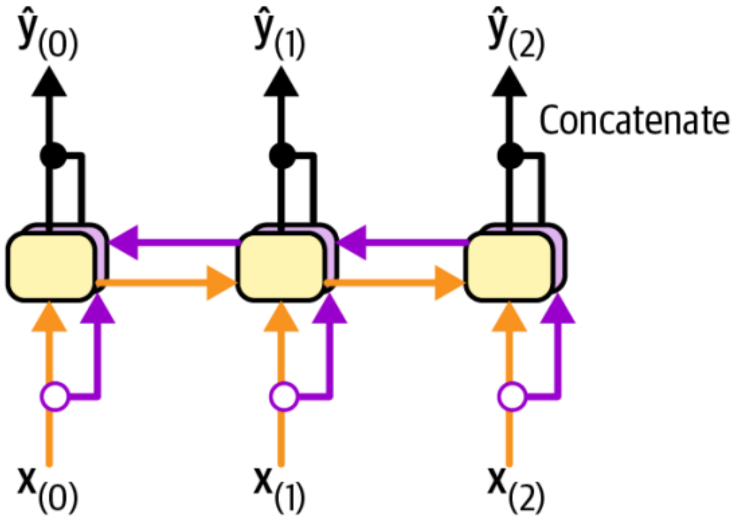
7.
Bidirectional RNNs

Regular Recurrent Layer

- A regular recurrent layer only looks at past and present inputs before generating its output.
  - It cannot look into the future.
- This type of RNN makes sense when forecasting time series, or in the decoder of a sequence-to-sequence (seq2seq) model.
- For tasks like text classification, or in the encoder of a seq2seq model, it is preferable to look ahead at the next words before encoding a given word.
- For example, consider the phrases “the right arm”, “the right person”, and “the right to criticize”: to properly encode the word “right”, you need to look ahead.

Bidirectional Recurrent Layer

- *Bidirectional recurrent layer:* run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left, then combine their outputs at each time step, typically by concatenating them.



Bidirectional RNN in Keras

- To implement a bidirectional recurrent layer in Keras, just wrap a recurrent layer in a `tf.keras.layers.Bidirectional` layer:

```
encoder = tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(256, return_state=True))
```

- The Bidirectional layer will create a clone of the `GRU` layer (but in the reverse direction), and it will run both and concatenate their outputs.
  - So although the `GRU` layer has 256 units, the `Bidirectional` layer will output 512 values per time step.
- This layer will now return four states instead of two: the final short-term and long-term states of the forward `LSTM` layer, and the final short-term and long-term states of the backward `LSTM` layer.

