Attention

Attention is the ability to *focus on different parts of the input*, according to the requirements of the problem being solved. It is an essential component of any intelligent behaviour, with potential application to a wide range of domains.

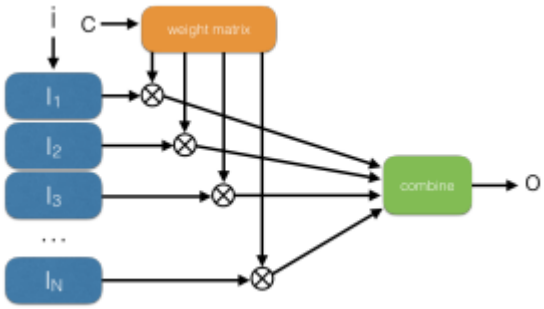
Usually, attention constist of either cropping or *masking* all the parts that are not important. *Masking especially is what is used*, rather than cropping.

A differential mechanism

From the point of view of Neural Networks, we would expect the attention mechanism to be *differentiable*, so that we can learn where to focus by *standard backpropagation techniques*. The current approach (not necessarily the best one) is to focus everywhere, just *to different extents*.

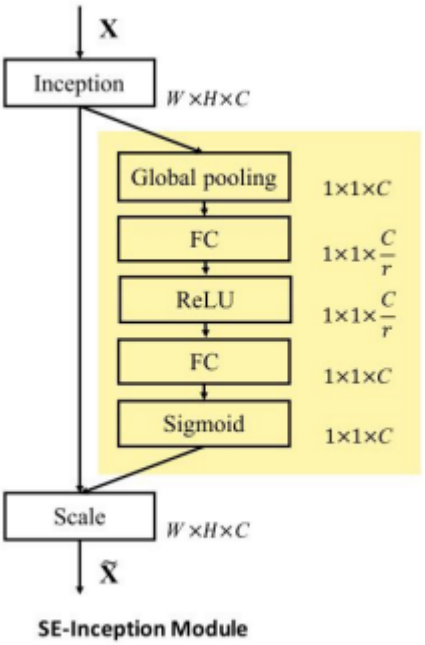
Attention as gating maps

Attention mechanisms can be implemented as *gating functions*. The gating maps are *dynamically generated by some neural net*, allowing to focus on different part on the input at different times. ==The forget map, input map and output map in [[2023-05-02 - Recurrent Neural Networks#Long-Short Term Memory (LSTM)|LSTMs) are examples of attention mechanisms==.



Another example: squeeze and excitation

SE layers are a building component of **Squeeze and Excitation Networks**. *SE layers* implement a form of self attention, allowing to *focus on particular channels in a* ***dinamical*** *way*, according to the input under consideration.



SE-Inception Module

The SE-inception module essentially does this: it reduces the number of channels, and then generates a boolean map for each one of the regional channels. This allows only to focus on the maps that we care about.

Key-value attention

The most typical attention layer is based on the key-value paradigm, implementing a sort of **associative memory** (i.e. a dictionary). We access this memory with queries to be matched with keys. The resulting *scores* generate *a boolean map that is used to weight values*.

In this approach, we have *key-value* pairs and a query. We need to find which key is matches the query. Since we want to compare 2 vectors, some typicals metrics that we can use are *cross-similarity*. So this is the general process:

- For each key k_i compute the *scores* a_i as:

  $$a_i = \alpha(1, k_i)$$

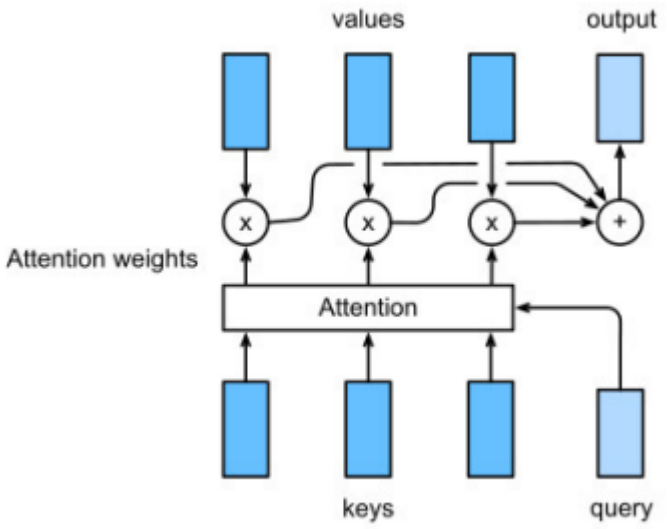
  (the α is the score function.)
- then, obtain *attention weights* via *softmax*:

  $$\vec{b} = softmax(\vec{a})$$

- return a *weighted sum* of the values:

  $$o = \sum_{i=1}^{n} b_i v_i$$

In many applications, *values* are also used *as keys* (**self-attention**).



