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*.

Summary

- 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
- The result is a "soft" lookup
    - always returns a value, even if there is no exact match between the query and any key
    - the results is a weighted sum of the values in the key/value pairs
    - with weights based on the similarity of the query and the key

Let's see how [Context Sensitive Memory (Context_Sensitive_Memory.ipynb)](Context_Sensitive_Memory.ipynb) works.

Cross-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 the Attention is between

- queries created by the Decoder
- keys and values created by the Encoder
    - keys and values are identical

We use a Context Sensitive Memory to implement the Attention lookup.

The CSM has $\bar{T}$ key/value pairs

- the key and value for row $\bar{t}$ of the CSM is state $\bar{\mathbf{h}}_{(t)}$

$$k_{\bar{t}} = v_{\bar{t}} = \bar{\mathbf{h}}_{(\bar{t})}$$

The Decoder creates one query for each of the T positions of the Decoder output

- the query for position t is Decoder state $\mathbf{h}_{(t)}$

$$q_t = \mathbf{h}_{(t)}$$

Thus, each position of the Decoder

- attends to all positions of the Encoder
- using Decoder state $\mathbf{h}_{(t)}$ as the query for output position t

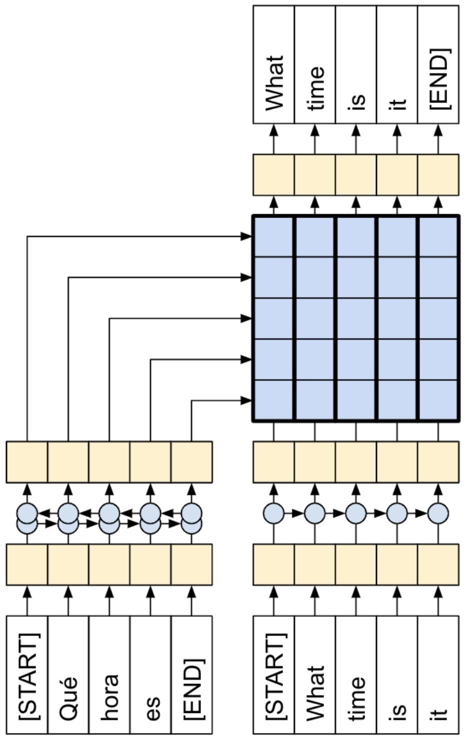
Here is an illustration of the Attention inputs of the Encoder Decoder.

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

Attention Lookup: general case

We assume that

- the Source context (the sequence being attended to) is length $\bar{T}$
    - e.g., Encoder states $\bar{\mathbf{h}}_{(t)}$ in an Encoder/Decoder
- the Target context is length T
    - e.g., Decoder states $\mathbf{h}_{(t)}$ in an Encoder/Decoder

All vectors ($\mathbf{h}$, $\bar{\mathbf{h}}$) are length d

This describes Cross-Attention as would be implemented from the Decoder to the Encoder in an Encoder-Decoder architecture.

For the special case of Self-Attention:

- $\bar{T} = T$
- $\bar{\mathbf{h}}_{(t)} = \mathbf{h}_{(t)}$

This is the case, for example, where a Decoder attends to itself.

Queries

Each of the T Target positions is a query

$$q_{(t)} = h_{(t)}$$

So the matrix Q of all queries is shape $(T \times d)$

Keys/Values

Each of the $\bar{T}$ Source positions is both a target and a query

$$k_t = \mathbf{v}_t = \bar{\mathbf{h}}_{(t)}$$

The matrix of all keys K, and the matrix of all values V are shape $(\bar{T} \times d)$

Pre-processing queries, keys and values

Rather than using the raw states of the Source and Target as queries (resp., keys/values)

- we can map them through *matrices* $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$
  - each mapping matrix shape is $(d \times d)$
  - thus, the mapping preserves the shapes of Q, K, V

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

This mapping potentially increases the power of a Transformer that uses Attention