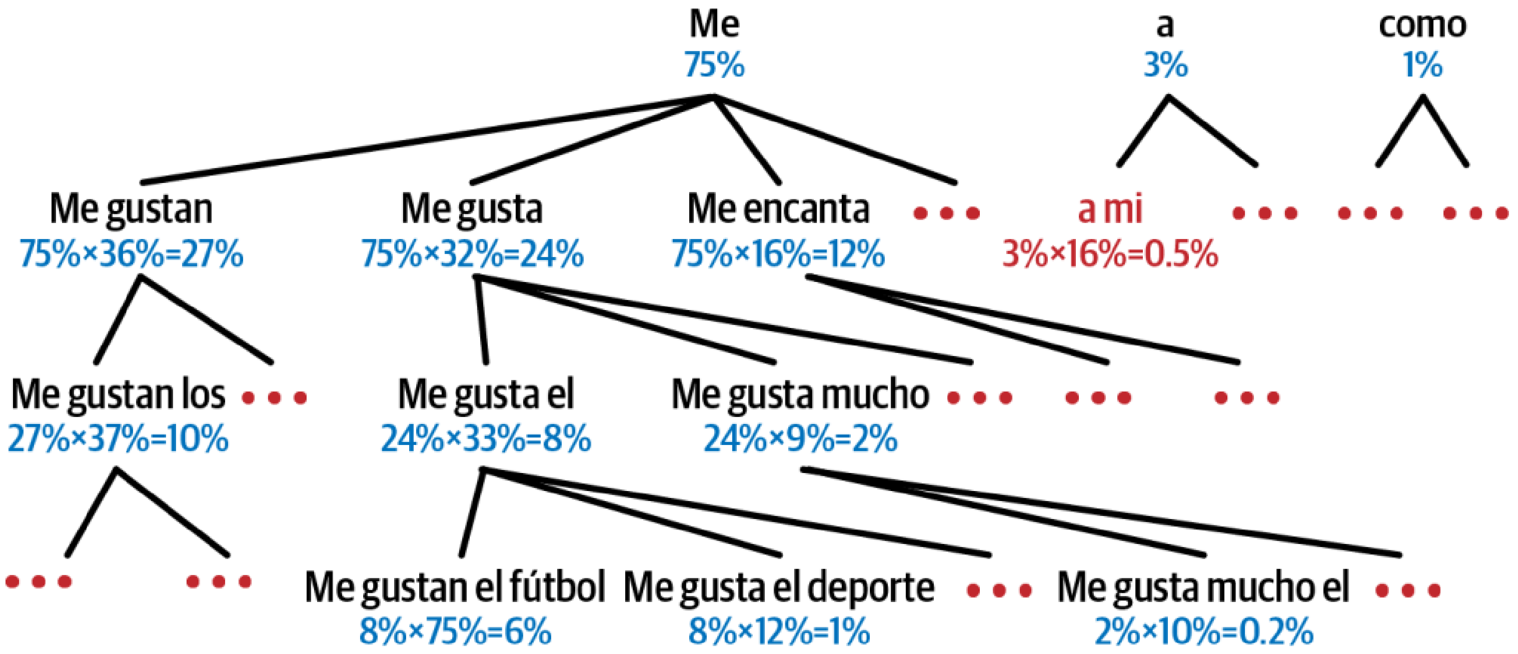
Bidirectional Recurrent Layer

- We cannot use the quadruple state directly as the initial state of the decoder’s LSTM layer, since it expects just two states (short-term and long-term).
- We cannot make the decoder bidirectional, since it must remain causal: otherwise it would cheat during training and it would not work.
- Instead, we can concatenate the two short-term states, and also concatenate the two long-term states:

```
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
encoder_state = [tf.concat(encoder_state[::2], axis=-1),  # short-term (0 & 2)
                 tf.concat(encoder_state[1::2], axis=-1)]  # long-term (1 & 3)
```

Beam Search

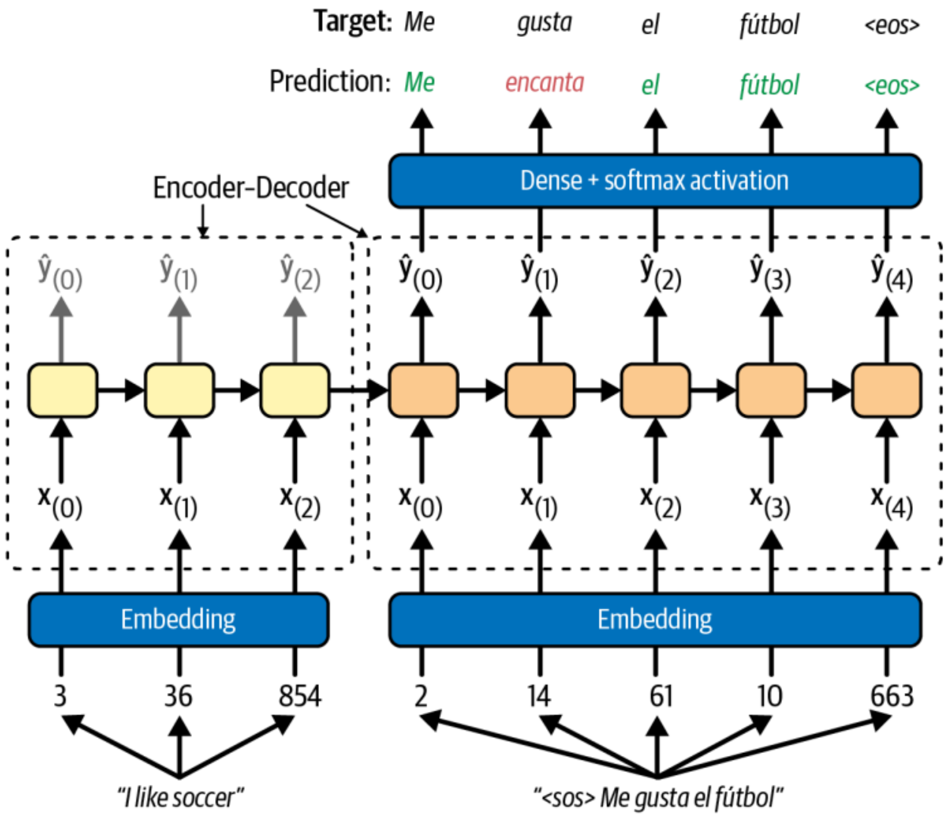
- *Beam search*: keeping track of a short list of the k most promising sentences.



