

# Lecture 11

# Transformers

Fall 2024

Zoran B. Djordjević

# References

This lecture follows material presented in:

- Chapter 16, Hands on Machine Learning with Scikit-learn & TensorFlow, 3<sup>rd</sup> Edition, by Aurélien Géron, O'Reilly 2022
- Neural Machine Translation Tutorial on TensorFlow.org site  
<https://www.tensorflow.org/tutorials/text/transformer>
- Chapter 11, “Deep Learning with Python” 2<sup>nd</sup> Edition by Francois Chollet, 2021, Manning Publishing,
- Chapter 10, Transformers and Pretrained Language Models, “Speech and Language Processing”. Daniel Jurafsky & James H. Martin. Copyright © 2023. All rights reserved. Draft of January 7, 2023.

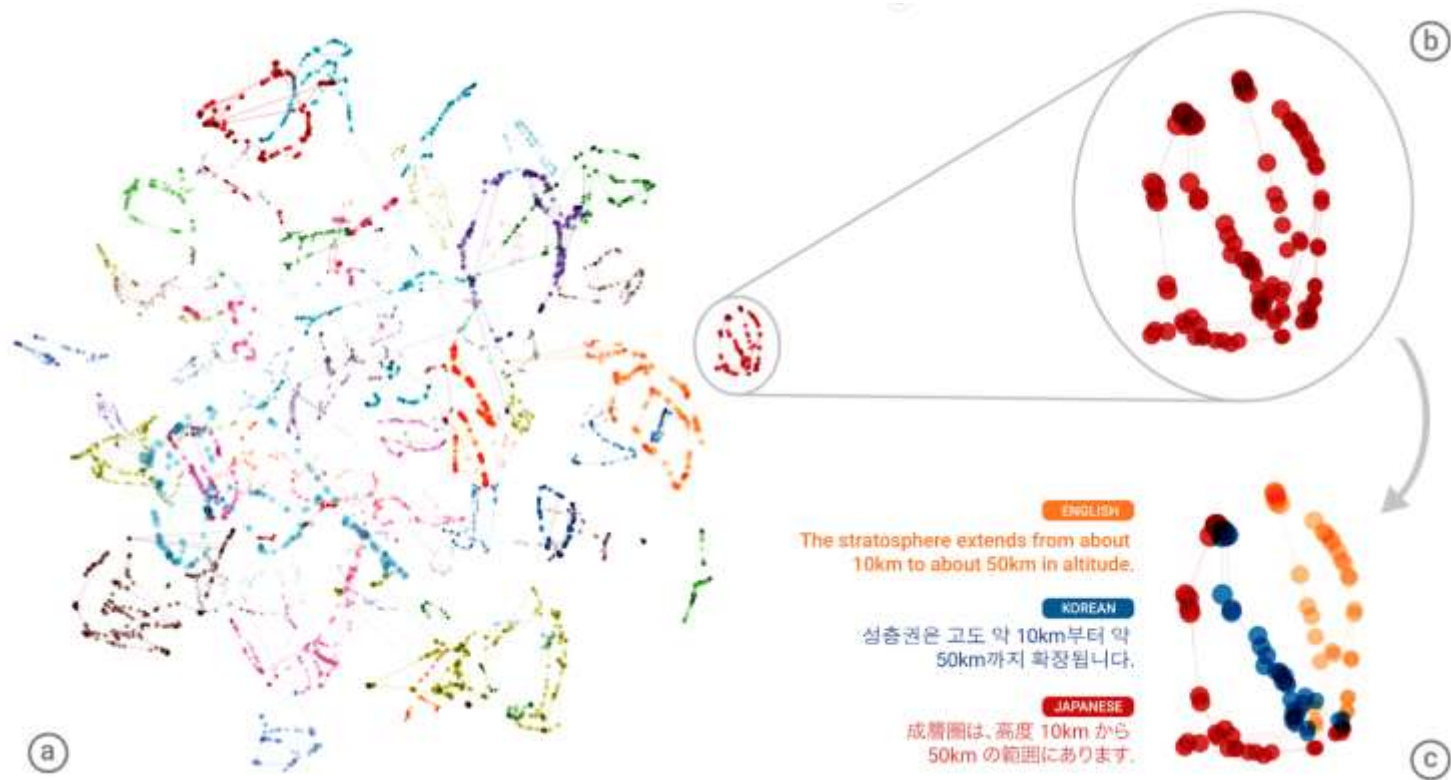
# Zero-Shot Translation and Interlingua

# Zero-Shot Translation with Google Multilingual NMT

- Google translates 100+ languages and a few years ago they operated many different systems translating between different pairs of languages.
- In September 2016, Google switched to a system called [Google Neural Machine Translation \(GNMT\)](#), an end-to-end learning framework that learns from millions of examples and provided significant improvements in translation quality. It was claimed that GNMT at the time relied on 5 layer LSTM bidirectional system.
- Google's new architecture requires no change in the base NMT system, but instead uses an additional "token" at the beginning of the input sentence to specify the required target language to translate to. In addition to improving translation quality, Google's new method also enables "Zero-Shot Translation" — translation between language pairs not seen before.
- For example, Google trains a multilingual system with Japanese $\rightleftarrows$ English and Korean $\rightleftarrows$ English examples. Multilingual system, with the same size as a single NMT system, shares its parameters for translation between different language pairs. This sharing enables the system to transfer the "translation knowledge" from one language pair to the others. This transfer learning and the need to translate between multiple languages forces the system to better use its modeling power.
- This system can perform satisfactory translations between language pairs which the system has never seen before? An example of this would be translations between Korean and Japanese where Korean $\rightleftarrows$ Japanese examples were not shown to the system beforehand.

# Universal Embeddings of Sentences

- The success of the zero-shot translation raises another important question: *Is the system learning a common representation in which sentences with the same meaning are represented in similar ways regardless of language — i.e., an "interlingua"?* Using a 3-dimensional representation of internal network data, Google could peek into the system as it translates a set of sentences between all possible pairs of the Japanese, Korean, and English languages.



# Interlingua

- Part (a) of the figure on previous slide shows an overall geometry of these translations.
  - The points in this view are colored by the meaning; a sentence translated from English to Korean with the same meaning as a sentence translated from Japanese to English share the same color.
  - From this view we can see distinct groupings of points, each with their own color.
- Part (b) zooms in to one of the groups.
- Part (c) colors sentences by the source language. Within a single group, we see a sentence with the same meaning but from three different languages.
- The above observations suggest that the network must be encoding something about the semantics of the sentence rather than simply memorizing phrase-to-phrase translations.
- Google researchers interpret this as a sign of existence of an "interlingua", a universal structure of the human language.
- This material is from: <https://research.google/blog/zero-shot-translation-with-googles-multilingual-neural-machine-translation-system/>
- Somewhat related concepts:
  - [https://en.wikipedia.org/wiki/Interlingual\\_machine\\_translation](https://en.wikipedia.org/wiki/Interlingual_machine_translation)
  - <https://en.wikipedia.org/wiki/Interlingua>

# Attention

# Are you paying attention?

- As you are reading a book, you are skimming some parts and attentively reading others, depending on what your goals or interests are. Could our DL models do the same?
- This is simple yet powerful idea: not all input information seen by a DL model is equally important to the task at hand, so models should “pay more attention” to some features and “pay less attention” to other features.
- We have already encountered a similar concept:
- Max pooling in CNNs looks at a pool of features in a spatial region and selects just one feature (the maximal value out of 4) to keep. That’s an “all or nothing” form of attention: keep the most important feature and discard the rest.
- TF-IDF normalization we encountered in NLP analysis assigns importance scores to tokens based on how much information they are likely to carry. Important tokens get boosted while irrelevant tokens get faded out. That’s a continuous form of attention.
- There are many different forms of attention you could imagine, but they all start by computing importance scores for a set of features, with higher scores for more relevant features and lower scores for less relevant ones.
- How these scores are computed, and what you do with them, varies from approach to approach.