- if no better representation exists: we presumably learned identity matrices

Mapping through these matrices:

out		left		right
Q	=	Q	*	$\mathbf{W}_Q$
$(T \times d)$		$(T \times d)$		$(d \times d)$

K | = | K | | $\boldsymbol{W}_K$ | V | = | V | | $\mathbf{W}_V$ | $(\bar{T} \times d)$ | | $(\bar{T} \times d)$ | | $(d \times d)$

Multi-head attention

By changing the mapping matrices shape to $(d \times d_{\text{attn}})$

- where d_{attn}
  $$= \frac{d}{n_{\text{head}}}$$

we can reduce the size of all mapped vectors (i.e., each query, key, value) from d to d_{attn}

One reason for doing this is *multi-head* attention

- Each target position can attend to n_{head} different (and shorter) source state representations
- Rather than a single source position of full length d

Think of this as the Target attending to a vector of Source positions

- each representing a different "feature" of the Source that is relevant to the Target

We want all internal vectors to have lengths consistently be equal to d

- we can concatenate the n_{head} shorter (length d_{attn}
- into a single attention output of length d

Performing the lookup

Next: comparing the query q at each Target position, to each of the keys at the $\bar{T}$ Source positions

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

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

- we ignore the softmax normalization of the weights
- we will treat the scores as weights for simplicity of presentation

Finally: take the weighted sum of the values

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

producing

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

Conclusion

Using matrix operations, we are performing *all* T queries simultaneously.

The end result is a vector of length d

- the value being attended to at each of the T Target positions
- this value is a weighted sum of the $\bar{T}$ Source states

Multi-head attention in detail

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

Decoder Multi-head Attention

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

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)](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/)](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

```python
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,
    )
```

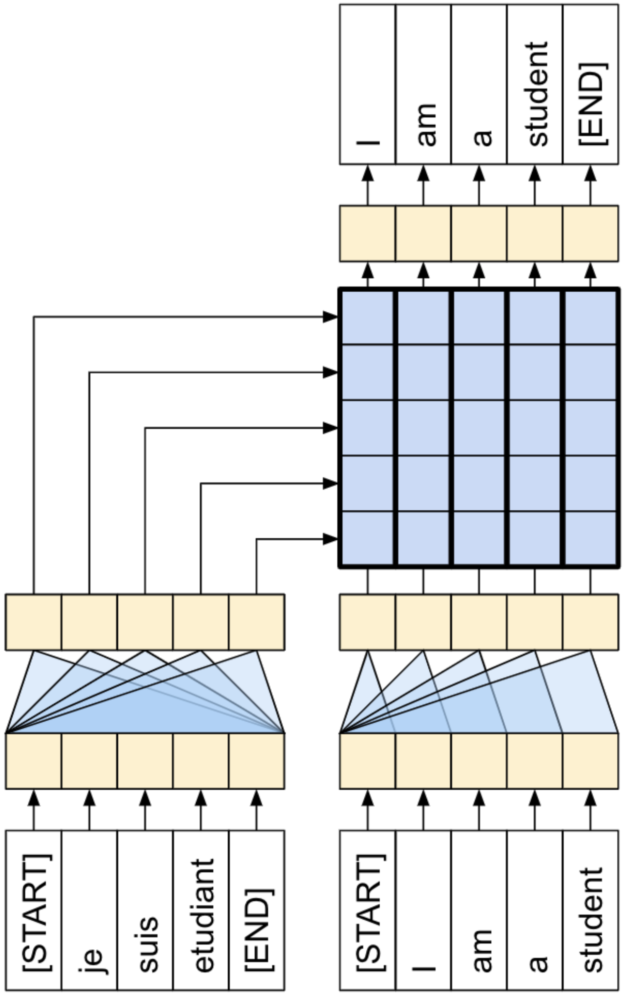
        - query is output of the Self-Attention
            - the query is created by self-attention of Decoder input
        - keys = values = `encoder_outputs` (sequence of Encoder latent states)

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
```