8.
Attention Mechanisms

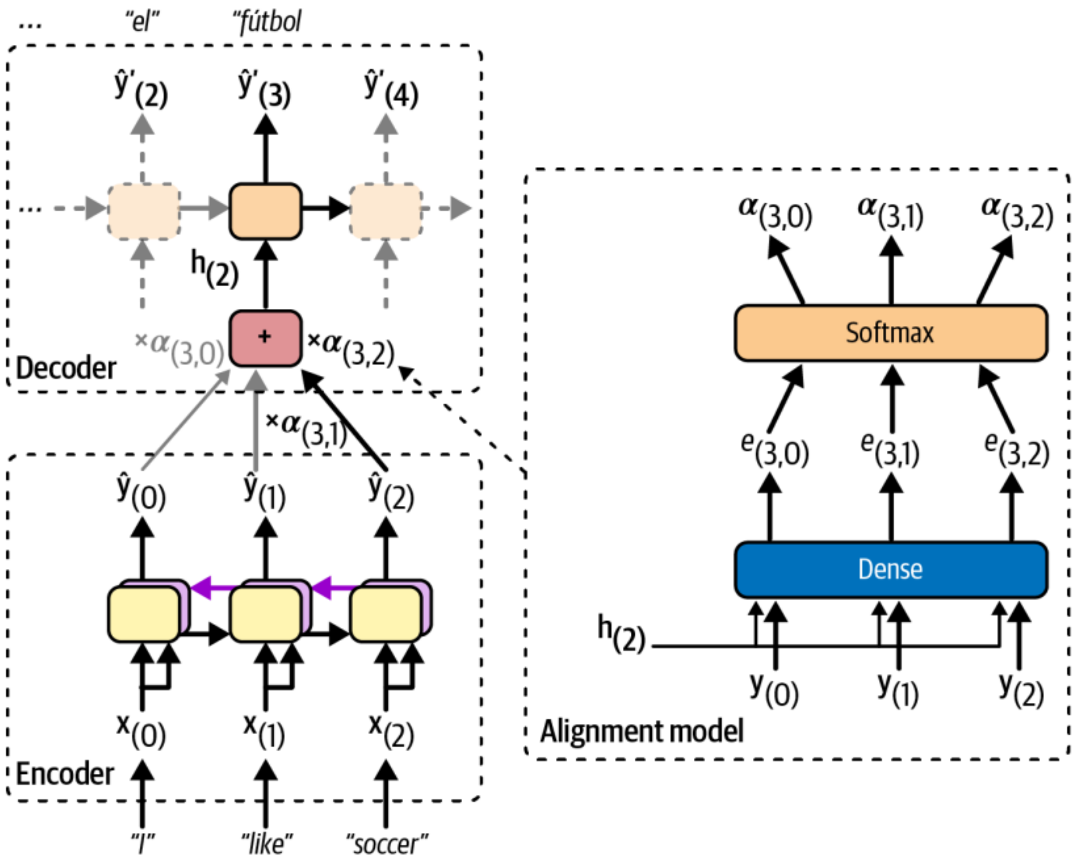
Encoder-Decoder Network

- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. ***Neural machine translation by jointly learning to align and translate***. In International Conference on Learning Representations (ICLR), 2015.


