{y}''_1$
$\overline{\mathbf{x}}_3$
$\mathbf{y}''_2$
$\overline{\mathbf{x}}_4$
$\overline{\mathbf{x}}_5$
$\overline{\mathbf{x}}_6$
$\overline{\mathbf{x}}_7$
$\mathbf{y}'_0$
$\mathbf{y}'_1$
$\mathbf{y}'_2$
$\times$Softmax($\;^\intercal\times\;$)+
$\mathbf{V}_{1:7}$
$\mathbf{K}_{1:7}^\intercal$
$\mathbf{q}_1$
$\mathbf{v}_1, \mathbf{k}_1$
$\mathbf{v}_2, \mathbf{k}_2$
$\mathbf{v}_3, \mathbf{k}_3$
$\mathbf{v}_4, \mathbf{k}_4$
$\mathbf{v}_5, \mathbf{k}_5$
$\mathbf{v}_6, \mathbf{k}_6$
$\mathbf{v}_7, \mathbf{k}_7$
$\mathbf{q}_0$
$\mathbf{q}_2$

We can see that the query vector $\mathbf{q}_1$ (shown in purple) is derived from $\mathbf{y}''_1$ and therefore relies on a vector representation of the word "Ich". The query vector $\mathbf{q}_1$ (shown in red) is then compared to the key vectors $\mathbf{k}_1, \ldots, \mathbf{k}_7$ (shown in yellow) corresponding to the contextual encoding representation of all encoder input vectors $\mathbf{X}_{1:n}$ = "I want to buy a car EOS". This puts the vector representation of "Ich" into direct relation with all encoder input vectors. Finally, the attention weights are multiplied by the value vectors $\mathbf{v}_1, \ldots, \mathbf{v}_7$ (shown in turquoise) to yield in addition to the input vector $\mathbf{y}''_1$ the output vector $\mathbf{y}'''_1$ (shown in dark red).

So intuitively, what happens here exactly? Each output vector $\mathbf{y}'_i$ is a weighted sum of all value projections of the encoder inputs $\mathbf{v}_1, \ldots, \mathbf{v}_7$ plus the input vector itself $\mathbf{y}_i$ (*c.f.* illustrated formula above). The key mechanism to understand is the following: Depending on how similar a query projection of the *input decoder vector* $\mathbf{q}_i$ is to a key projection of the *encoder input vector* $\mathbf{k}_j$, the more important is the value projection of the encoder input vector $\mathbf{v}_j$. In loose terms this means, the more "related" a decoder input representation is to an encoder input representation, the more does the input representation influence the decoder output representation.

Cool! Now we can see how this architecture nicely conditions each output vector $\mathbf{y}'''_i$ on the interaction between the encoder input vectors $\overline{\mathbf{X}}_{1:n}$ and the input vector $\mathbf{y}''_i$. Another important observation at this point is that the architecture is completely independent of the number n of contextualized encoding vectors $\overline{\mathbf{X}}_{1:n}$ on which the output vector $\mathbf{y}'''_i$ is conditioned on. All projection matrices $\mathbf{W}^{\text{cross}}_{k}$ and $\mathbf{W}^{\text{cross}}_{v}$ to derive the key vectors $\mathbf{k}_1, \ldots, \mathbf{k}_n$ and the value vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ respectively are shared across all positions $1, \ldots, n$ and all value vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ are summed together to a single weighted averaged vector. Now it becomes obvious as well, why the transformer-based decoder does not suffer from the long-range dependency problem, the RNN-based decoder suffers from. Because each decoder logit vector is *directly* dependent on every single encoded output vector, the number of mathematical operations to compare the first encoded output vector and the last decoder logit vector amounts essentially to just one.

To conclude, the uni-directional self-attention layer is responsible for conditioning each output vector on all previous decoder input vectors and the current input vector and the cross-attention layer is responsible to further condition each output vector on all encoded input vectors.

To verify our theoretical understanding, let's continue our code example from the encoder section above.

---

[1] The word embedding matrix $\mathbf{W}_{\text{emb}}$ gives each input word a unique *context-independent* vector representation. This matrix is often fixed as the "LM Head" layer. However, the "LM Head" layer can very well consist of a completely independent "encoded vector-to-logit" weight mapping.

[2] Again, an in-detail explanation of the role the feed-forward layers play in transformer-based models is out-of-scope for this notebook. It is argued in Yun et. al, (2017) that feed-forward layers are crucial to map each contextual vector $\mathbf{x'}_i$ individually to the desired output space, which the *self-attention* layer does not manage to do on its own. It should be noted here, that each output token $\mathbf{x'}$ is processed by the same feed-forward layer. For more detail, the reader is advised to read the paper.