From Tensorflow official explanation: An attention layer does a fuzzy lookup like this, but it's not just looking for the best key. *It combines* the `values` based *on how well the* `query` matches each `key`.

How does that work? In an attention layer the `query`, `key`, and `value` are each vectors. Instead of doing a hash lookup the attention layer combines the `query` and `key` vectors to determine how well they match, the "**attention score**". The layer returns *the average across all the* `values`, *weighted* by the "attention scores".

Each location the query-sequence provides a `query` vector. The context sequence acts as the dictionary. At each location in the context sequence provides a `key` and `value` vector. The input vectors are not used directly, the `layers.MultiHeadAttention` layer includes `layers.Dense` layers to project the input vectors before using the

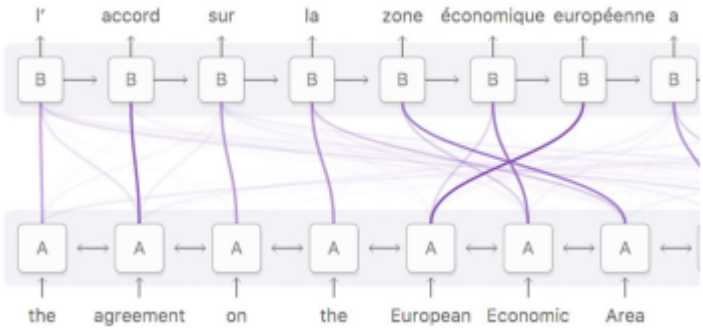
Score function Different score functions lead to different attention layers. Two commonly used approaches are:

- **Dot product**:
$$\alpha(q, k) = q \cdot k\sqrt{d}$$
The query and the key must have the *same dimension* d.
- **MLP**: α is computed by a *neural network* (usually composed by a single layer):
$$\alpha(k, q) = tanh(W_k\vec{k} + W_q\vec{q})$$

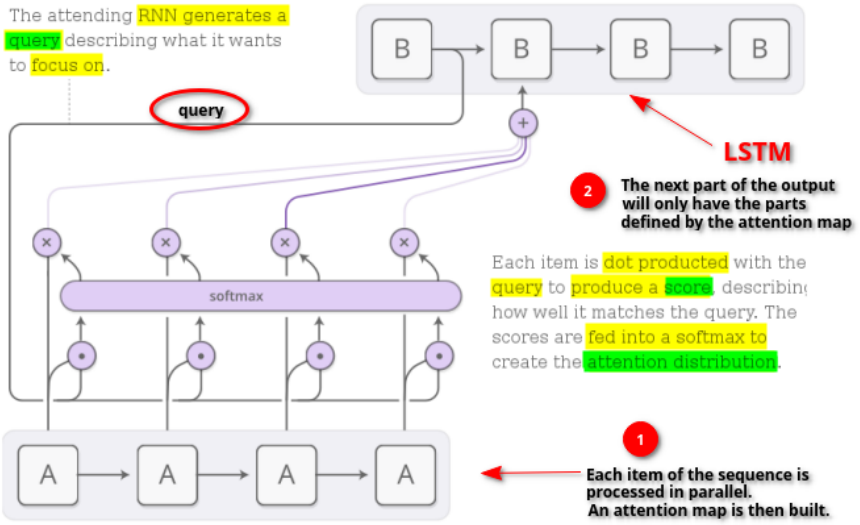
Attention applied to translations

First, here's some terms that you need to know:

- **alignment**: identify which parts of the *input sequence* are **__relevant** to each word in the output__.
- translation: y'all already know; is the process of using the relevant information to select the appropriate output.



Alignement is a form of attention!