Target: Me gusta el fútbol <eos>
Prediction: Me encanta el fútbol <eos>
Dense + softmax activation
Encoder-Decoder
$\hat{y}_{(0)}$ $\hat{y}_{(1)}$ $\hat{y}_{(2)}$
$\hat{y}_{(0)}$ $\hat{y}_{(1)}$ $\hat{y}_{(2)}$ $\hat{y}_{(3)}$ $\hat{y}_{(4)}$
$x_{(0)}$ $x_{(1)}$ $x_{(2)}$
$x_{(0)}$ $x_{(1)}$ $x_{(2)}$ $x_{(3)}$ $x_{(4)}$
Embedding
Embedding
3 36 854
2 14 61 10 663
"I like soccer"
"<sos> Me gusta el fútbol"

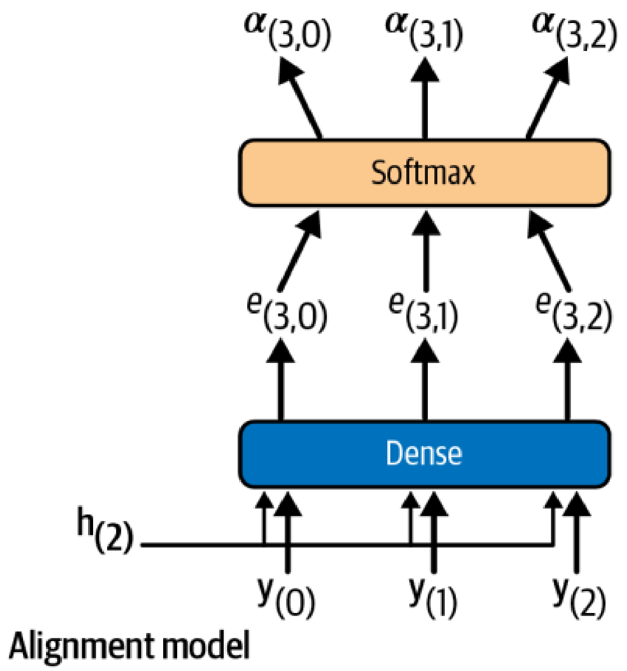
Alignment Model

- ➢ $\alpha_{(t,i)}$: weight of the i-th encoder output at the t-th decoder time step.
- ➢ These $\alpha_{(t,i)}$ weights are generated by a small neural network called an *alignment model* (or an *attention layer*), which is trained jointly with the rest of the encoder–decoder model.



Alignment Model

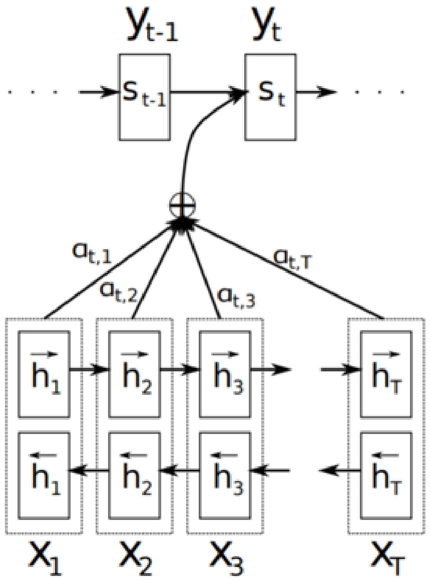
- The model starts with a dense layer that outputs a score for each encoder output which measures how well each output is aligned with the decoder's previous hidden state.
- All the scores go through a softmax layer to get a final weight for each encoder output ($\alpha_{(t,i)}$).
- This particular attention mechanism is called *Bahdanau attention.*
- Since it concatenates the encoder output with the decoder's previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).



