Implementing attention: High level view

To state the problem of Attention more abstractly as follows

Given

- Source sequence $\bar{c}_{([1:\bar{T}])}$
    - the sequence being "attended to"
    - a sequence of source "contexts"
- and a Target context $c_{(t)}$
    - called the "query"

Output

- the Source context $\bar{c}_{(\bar{t})}$
- that most closely matches the desired Target context $c_{(t)}$

For example, let's consider Cross Attention in an Encoder-Decoder architecture

- $\bar{c}_{([1:\bar{T}])}$ may be the sequence of latent states of an Encoder
- "query" $c_{(t)} = \mathbf{h}_{(t)}$ is the state of the Decoder when generating output $\hat{\mathbf{y}}_{(t)}$ at position t
- we want to output $\bar{c}_{(\bar{t})}$: one latent state of the Encoder
    - relevant for output position t
    - as described by $c_{(t)} = \mathbf{h}_{(t)}$

The mechanism we use to match Target and Source contexts is called *Context Sensitive Memory* which we introduced in a previous module (Neural_Programming.ipynb#Soft-Lookup)

To recap:

- Context Sensitive Memory is similar to a Python `dict`
    - consists of a collection of Key/Value pairs
- One may perform a "lookup"
    - By presenting a "query"
    - Which matches the query against each key

In a Python `dict`

- the lookup fails if the query does not match any key *exactly*

In a Context Sensitive Memory

- each key k is matched against the query q
- the *similarity* of k and q is measured by a score

$$\text{score}(q, k)$$

- the output is a *weighted sum* of the values of the CSM
    - with weights equal to the score

This is called a *soft lookup*.

Thus, a lookup in a CSM

- always produces a result
- that is the weighted sum of *all* values
- special case: Python `dict` behavior occurs where the weight is 100% for a single key

Let the collection of key/value pairs be M

$$M = \{(k_{\bar{t}}, v_{\bar{t}}) \,|\, 1 \leq \bar{t} \leq \bar{T}\}$$

Soft lookup of query q outputs

$$\text{lookup}(q, M) = \sum_{(k,v) \in M} \alpha(q, k) * v$$

- the weighted sum (across the value component of each key/value pair)
- where the weight $\alpha(q, k)$ of the value with key k is computed via a Softmax on the value $\text{score}(q, k)$

$$\alpha(q, k) = \frac{\exp(\text{score}(q, k))}{\sum_{k' \in \text{keys}(M)} \exp(\text{score}(q, k')}$$

- where $\text{score}(q, k)$ is a measure of the similarity of query q with key k
    - a very common score is the dot product (cosine similarity)
- the Softmax normalizes the scores to sum to 100%

Attention lookup: detailed view

In general the keys, values and queries could be generated by arbitrary parts of a larger Neural Network that uses Attention.

In the case of an Encoder-Decoder architecture, a typical arrangement

- keys and values are identical
    - each are vectors of length d
    - there is one per position in Encoder (i.e., $\bar{T}$)
- the query is created by the output of an earlier layer of the Decoder
    - each is a vector of length d
    - there is one per position of the Decoder (i.e., T)

The Decoder generates a query (per position) which returns a (weighted sum of) latent state(s) of the Encoder.

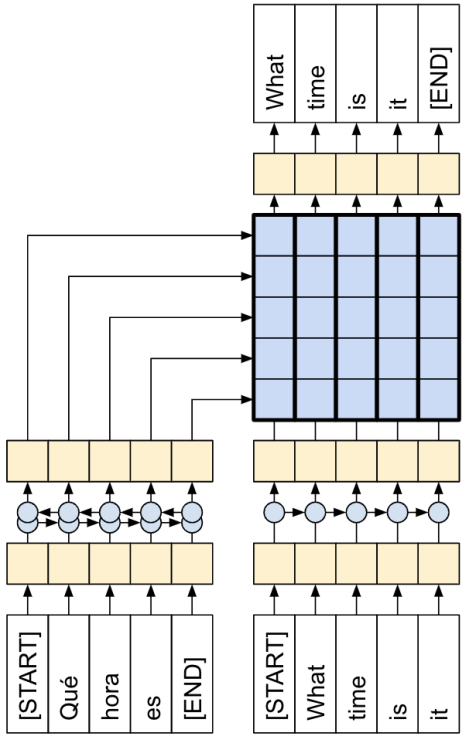
Here is an illustration of the Attention inputs of the Encoder Decoder.

Here is a picture of the complete RNN Encoder Decoder designed to translate Spanish to English

Both the Encoder and Decoder are RNN's.