# Attention Mechanism

- [Dzmitry Bahdanau](#), [Kyunghyun Cho](#), & [Yoshua Bengio](#) in a paper entitled: *Neural Machine Translation by Jointly Learning to Align and Translate*, 2015 (<https://arxiv.org/abs/1409.0473>) introduced the attention mechanism.

Paraphrasing the paper:

- *In RNN (LSTM) encoder-decoder architecture for the language translation every word in an input sentence is first encoded as an embedded vector. The sequence of such vectors is then transformed into a fixed length context vector or latent space vector. The decoder extracts the translation out of that context vector.*
- *With attention mechanism, a fuller representation of the encoded input is kept and the decoder, for each word in the output, learns to pay attention to different parts of the input.*
- *Each time this new model generates a word in a translation, it (soft-)searches for a set of positions in a source sentence where the most relevant information is concentrated. The model pays more attention to more relevant information.*
- *The model predicts a target word based on the vectors associated with these source positions and all the previous generated target words.*

# Self Attention

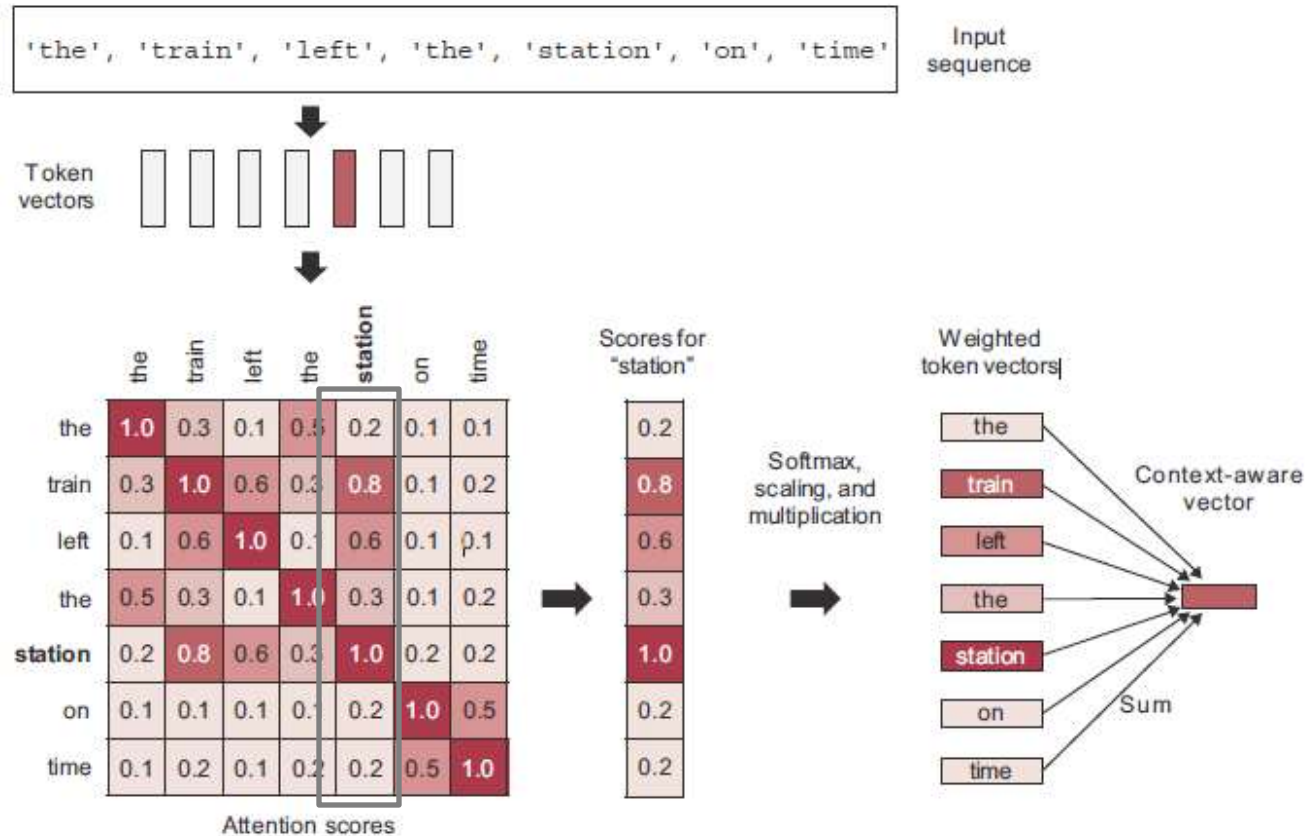
- A serious problem with embedding spaces in Word2Vec is that words with different meaning (river **bank** vs. financial **bank**) but same spelling, have the same embedding vector.
- Smart embedding space would provide a different vector representation for a word depending on the other words surrounding it. That is where *self-attention* comes in.
- The purpose of self-attention is to modulate the representation of a token by using the representations of related tokens in the sequence. Self-attention produces context aware token representations.
- Consider an example sentence: “*The train left the station on time.*” Now, consider one word in the sentence: *station*. What kind of station are we talking about? Could it be a radio station? We use attention to modify the embedding representation of word “*station*”, so it reflects the context around it.

We do it in the following two steps:

1. Compute relevancy scores between the vector for “station” and every other word in the sentence. These are our “attention scores.” We use the dot product between every two word vectors as a measure of the strength of their relationship. This is a very computationally efficient distance function. These scores also go through a scaling function and a softmax.
  2. Compute the sum of all word vectors in the sentence, weighted by our relevancy scores. Words closely related to “station” will contribute more to the sum (including the word “station” itself. The irrelevant words will contribute almost nothing. The resulting vector is our new representation for “station”: a representation that incorporates the surrounding context. In particular, it includes part of the “train” vector, clarifying that it is, in fact, a “train station.”
- These calculations are illustrated on the next two slides

# Calculation of Self-Attention

- We repeat this process for every word in the sentence, producing a new sequence of new vectors encoding the sentence. Below, we illustrate the calculation for the word “station”



- Self-attention scores are computed between “station” and every other word in the sequence. Those scores for word “station” are scaled by the square root of sentence length and passed through softmax function which generates “probabilities”. Those probabilities are weights of all word vectors in the sentence in the sum that becomes the new “station” vector.

# Pseudo code for Self-Attention Calculation


- A pseudo Numpy –like code for the calculation of the self-attention for a sentence reads like:

```
def self_attention(input_sequence):  
    output = np.zeros(shape=input_sequence.shape)  
    # Iterate over each token in the input sequence.  
    for i, pivot_vector in enumerate(input_sequence):  
        scores = np.zeros(shape=(len(input_sequence),))  
        for j, vector in enumerate(input_sequence):  
            # Compute the dot product (attention score) between  
            # the token and every other token.  
            scores[j] = np.dot(pivot_vector, vector.T)  
        scores /= np.sqrt(input_sequence.shape[1])  
        # Scale by a normalization factor, and apply a softmax  
        scores = softmax(scores)  
        new_pivot_representation = np.zeros(shape=pivot_vector.shape)  
        for j, vector in enumerate(input_sequence):  
            # Take the sum of all tokens weighted by the attention scores.  
            new_pivot_representation += vector * scores[j]  
        # The above sum is new representation of every word in the sentence  
        output[i] = new_pivot_representation  
    return output
```

# Generalized Attention

- Schematically, the self-attention mechanism performs the following:

```
outputs = sum(inputs * pairwise_scores(inputs, inputs))
```



- This means “for each token in `inputs` (A), compute how much that token is related to every token in `inputs` (B) and use these scores as weights in a sum of tokens from `inputs` (C).” A, B, and C do not have to refer to the same input sequence. In the general case, you could be doing this with three different sequences. We call them “query,” “keys,” and “values.” The operation becomes “for each element in the query (A) compute how much the element is related to every key (B) and use these scores to weight a sum of values C”.