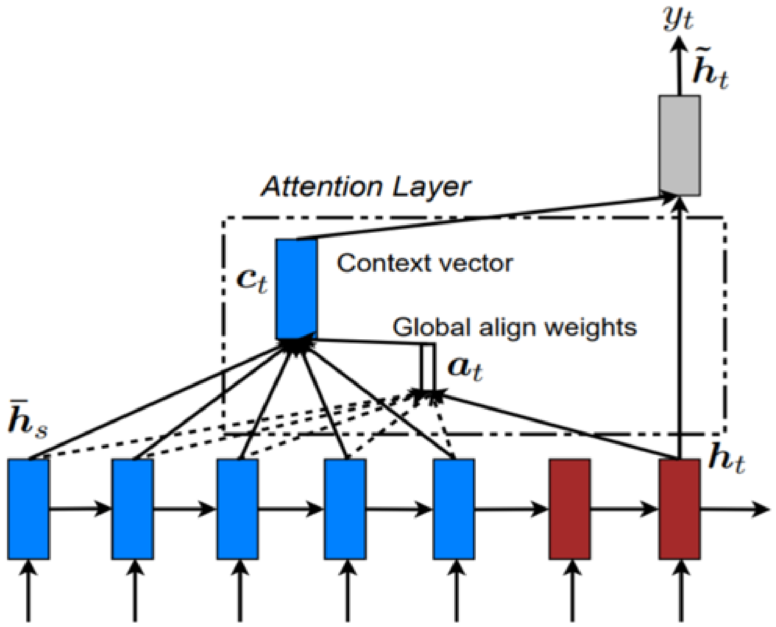
Multiplicative Attention

- *Luong attention* or *multiplicative attention*, was proposed in 2015, by Minh-Thang Luong et al.
- The alignment model has to measure the similarity between one of the encoder's outputs and the decoder's previous hidden state, so the authors proposed to compute the dot product of these two vectors, as this is often a fairly good similarity measure.
- Also use the decoder's hidden state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$ rather than $\mathbf{h}_{(t-1)}$), then use the output of the attention mechanism ($\tilde{\mathbf{h}}_{(t)}$) directly to compute the decoder's predictions, instead of using it to compute the decoder's current hidden state.
- Keras provides a `tf.keras.layers.Attention` layer for Luong attention, and an `AdditiveAttention` layer for Bahdanau attention.

Additive vs. Multiplicative Attention



Bahdanau attention mechanism



Luong attention mechanism

Adding Attention to Encoder-Decoder Model

- Since we will need to pass all the encoder’s outputs to the Attention layer, we first need to set `return_sequences=True` when creating the encoder:

```
encoder = tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(256, return_sequences=True, return_state=True))
```

- The rest of the model is exactly the same as earlier:

```
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
encoder_state = [tf.concat(encoder_state[::2], axis=-1),  # short-term (0 & 2)
                 tf.concat(encoder_state[1::2], axis=-1)]  # long-term (1 & 3)
decoder = tf.keras.layers.LSTM(512, return_sequences=True)
decoder_outputs = decoder(decoder_embeddings, initial_state=encoder_state)
```

Adding Attention to Encoder-Decoder Model

- We create the attention layer and pass it the decoder’s states (outputs for simplicity) and the encoder’s outputs:

```
attention_layer = tf.keras.layers.Attention()
attention_outputs = attention_layer([decoder_outputs, encoder_outputs])
```

- Pass the attention layer’s outputs directly to the output layer:

```
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(attention_outputs)
```

- After training this model, we find that it now handles much longer sentences. For example:

```
translate("I like soccer and also going to the beach")
```

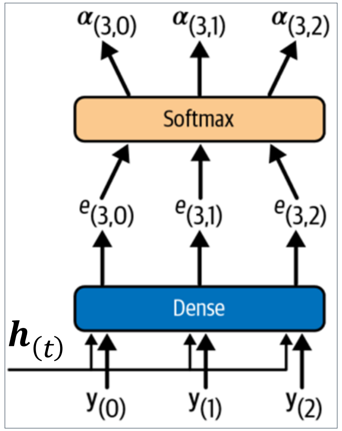
```
'me gusta el fútbol y también ir a la playa'
```