- Encoder: left side (bottom to top)
    - bottom row: sequence of token ids of Spanish language input
    - middle row: an unrolled, bidirectional RNN computation
        - computing an encoding (latent representation) for each of the $\bar{T}$ Spanish tokens
    - top row: sequence of latent representations of Spanish tokens
        - used as keys/values for Attention
- Decoder: similar to Encoder
    - top row: latent representation of generated English token ids
        - used as queries for Attention

RNN Encoder-Decoder for Spanish to English translation



Attribution: https://www.tensorflow.org/text/tutorials/nmt_with_attention

We will give the details through an example that illustrates the Self Attention lookup behavior of the Transformer.

Aside

- we may not yet have covered the Transformer
- just know that the Decoder uses both
    - Masked Causal Self-Attention on its inputs
    - Cross Attention between the Decoder and the Encoder

In the Transformer use of Self-Attention

- keys, values and queries
- are identical !

The Transformer has a context for each of the T positions in the input sequence.

We represent this as a matrix $\mathbf{X}$ of dimension $(T \times d)$

- where $d = d_{\text{model}}$ is the internal dimension of all vectors

That is

- the Transformer has a source context for each position
- which it uses as a query to "look up" the most similar context

We can potentially increase the power of the Transformer

- my mapping the keys, values and queries
- through
- producing alternate representation
    - key $\mathbf{x} \mapsto \mathbf{x}W_K$
    - value $\mathbf{x} \mapsto \mathbf{x}W_V$
    - query $\mathbf{x} \mapsto \mathbf{x}W_Q$
- that may better be adapted to the task described by the training data

Embedding matrices W_K, W_V, W_Q are *learned* through training

- if no better representation exists: we presumably learned identity matrices
- the embedding matrices can also reduce all vectors to length $d_{\text{attn}} = \frac{d}{n}$
    - to facilitate multi-head attention with n heads

Multiple lookups can be performed in parallel via matrix multiplication

- when the score measuring the similarity of key k and query q is the dot product

In the case of the Transformer

- where keys, values and queries are identical
- and a lookup is performed for each of the T positions

we use matrix multiplication of matrix $\mathbf{X}$

We keep track of the matrix sizes below (assuming $d_{\text{attn}} \leq d$)

First: we map all vectors through the embedding matrices

out		left		right
Q	=	$\mathbf{X}$	*	$\mathbf{W}_Q$
K	=	$\mathbf{X}$	*	$\mathbf{W}_K$
V	=	$\mathbf{X}$	*	$\mathbf{W}_V$
$(T \times d)$		$(T \times d)$		$(d \times d)$

Next: comparing the query q at each positions, to all of the keys

- producing scores $\alpha(q, k)$ that are implemented as dot product (matrix multiplication)

out		left		right
$\alpha(q, k)$	=	Q	*	K^T
$(T \times T)$		$(T \times d)$		$(d \times T)$

- we ignore the softmax normalization of the weights

Finally: multiply the weights by the values

out		left		right
	=	$\alpha(q, k)$	*	V
$(T \times d)$		$(T \times T)$		$(T \times d)$