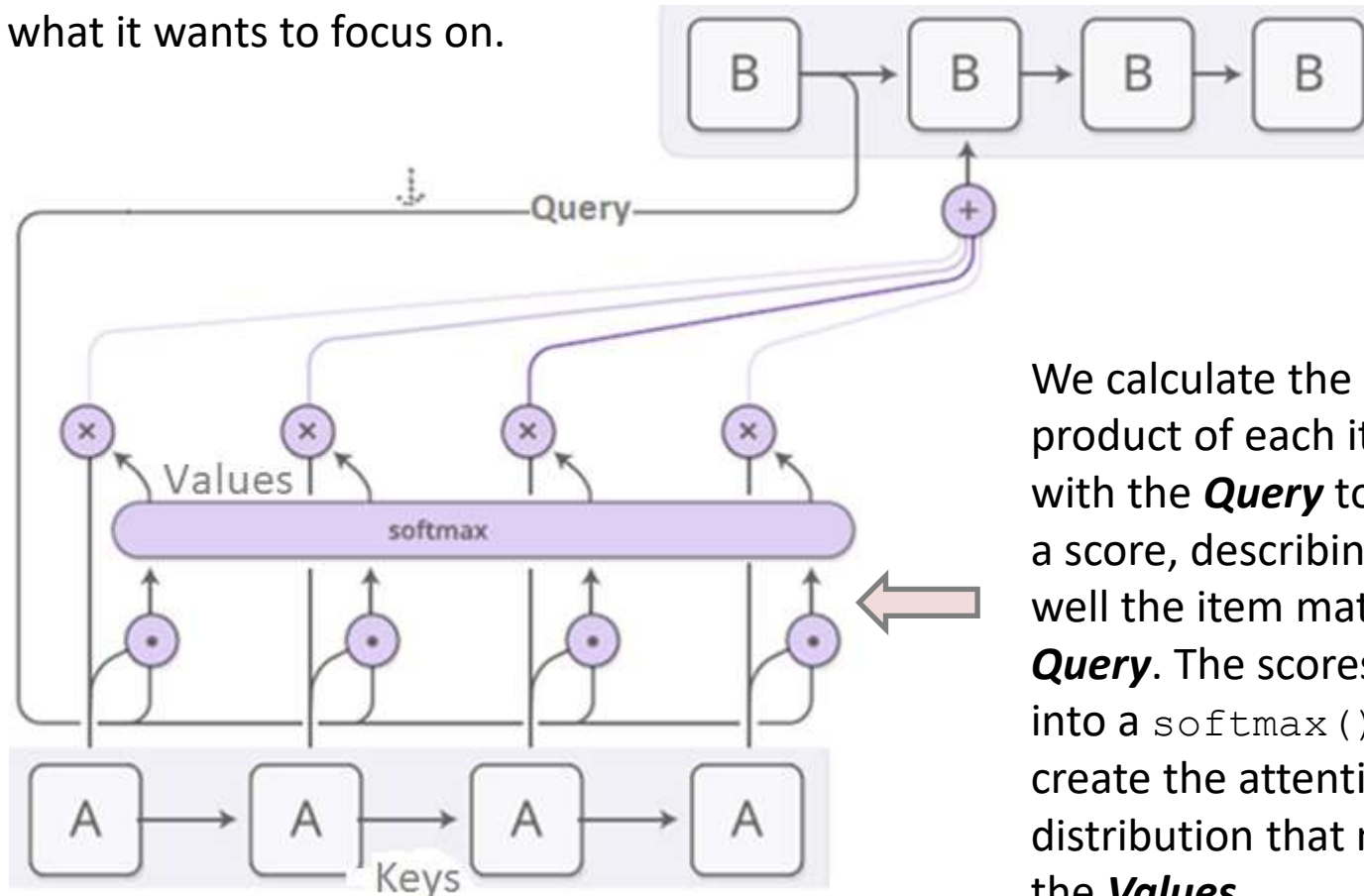
```
outputs = sum(values * pairwise_scores(query, keys))
```

- Conceptually, this is what generalized attention is doing.
- A reference sequence describes something you are looking for: a `query`.
- You are trying to extract information from a body of knowledge: the `values`.
- Each `value` is assigned a `key` that describes the value in a format that can be readily compared to a `query`. You simply match the `query` to the `keys`. Then you return a weighted sum of `values`.
- In practice, the `keys` and the `values` are often the same sequence. In machine translation, for instance, the `query` would be the target sequence, and the source sequence would play the roles of both `keys` and `values`: for each element of the target. When translating to Spanish each element of the target (“tiempo”) will require us to go back to the source (“How is the weather today?”) and identify different tokens that are related. “tiempo” and “weather” should have a strong match.

# Calculation of Generalized Attention

- The distribution of attention is usually generated with content-based attention.

The attending RNN generates a *Query* describing what it wants to focus on.



We calculate the dot product of each item (**Key**) with the **Query** to produce a score, describing how well the item matches the **Query**. The scores are fed into a `softmax()` unit to create the attention distribution that modifies the **Values**

# Another look at query, key and value

- Self-attention layer can be compared with the standard Python dictionary.
- Formally the dictionary is a set of pairs of the form (key, value), where the key acts as the unique ID to retrieve the corresponding value. For example, in the third and fourth line of the code below, we query the dictionary with two different strings ("Alice" and "Alce"). The dictionary compares the query string to all keys which are stored inside and returns the corresponding value if a perfect match is found, or an error otherwise.

```
D = dict()  
D["Alice"] = 2  
D["Alice"] # returns 2  
D["Alce"]  # returns an error
```

- One could envision a dictionary where the search is performed over some kind of similarity measure and the dictionary would return the value corresponding to the closest key provided.