Trainable Memory Retrieval System

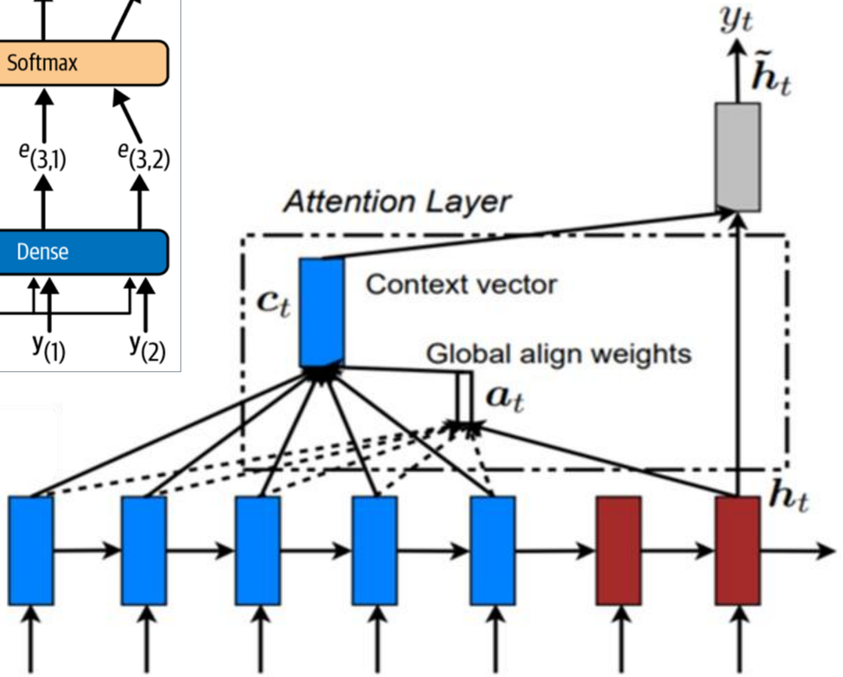
- Input sentence: "I like soccer"
- Encoder processes the input and finds: "I" → subject , "like" → verb , …
  - Similar to creating a dictionary {"subject": "They", "verb": "played", …}
- Suppose the decoder has already translated the subject, and it thinks that it should translate the verb next.
  - It needs to fetch the verb from the input sentence.
  - Similar to looking up the value that corresponds to the key "verb" in the dictionary created by the encoder.
- The model does not have discrete tokens to represent the keys (like "subject" or "verb"); instead, it has vectorized representations of these concepts that it learned during training.

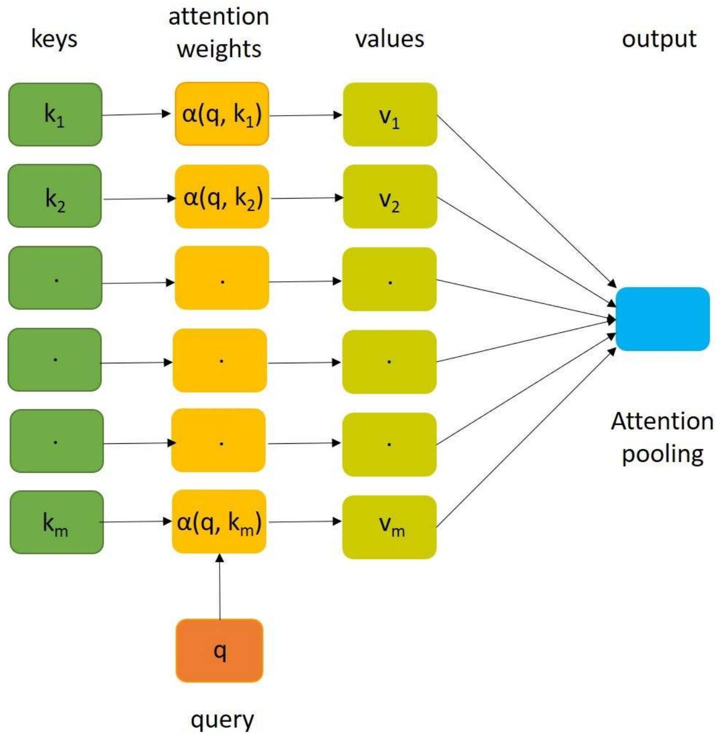
Trainable Memory Retrieval System

- Model uses vectorized representations, not discrete tokens, thus the decoder’s query may not perfectly match any key in the encoder’s dictionary {"subject”: “They”, “verb”: “played”, …}.
- Solution: measure similarity between the query and each key in the dictionary, then use the softmax function to convert these similarity scores to weights that add up to 1.
    - If the key that represents the verb is by far the most similar to the query, then that key’s weight will be close to 1.
- The attention layer computes a weighted sum of the corresponding values: if the weight of the “verb” key is close to 1, then the weighted sum will be very close to the representation of the word “played”.