producing

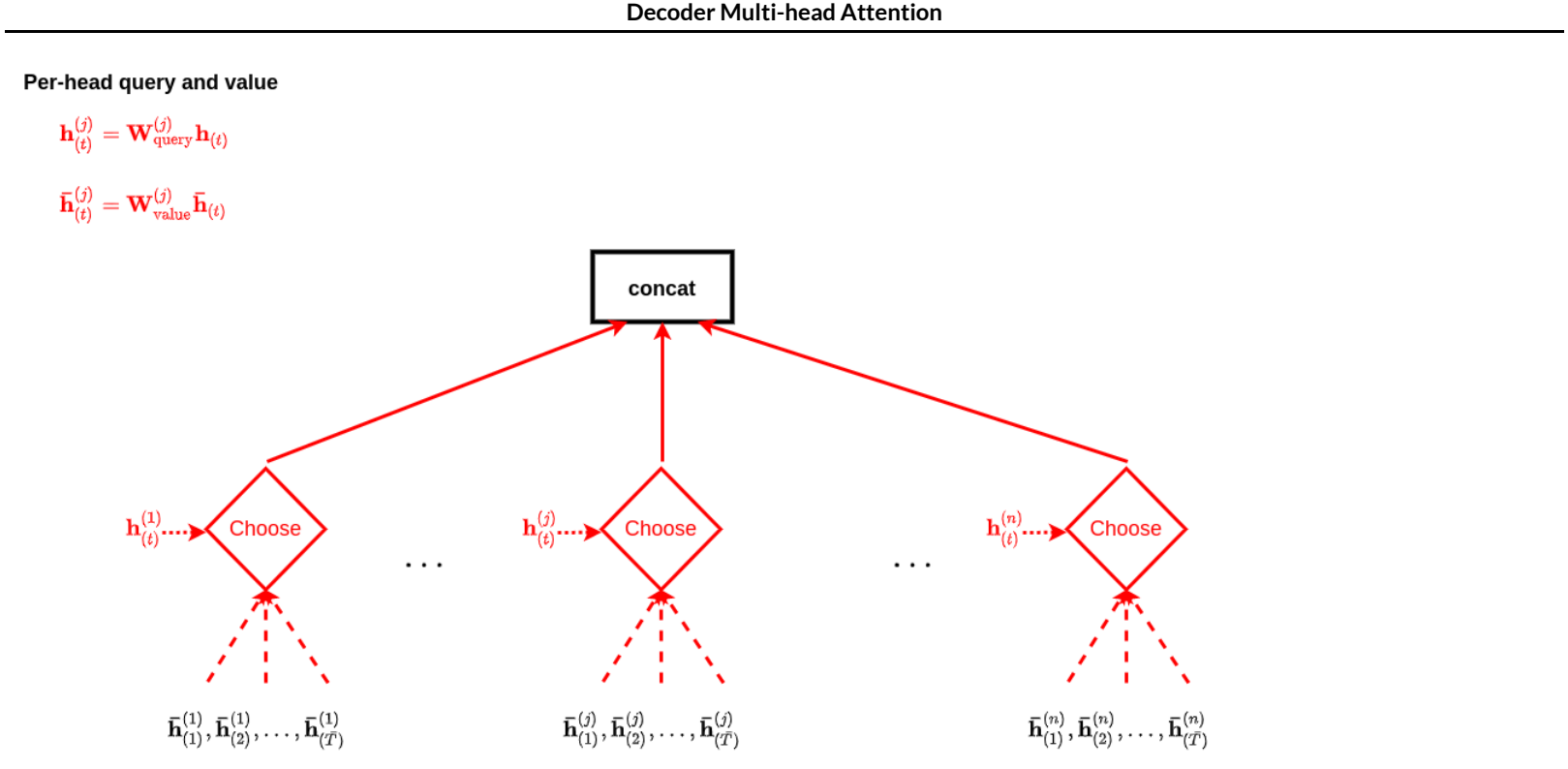
- a single attention value of length d
- for each of the T positions

Multi-head attention

The picture shows n Attention heads.

Note that each head is working on vectors of length $d_{\text{attn}} = \frac{d}{n}$ rather than original dimensions d.

- variables with superscript (j) are of fractional length



How do we create the shorter length $\frac{d}{n}$ vectors ?

We use projection matrices of size $(d \times \frac{d}{n})$ **for each head** j

- multiplying each key by matrix $\mathbf{W}_{\text{key}}^{(j)}$
- multiplying each value by matrix $\mathbf{W}_{\text{value}}^{(j)}$
- multiplying the original length d query by matrix $\mathbf{W}_{\text{query}}^{(j)}$

Head j

- uses query $\mathbf{h}^{(j)} = \mathbf{h} * \mathbf{W}_{\text{query}}^{(j)}$
- against keys/values $\bar{\mathbf{h}}^{(j)} = \bar{\mathbf{h}} * \mathbf{W}_{\text{value}}^{(j)}$

Advanced material

The remaining sections include code references to models constructed using the Functional API of Keras.

Even if you don't understand the code in detail, the intuition it conveys may be useful.

Code: RNN Encoder-Decoder

The code for the Spanish to English Encoder Decoder can be found in a [TensorFlow tutorial (https://www.tensorflow.org/text/tutorials/nmt_with_attention)]

- requires knowledge of Functional models in Keras
- Multi-head Attention implemented by a Keras layer
    - code not visible directly
    - but is a link to source on Githb
        - a bit complex since it is production code
- Colab notebook you can play with
    - substitute your own Spanish sentences as input
    - make Attention plots

A good web post on implementing MultiHead Attention can be found here (https://machinelearningmastery.com/how-to-implement-multi-head-attention-from-scratch-in-tensorflow-and-keras/)

- rather than using $(d_{\text{model}} \times d_{\text{attn}})$ embedding matrices to project vectors from d_{model} to d_{attn}
- it uses `Dense` layers with d_{attn} units to achieve the same
- multi-head attention is achieved by *reshaping* the input
    - from 3D shape $(\text{batch\_size} \times T \times d_{\text{model}})$
    - to 4D shape $(\text{batch\_size} \times T \times n_{\text{head}} \times d_{\text{attn}})$
        - where d_{model} should be equal to $n_{\text{head}} * d_{\text{attn}}$

Here is a [Keras tutorial (https://keras.io/examples/nlp/neural_machine_translation_with_transformer/)](https://keras.io/examples/nlp/neural_machine_translation_with_transformer/) that uses an Encoder and Decoder that are both Transformers