# “Attention is All You Need”

## Transformers

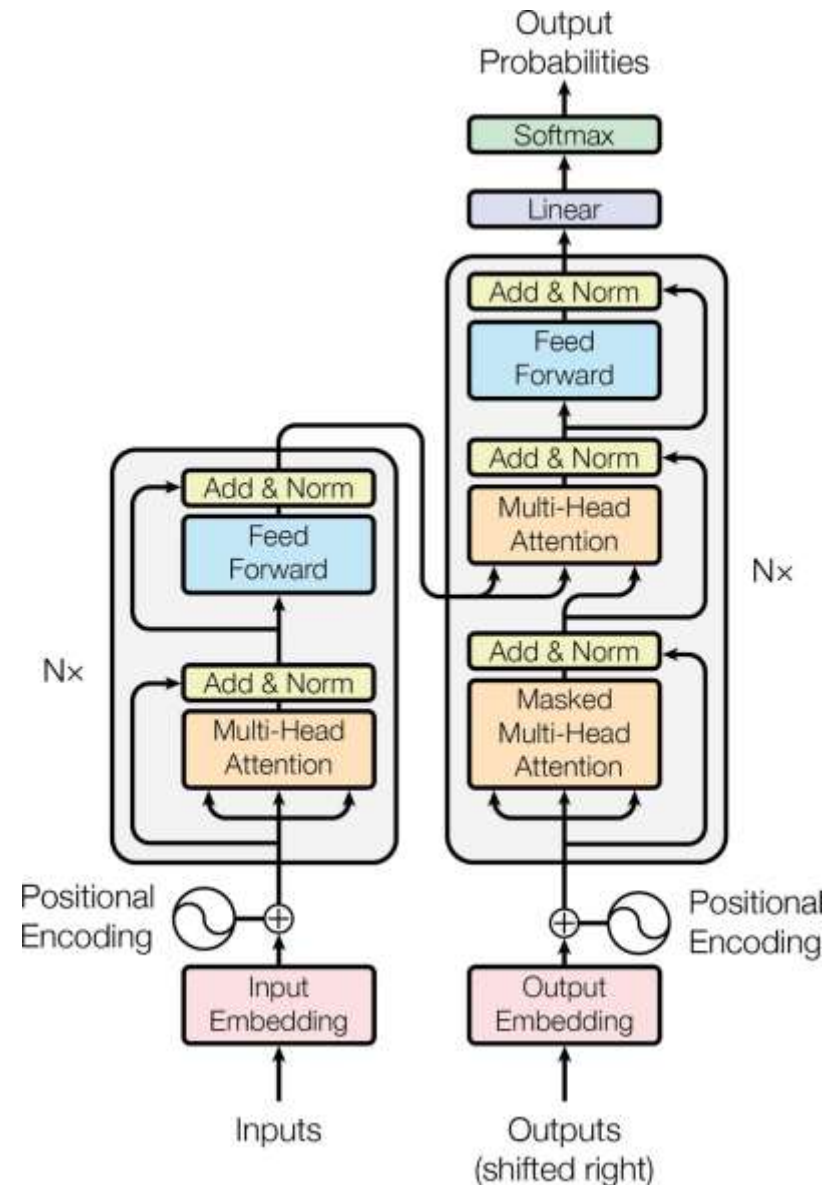


# Attention is All You Need

- In **2017 paper** (<https://arxiv.org/abs/1706.03762>) "*Attention is All You Need*" a team of Google researchers led by **Ashish Vaswani** incorporated the attention mechanism developed by **Dzmitry Bahdanau** et al (2015) into *a new and very efficient* architecture called the **Transformer**.
- The transformer significantly improved the state of the art in NMT without using any recurrent or convolutional layers.
- Transformers use the attention layers, embedding layers, dense layers, normalization layers, and a few other bits and pieces.
- Transformer architecture is much faster to train and easier to parallelize. Google managed to train the transformer models at a fraction of the time and cost of the previous state-of-the-art LSTM models.
- The transformer architecture is the basis of all large language models (LLMs) and most foundational models in use today.
- The original transformer architecture is presented on the next slide.
- Slightly more modern versions have some additional normalization layers but are basically unmodified from the original design .

# The Transformer Architecture

- The left-hand part is the encoder. It takes as input a batch of sentences represented as sequences of word IDs (the input shape is [batch size, max input sentence length]), and it encodes each word into a 512-dimensional representation.
- The encoder's output shape is [batch size, max input sentence length, 512]. The top part of the encoder is stacked N times (in the paper,  $N = 6$ ).
- The righthand part is the decoder. During training, it takes the target sentence as input (also represented as a sequence of word IDs), shifted one time step to the right (i.e., a start-of-sequence token is inserted at the beginning).
- Decoder also receives the outputs of the encoder (the arrows coming from the left side). Note that the top part of the decoder is also stacked N times. The encoder stack's final outputs are fed to the decoder at each of these N levels.
- The decoder outputs a probability for each possible next word, at each time step (its output shape is [batch size, max output sentence length, vocabulary length]).

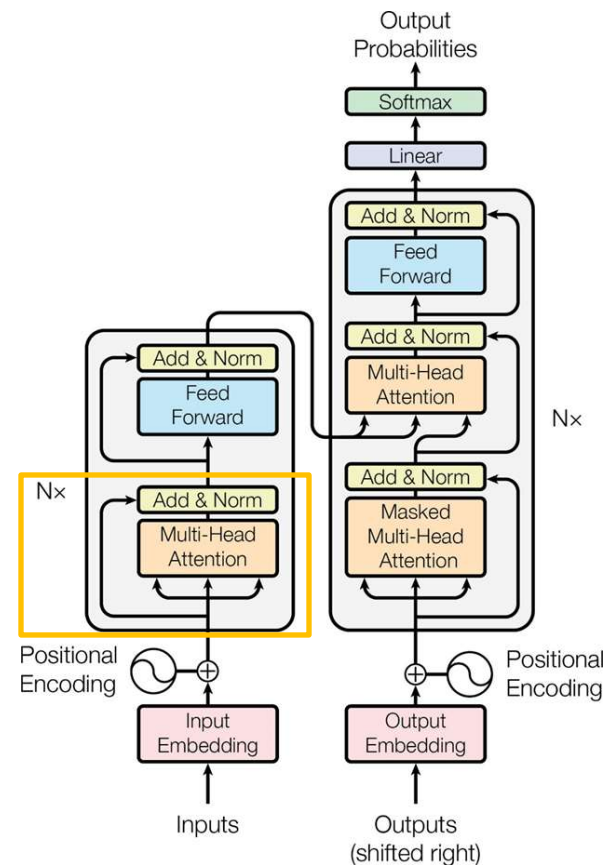


# Why Transformers are significant

- Transformers excel at modeling sequential data, such as natural language.
- Unlike recurrent neural networks (RNNs), Transformers are parallelizable. This makes them efficient on hardware like GPUs and TPUs. The main reason is that **Transformers replaced recurrence with attention, and computations can happen simultaneously**. Layer outputs can be computed in parallel, instead of in a series like in an RNN.
- Unlike RNNs or convolutional neural networks (CNNs), Transformers are able to capture distant or long-range contexts and dependencies in the data between distant positions in the input or output sequences. Thus, longer connections can be learned.
- Attention allows each location to have access to the entire input at each layer, while in RNNs and CNNs, the information needs to pass through many processing steps to move a long distance, which makes it harder to learn.
- Transformers make no assumptions about the temporal/spatial relationships across the data.