$\boldsymbol{y}_{(1)}$





$$e(t,i) = \boldsymbol{h}_{(t)}{}^{T} \boldsymbol{y}_{(i)}$$

$$\alpha(t,i) = \frac{\exp\big(e(t,i)\big)}{\sum_j \exp\big(e(t,j)\big)} \qquad \tilde{\boldsymbol{h}}_{(t)} = \sum_j \alpha(t,j) \boldsymbol{y}_{(j)}$$

Trainable Memory Retrieval System

- The Keras `Attention` and `AdditiveAttention` layers both expect a list as input, containing two or three items: the *queries*, the *keys*, and optionally the *values*.
  - If you do not pass any values, they are automatically equal to the keys.
- In our code, the decoder outputs are the queries, and the encoder outputs are both the keys and the values:

```
attention_layer = tf.keras.layers.Attention()
attention_outputs = attention_layer([decoder_outputs, encoder_outputs])
```

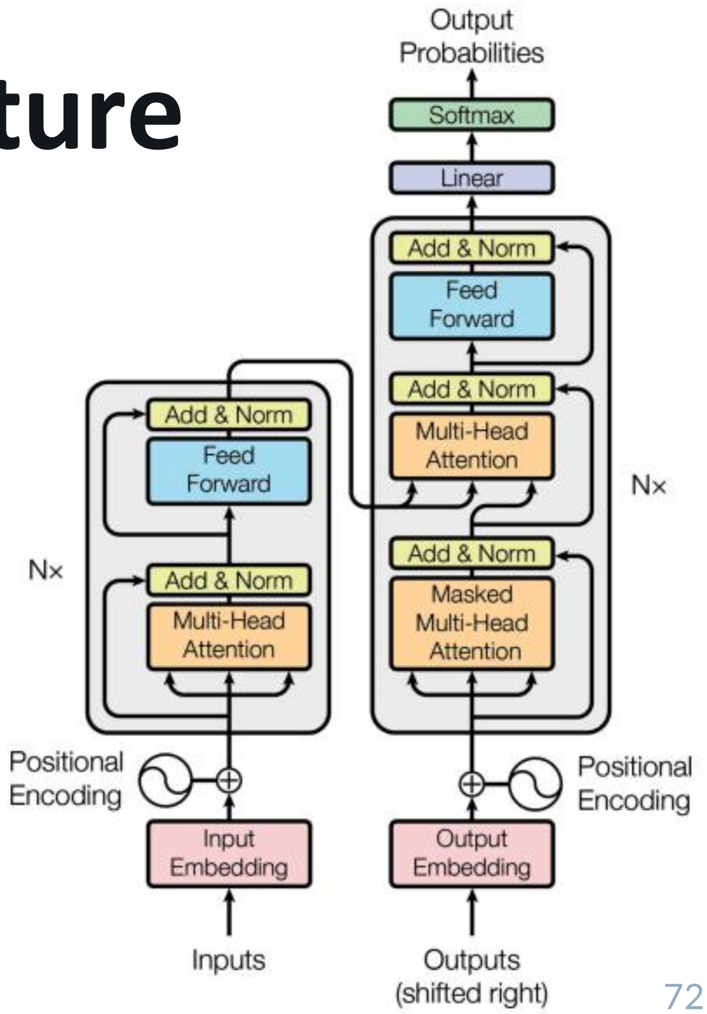
- For each decoder output (i.e., each query), the attention layer returns a weighted sum of the encoder outputs (i.e., the keys/values) that are most similar to the decoder output.

Attention Is All You Need!

- Key Breakthrough (2017):
  - Google researchers introduced the Transformer architecture in their paper, *"Attention Is All You Need."*
- Why Transformers Stand Out:
  - No Recurrent or Convolutional Layers: relies solely on attention mechanisms (plus embeddings, dense layers, normalization, ...).
  - Advantages over RNNs:
    - Avoids vanishing/exploding gradients.
    - Faster training with fewer steps.
    - Highly parallelizable across GPUs.
    - Better captures long-range patterns.

Transformer Architecture

- Each embedding layer outputs a 3D tensor of shape [*batch size*, *sequence length*, *embedding size*].
- If you use the transformer for NMT:
  - feed the English sentences to the encoder
  - feed the Spanish translations to the decoder
- The encoder's role: transform the inputs until each word's representation captures meaning of the word, in the context of the sentence.
- The decoder's role: transform each word representation in the translated sentence into a word representation of the next word in the translation.



Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)