Here's how alignment works:

Transformers

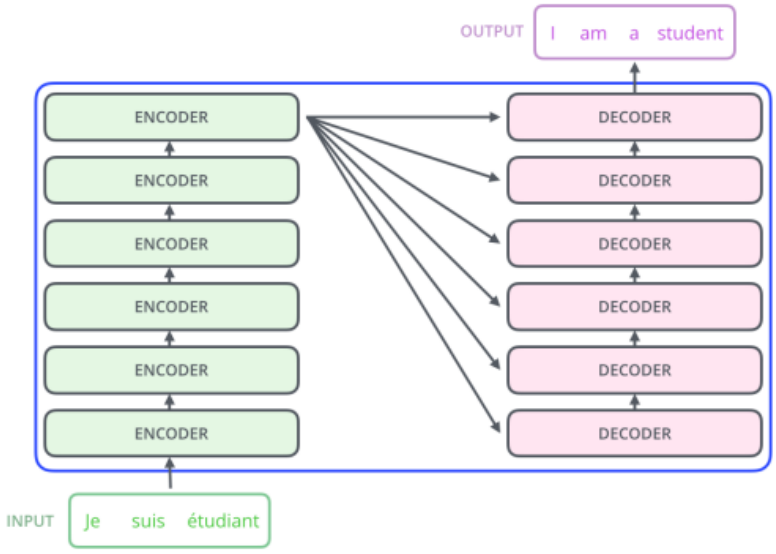
Transformers have been introduced in Attention is All You Need, one of the most influential works of recent years. This works stated that the LSTM where not needed anymore to process text, but rather, it was possible to achieve even better result by simply using attention and feed-forward networks. **Transformers** have rapidly become the *model of choice for NLP*. Applications like Bert and

GPT (General Pre-trained Trainsformer), (with all relative families) are based on Transformers.

General structure of transformers

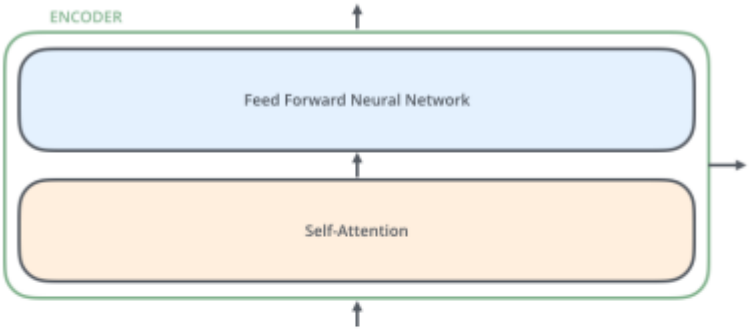
Encoder & decoder A transformer has a traditional *encoder-decoder structure*, with connections between them. The **encoding component** is a stack of *encoders*. Similarly, the **decoding component** is a stack of *decoders*.

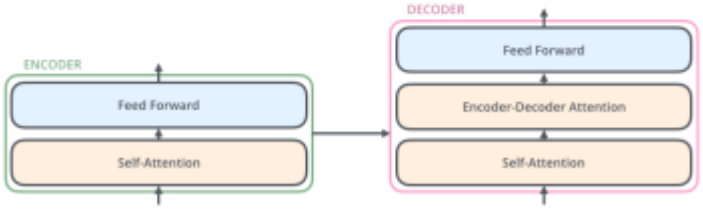
- The encoder is supposed to extract information about the input (i.e. semantics), and is used to condition the generation of the output made by the decoder (typically, is just the last encoder of the stack that condition all the elements of the decoder).



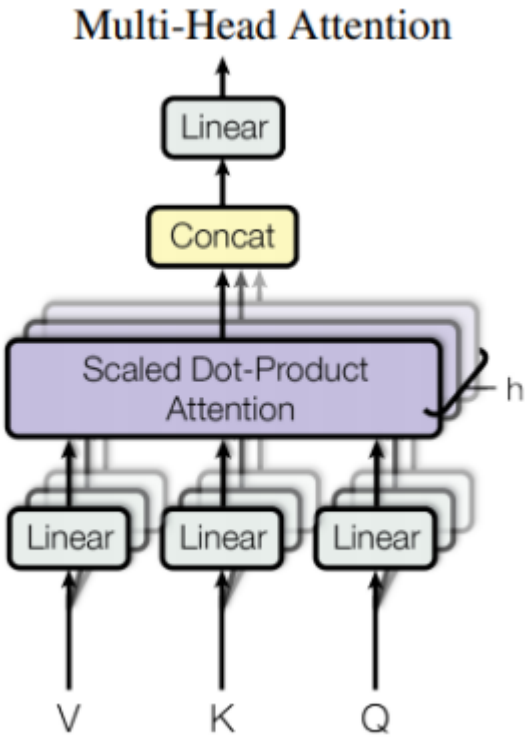
Structure of encoder and decoder modules

- The **encoder** is organized as a **__self-attention** layer* (query, key and value are shared), followed by _feedfoward component* (a couple of dense layers).
  - Each output is obtain as a weighted combination of the input.
- The **decoder** is *similar*, with *an additional attention layer* that helps the decoder to *focus on relevant parts* of the input sentence (meaning, according to the information passed on by the decoder).