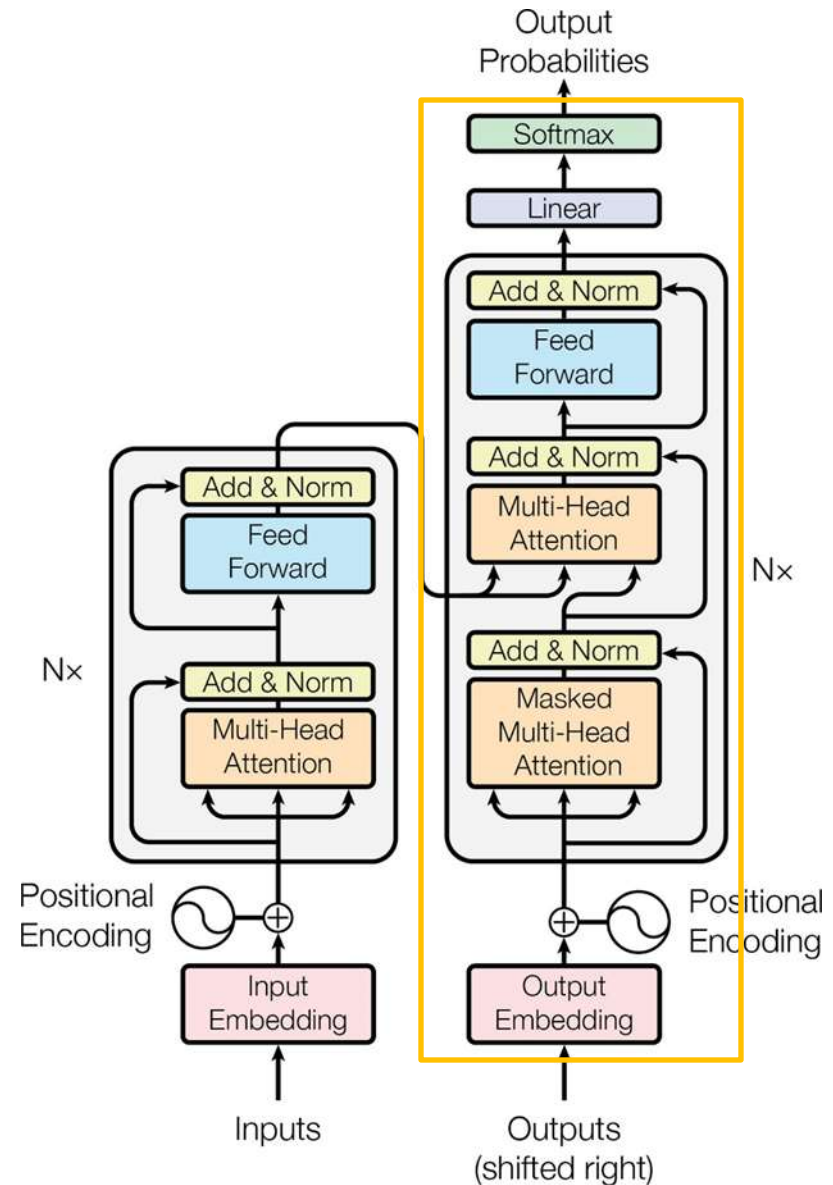
# NMT with Transformer, Encoder

- If you use the transformer for NMT, then during training you must feed the English sentences to the encoder and the corresponding Spanish translations to the decoder, with an extra SOS token inserted at the start of each sentence.
- At inference time, you must call the transformer multiple times, producing the translations one word at a time and feeding the partial translations to the decoder at each round, just like we did earlier in the `translate()` function.
- The encoder's role is to gradually transform the inputs—word representations of the English sentence—until each word's representation perfectly captures the meaning of the word, in the context of the sentence.
- For example, if you feed the encoder with the sentence “I like soccer”, then the word “like” will start off with a rather vague representation, since this word could mean different things in different contexts: think of “I like soccer” versus “It's like that”.
- After the encoder, the word's representation should capture the correct meaning of “like” in the given sentence (i.e., to be fond of), as well as any other information that may be required for translation (e.g., it's a verb). This is mostly done in **Multi-Head Attention** portion of the encoder.



# NMT with Transformer, Decoder

- The decoder's role is to gradually transform each word representation in the translated sentence into a word representation of the next word in the translation.
- For example, if the sentence to translate is “I like soccer”, and the decoder's input sentence is “<SOS> me gusta el fútbol”, then after going through the decoder, the word representation of the word “el” will end up transformed into a representation of the word “fútbol”. Similarly, the representation of the word “fútbol” will be transformed into a representation of the EOS token.
- After going through the decoder, each word representation goes through a final Dense layer with a softmax activation function, which will hopefully output a high probability for the correct next word and a low probability for all other words. The predicted sentence should be “me gusta el fútbol <EOS>”.

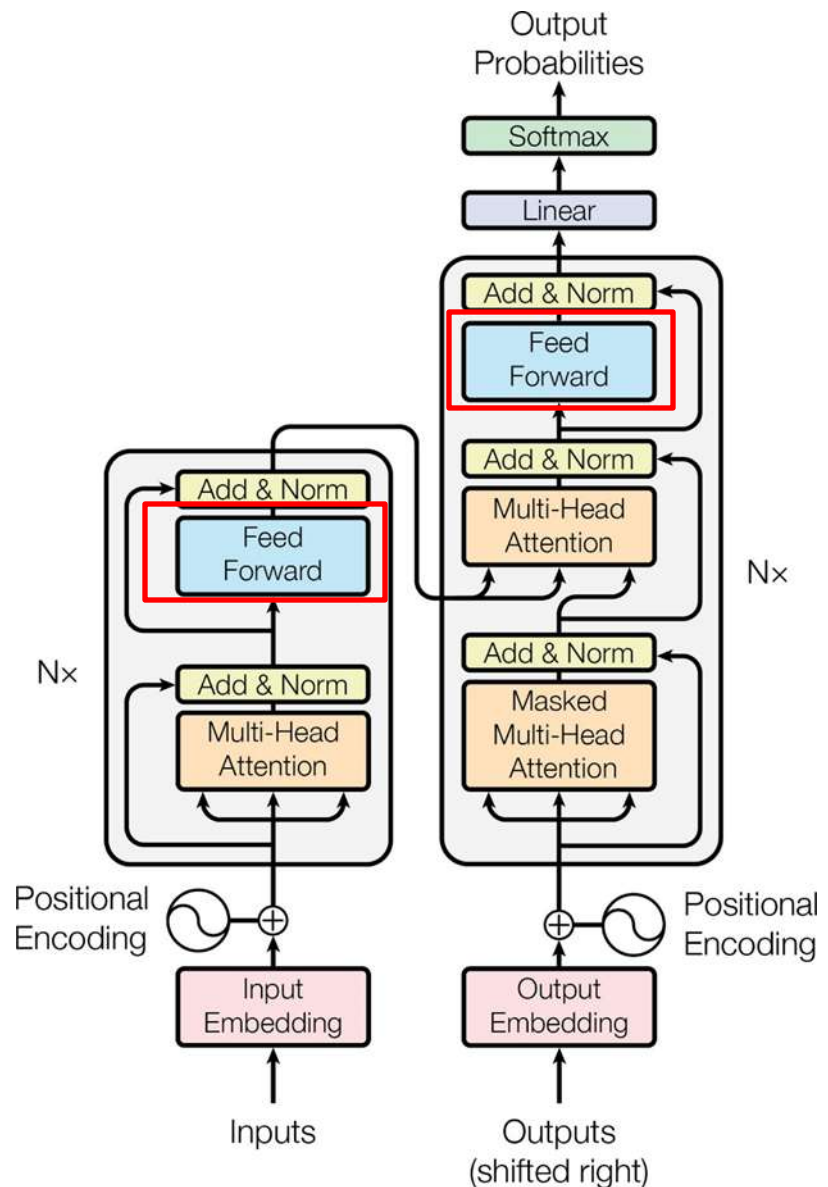


# Elements of the Stack

- The core idea behind the Transformer model is *self-attention*—the ability to attend to different positions of the input sequence to compute a representation of that sequence. Transformer implements stacks of self-attention layers.
- A transformer model handles variable-sized input using stacks of self-attention layers.
- This general architecture has several advantages:
  - It make no assumptions about the temporal/spatial relationships across the data.
  - Layer outputs can be calculated in parallel, instead of a series like an RNN.
  - Distant items can affect each other's output without passing through many RNN-steps, or convolution layers.
  - It can learn long-range dependencies. This is a challenge in many sequence tasks.
- The downsides of this architecture are:
  - For a time-series, the output for a time-step is calculated from the *entire history* instead of only the inputs and current hidden-state. This *may* be inefficient.
  - If the input *does* have a temporal/spatial relationship, like text, some positional encoding must be added, or the model will effectively see a bag of words.
- **During inference**, the decoder cannot be fed targets, so we feed it the previously discovered (translated) words, starting with a start-of-sequence (SOS) token.
- During the inference, the model needs to be called repeatedly, predicting one more word at every next round until the end-of-sequence (EOS) token is produced.