- Self attention on the Decoder
- Cross attention from the Decoder to the Encoder

Here is the relevant code for the Decoder

```
def call(self, inputs, encoder_outputs, mask=None):
        causal_mask = self.get_causal_attention_mask(inputs)
        if mask is not None:
            padding_mask = tf.cast(mask[:, tf.newaxis, :], dtype="int32")
            padding_mask = tf.minimum(padding_mask, causal_mask)

        attention_output_1 = self.attention_1(
            query=inputs, value=inputs, key=inputs, attention_mask=causal_mask
        )
        out_1 = self.layernorm_1(inputs + attention_output_1)

        attention_output_2 = self.attention_2(
            query=out_1,
            value=encoder_outputs,
            key=encoder_outputs,
            attention_mask=padding_mask,
        )
        out_2 = self.layernorm_2(out_1 + attention_output_2)

        proj_output = self.dense_proj(out_2)
```

- The Decoder input (partially generated English Translation)
    - Masked Self Attention on the input via the statement

```
  attention_output_1 = self.attention_1(
          query=inputs, value=inputs, key=inputs, attention_mask
=causal_mask
      )
```

        - keys = values = queries = inputs
        - **causal masked**: via the option `attention_mask=causal_mask`

    - uses Cross attention via the statement

      attention_output_2 = self.attention_2(

```
      query=out_1,
      value=encoder_outputs,
      key=encoder_outputs,
      attention_mask=padding_mask,
  )
```

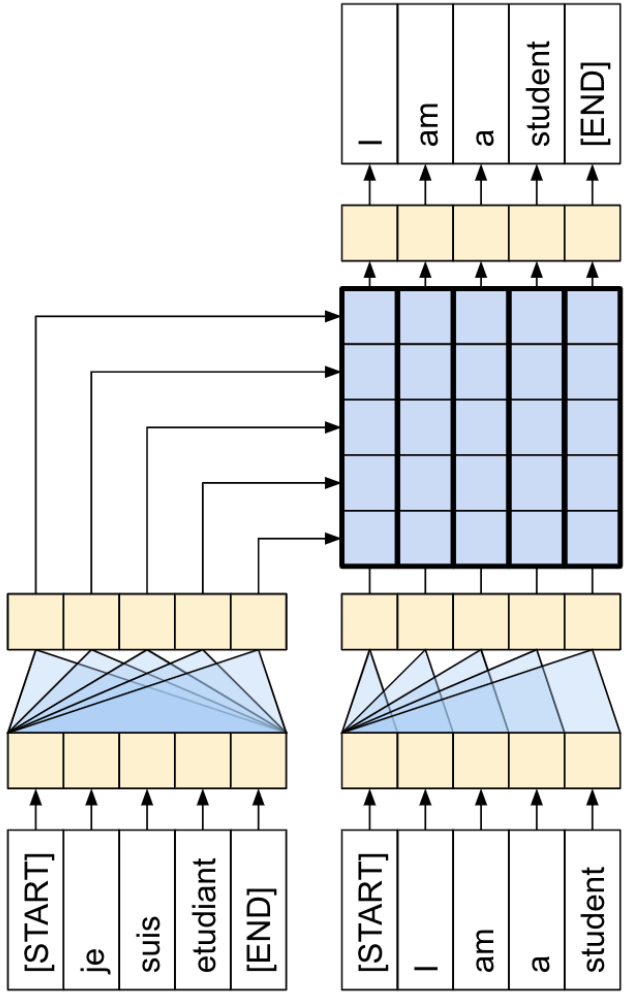
        - query is output of the Self-Attention
            - the query is created by self-attention of Decoder input
        - keys = values = `encoder_outputs` (sequence of Encoder latent states)

Code: Encoder-Decoder Transformer

Here is the Encoder-Decoder for Spanish to English Translation, using Transformers for both the Encoder and Decoder

- Encoder: left-side
    - Bottom row: Encoder Spanish Tokens
    - Top row: Self-Attention to Spanish tokens
- Decoder: right side
    - Bottom row: latent representation of English tokens generated so far
    - Next row: Decoder Masked Self Attention
- Matrix: column t
    - Attention weight of Decoder output at position t on each of the $\bar{T}$ latent representation of the Encoder's Spanish tokens

Transformer Encoder-Decoder for Spanish to English translation



Attribution: https://www.tensorflow.org/images/tutorials/transformer/Transformer-1layer-words.png

Conclusion

We introduced Context Sensitive Memory as the vehicle with which to implement the Attention mechanism.

Context Sensitive Memory is similar to a Python dict/hash, but allowing "soft" matching.

It is easily built using the basic building blocks of Neural Networks, like Fully Connected layers.

This is another concrete example of Neural Programming.

In [2]:
```
print("Done")
```

```
Done
```