Multi-head attention Using *multiple heads* for *attention* expands the model's ability to focus on *different positions*, for different purposes. As a result, multiple "representation subspaces" are created, focusing on potentially different aspects of the input sequence.

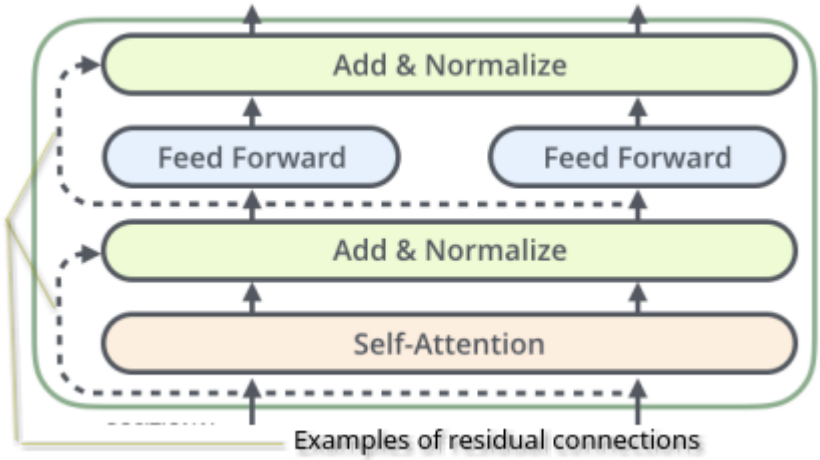


- This focus on different positions happens in parallel.

Masking the future The process of masking in transformers is used in the *decoder part*. Essentially, we can apply a boolean mask to the input, to hide part of its content. This is frequently used in the decoder to prevent it to attend at future positions during generation.

- In general, it prevents confusion inside the NN (i.e. some tokens are only the initial part of a phrase that is not relevant to the generation of text etc…)

Residual connections Each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and it is followed by a layernormalization

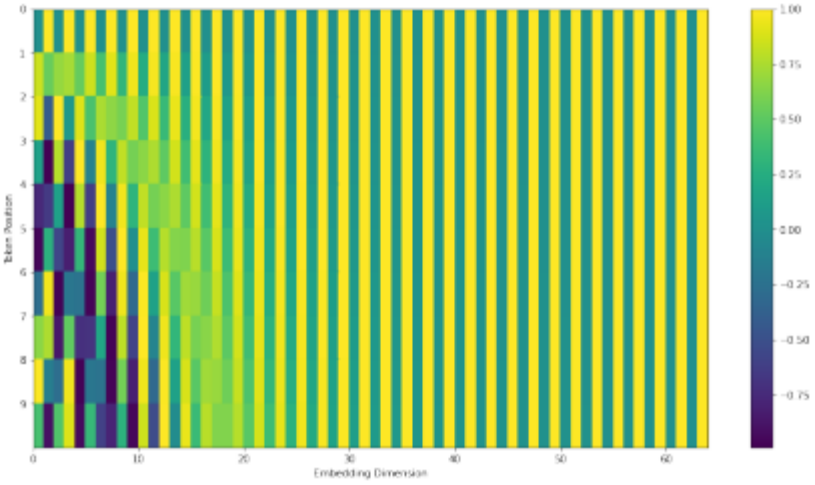


step.
Why? who knows. Not me for sure, since neither asperti nor the slides mention it.

Positional encoding **Positional encoding** is added to word embeddings to give the model some information about the *relative position* of the *words in the sentence*. The positional information is a vector of the same dimensions d_{model}, of the word embedding. The authors use *sine* and *cosine* functions of different frequencies.

- It provides some structural information in relation to a relative position of phrase in a certain language (i.e. i should always be more focused to 2 tokens on the right of the current token etc...)

Each important position is then encoded as a *sequence of frequencies*, that form some barcode like structure.



This

type of representation is also used since it can be expressed as a linear mapping, which is easily learn by a linear perceptron.