# Elements of the Stack

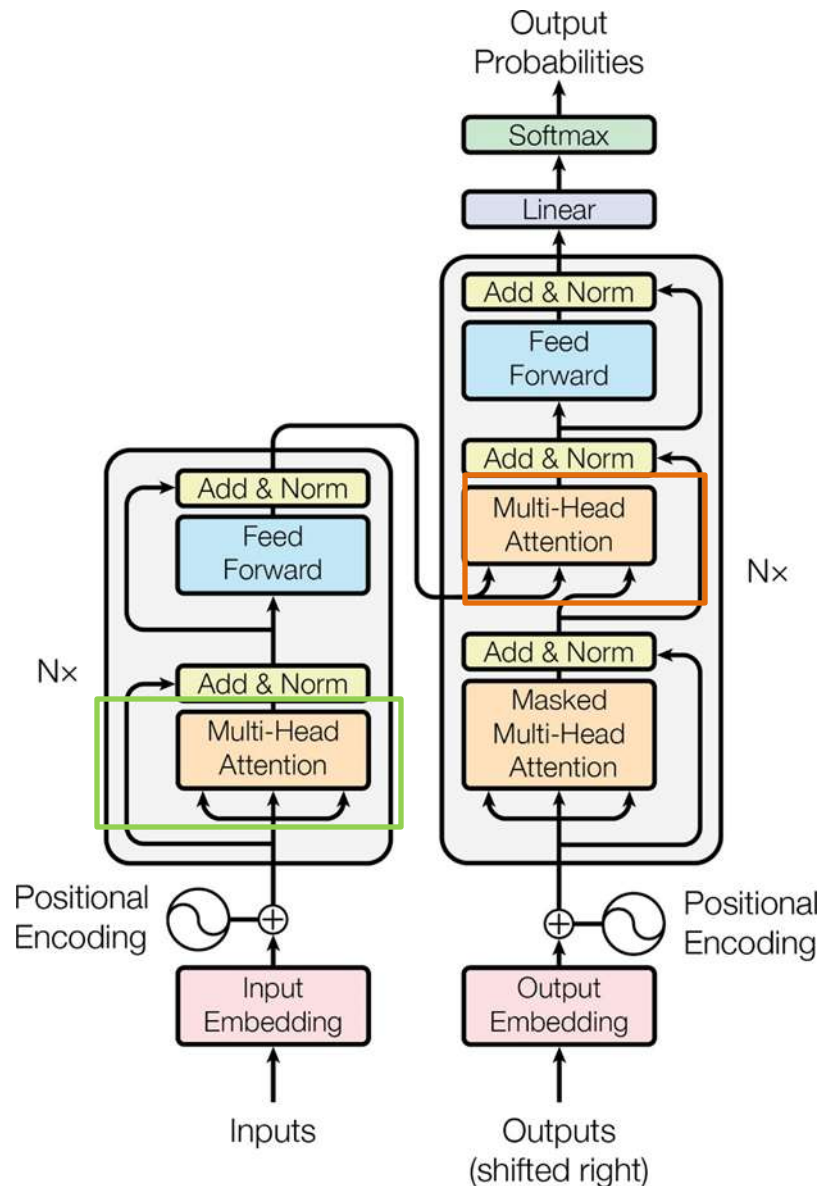
- There are two embedding layers,  $5 \times N$  skip (residual) connections, each of them followed by a normalization layer. Skip connections are needed to make sure we do not destroy any valuable information along the way.
- $2 \times N$  "Feed Forward" (MLPs, multi layer perceptron) modules that are composed of two dense layers each. The first dense layer uses the ReLU activation function. The second has no activation function.
- The output layer on the decoder is a dense layer with **softmax** activation function.
- To translate a sentence, we cannot look at one word at a time? We need components that would let us look at the whole sentences at once.





# Elements of the Stack, Multi-Head Attention

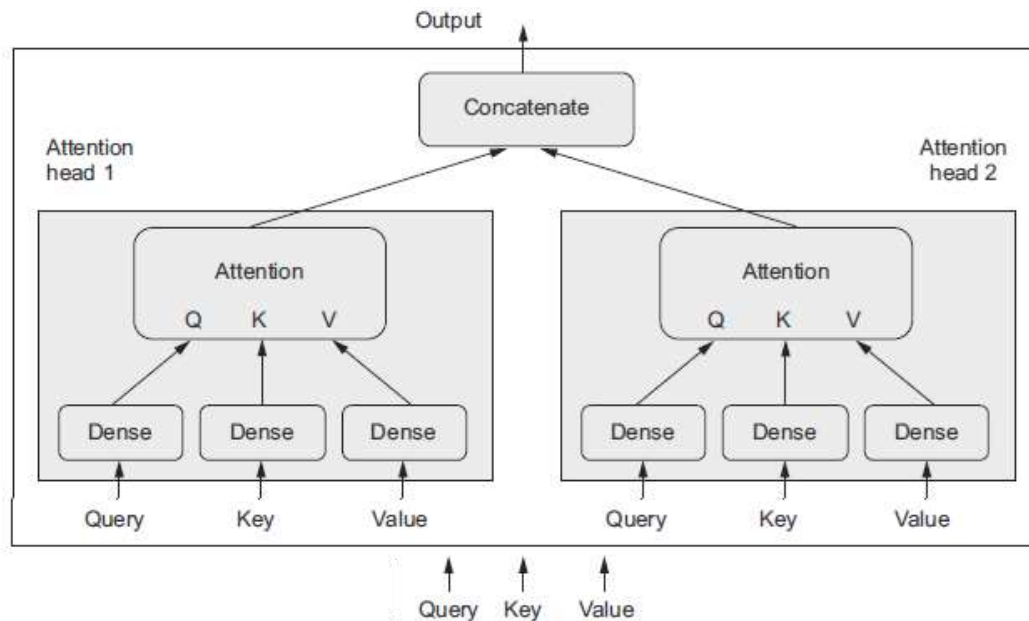
- The *encoder's Multi-Head Attention* layer encodes each word's relationship with every other word in the same sentence, paying more attention to the most relevant ones. For example, the output of this layer for the word "Queen" in the sentence "They welcomed the Queen of the United Kingdom" will depend on all the words in the sentence, but it will probably pay more attention to the words "United" and "Kingdom" than to the words "They" or "welcomed."
- This attention mechanism is called *self-attention* (the sentence is paying attention to itself).
- The decoder's upper *Multi-Head Attention* layer is where the decoder pays attention to the words in the input sentence. For example, the decoder will probably pay close attention to the word "Queen" in the input sentence when it is about to output this word's translation, e.g. "Reina".





# Multi-head Attention, Again

- Multi-head” refers to the fact that the output space of the self-attention layer gets factored into a set of independent subspaces, learned separately: the initial query, key, and value are sent through three independent sets of Dense projections, resulting in three separate vectors.
- Each vector is processed via neural attention, and the three outputs are concatenated back together into a single output sequence. Each such subspace is called a “head.” The full picture is shown in figure on the right
- The presence of the learnable dense projections enables the layer to actually learn something. In addition, having independent heads helps the layer learn different groups of features for each token, where features within one group are correlated with each other but are mostly independent from features in a different group.



# Multi-Head Attention & Scaled Dot-Product Attention

- Multi-Head Attention layer is based on *Scaled Dot-Product Attention* layer.
- Suppose the encoder analyzed the input sentence "They played chess," and it managed to understand that the word "They" is the subject and the word "played" is the verb, so it encoded this information in the representations of these words. Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next. For this, it needs to fetch the verb from the input sentence.
- This is analog to a dictionary lookup: it's as if the encoder created a dictionary {"subject": "They", "verb": "played",...} and the decoder wanted to look up the value that corresponds to the key "verb." However, the model does not have discrete tokens to represent the keys (like "subject" or "verb"); it has vectorized representations of these concepts (which it learned during training), so the key it will use for the lookup (called the query) will not perfectly match any key in the dictionary.
- The solution is to compute a similarity measure between the query and each key in the dictionary, and 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.
- Then the model can compute a weighted sum of the corresponding values, so 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."

# Scaled Dot-Product Attention

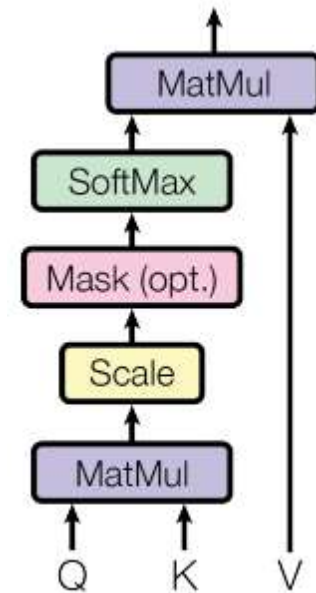
- Scaled Dot Product Attention, based on inputs: Q (query), K (key), and V (value) is calculated as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

- In some implementations, vectors  $Q$ ,  $K$  and  $V$  pass through dense layers which multiply them by trainable matrices:  $W_q$ ,  $W_k$ , and  $W_v$

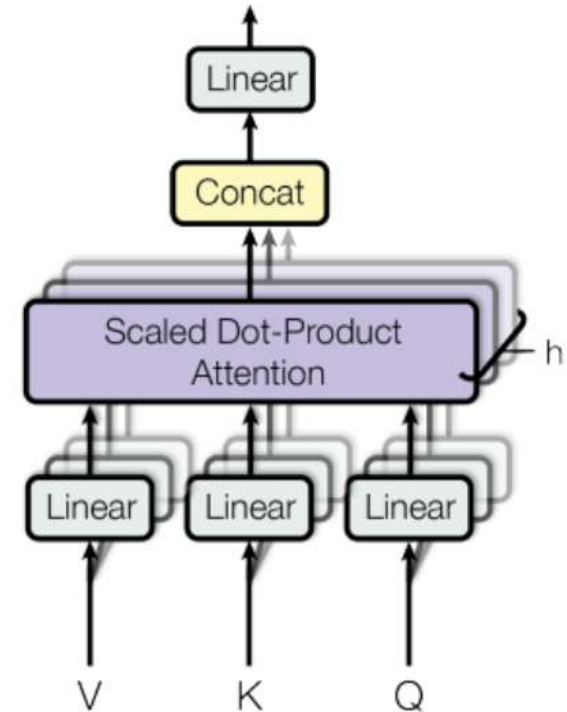
$$Attention(Q, K, V) = softmax(\frac{QW_qW_k^TK_T}{\sqrt{d_k}})W_vV$$

- Matrix  $QK^T$  has one similarity score for each query-key pair.
- The dot-product attention is scaled by a factor of square root of the depth. This is done because for large values of depth, the dot product grows large in magnitude pushing the softmax function where it has small gradients resulting in a very hard softmax.
- For example, consider that  $Q$  and  $K$  have a mean of 0 and variance of 1. Their matrix multiplication will have a mean of 0 and variance of  $d_k$ . Hence, square root of  $d_k$  is used for scaling (and not any other number) because the  $matmul(Q, K)$  should have a mean of 0 and variance of 1, and we get a gentler softmax.



# Multi-head attention

- Multi-head attention consists of four parts:
  - Linear layers and split into heads.
  - Scaled dot-product attention.
  - Concatenation of heads.
  - Final linear layer.
- Each multi-head attention block gets three inputs matrices:  $Q$  (query),  $K$  (key),  $V$  (value). Dimensions of those matrices are the number of queries times dimension of queries (values) times batch size. These are put through linear (Dense) layers and split up into multiple heads.
- The `scaled_dot_product_attention()` is applied to each head (broadcasted for efficiency). An appropriate mask must be used in the attention step. The attention output for each head is then concatenated (using `tf.transpose`, and `tf.reshape`) and put through a final Dense layer.
- Instead of one single attention head matrices  $Q$ ,  $K$ , and  $V$  are split into multiple heads because it allows the model to jointly attend to information at different positions from different representational spaces.
- After the split each head has a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.



# Normalization Layers

- The normalization layers we are using here are not `BatchNormalization` layers like those we used before in image models. `BatchNormalization` does not work well for sequence data. Instead, we are using `keras.layers.LayerNormalization` layer, which normalizes each sequence independently from other sequences in the batch.
- In NumPy-like pseudocode two layers read:

```
# Input shape: (batch_size, sequence_length, embedding_dim)
```

```
def layer_normalization(batch_of_sequences):
```

```
    # To compute mean and variance, we only pool data over the last axis(axis -1).
```

```
    mean = np.mean(batch_of_sequences, keepdims=True, axis=-1)
```

```
    variance = np.var(batch_of_sequences, keepdims=True, axis=-1)
```

```
    return (batch_of_sequences - mean) / variance
```

```
# Input shape: (batch_size, height, width, channels)
```

```
def batch_normalization(batch_of_images):
```

```
    # Pool data over the batch axis (axis 0), which creates interactions  
    # between samples in a batch.
```

```
    mean = np.mean(batch_of_images, keepdims=True, axis=(0, 1, 2))
```

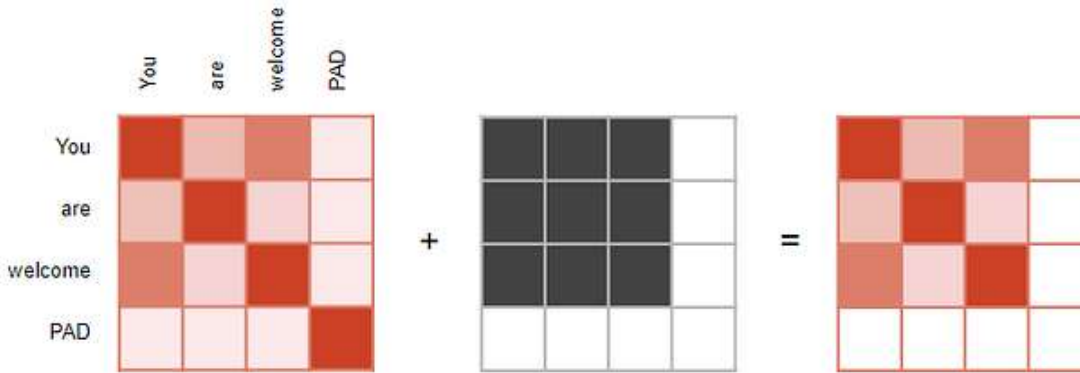
```
    variance = np.var(batch_of_images, keepdims=True, axis=(0, 1, 2))
```

```
    return (batch_of_images - mean) / variance
```

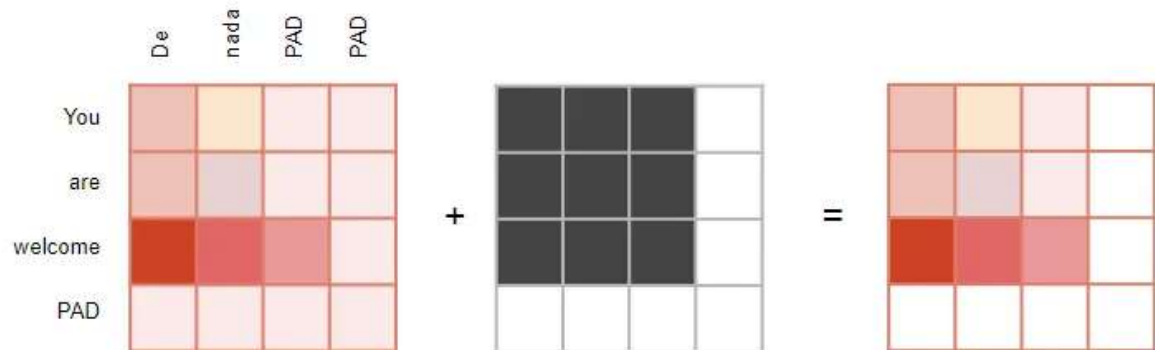
- While `BatchNormalization` collects information from many samples to obtain accurate statistics for the feature means and variances, `LayerNormalization` pools data within each sequence separately, which is more appropriate for sequence data

# Elements of the Stack, Masked Multi-Head Attention

- In the **Encoder Self-attention** and in the **Encoder-Decoder-attention** masking serves to zero attention outputs where there is padding in the input sentences, to ensure that padding doesn't contribute to the self-attention. Input sequences could be of different lengths, they are extended with padding tokens like in most NLP applications so that fixed-length vectors can be input to the Transformer.



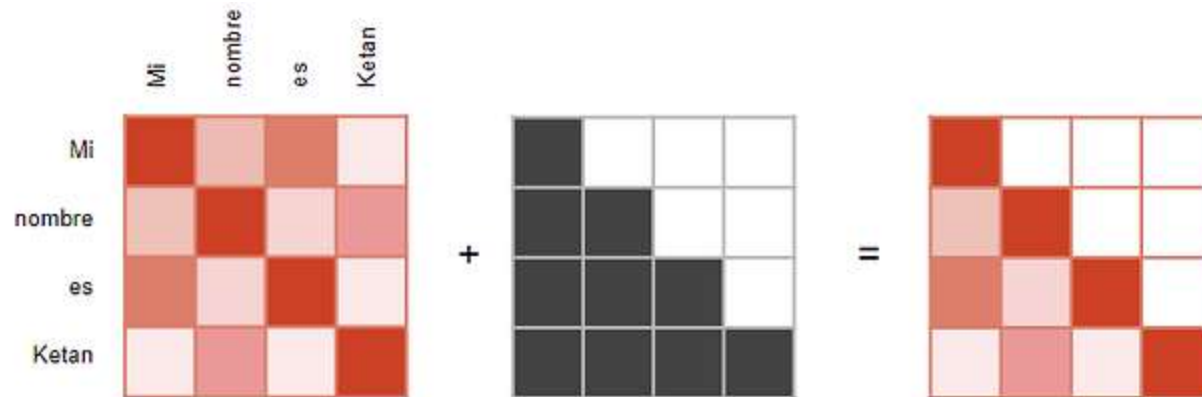
Encoder Self-Attention Scores



Encoder-Decoder Attention Scores

# Elements of the Stack, Masked Multi-Head Attention

- The decoder's *Masked Multi-Head Attention* layer does similar transformations as the other decoder's multi-head attention layer with one important difference.
- As translation of new words in the target language is gradually refined, we must make sure that the attention scan considers only previously discovered words. This is achieved with the *casual multi-head attention* or *masked multi-head attention*.
- **In the Decoder Self-attention:** masking serves to prevent the decoder from 'peeking' ahead at the rest of the target sentence when predicting the next word. Therefore, the Decoder masks out input words that appear later in the sequence.



Decoder Self-Attention Scores

# Keras version of Positional Embedding

- The idea behind positional encoding is very simple: to give the model access to word order information, we will add the word's position in the sentence to each word embedding. Our input word embeddings will have two components: the usual word vector, which represents the word independently of any specific context, and a position vector, which represents the position of the word in the current sentence. Hopefully, the model will then figure out how to best leverage this additional information.
- The simplest scheme you could come up with would be to concatenate the word's position to its embedding vector. You add a “position” axis to the vector and fill it with 0 for the first word in the sequence, 1 for the second, and so on.
- That may not be ideal, however, because the positions can potentially be very large
- integers, which will disrupt the range of values in the embedding vector. Neural networks don't like very large input values, or discrete input distributions.
- We will do something simpler and more effective: we will learn position embedding vectors the same way we learn to embed word indices. We'll then proceed to add our position embeddings to the corresponding word embeddings, to obtain a position-aware word embedding. This technique is called “positional embedding.”
- A downside of position embeddings is that the `sequence_length` needs to be known in advance



# Positional embedding as a subclassed layer

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding( # Embedding layer for token indices
            input_dim=input_dim, output_dim=output_dim)
        self.position_embeddings = layers.Embedding(# Embedding layer for token positions
            input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions # Add both vectors together

    def compute_mask(self, inputs, mask=None):# Method called automatically by Keras
        return tf.math.not_equal(inputs, 0) # the mask propagated to next layer

    def get_config(self):
        config = super().get_config()
        config.update({
            "output_dim": self.output_dim,
```

# Signal Processing Positional Embedding

- We can create `PositionalEmbedding` layer in Keras. For efficiency reasons, we precompute the positional embedding matrix in the constructor (so we need to know the maximum sentence length, `max_steps`, and the number of dimensions for each word representation, `max_dims`).
- Then, the `call()` method crops this embedding matrix to the size of the inputs, and it adds it to the inputs. Since we added an extra first dimension of size 1 when creating the positional embedding matrix, the rules of broadcasting will ensure that the matrix gets added to every sentence in the inputs:

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims must be even
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**(2 * i / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**(2 * i / max_dims)).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

# Positional Embeddings

- Instead of using trainable positional encodings, the authors of the 2017 Transformer paper chose to use fixed positional encodings, based on the sine and cosine functions at different frequencies. They added to the word embeddings a vector containing values in the range  $[-1, 1]$  that varied cyclically depending on the position (it used sine and cosine functions to achieve this). This trick offers a way to uniquely characterize any integer in a large range via a vector of small values.
- A positional embedding is a dense vector that encodes the position of a word within a sentence: positional embedding for position  $i$  is simply added to the word embedding of the  $i^{\text{th}}$  word in the sentence. These positional embeddings can be learned by the model, but in the paper the authors preferred to use fixed positional embeddings, defined using the *sine* and *cosine* functions of different frequencies. The elements of positional embedding matrix  $\mathbf{P}$  are defined as:

$$P_{p,2i} = \sin(p/10000^{2i/d}) \quad \text{and} \quad P_{p,2i+1} = \cos(p/10000^{2i/d})$$

- Here  $P_{p,i}$  is the  $i^{\text{th}}$  component of the embedding for the word located at the  $p^{\text{th}}$  position in the sentence. Notice that 2017 paper, even positions are encoded with  $\sin()$  and odd positions with  $\cos()$  function.

# Clock like positional embeddings

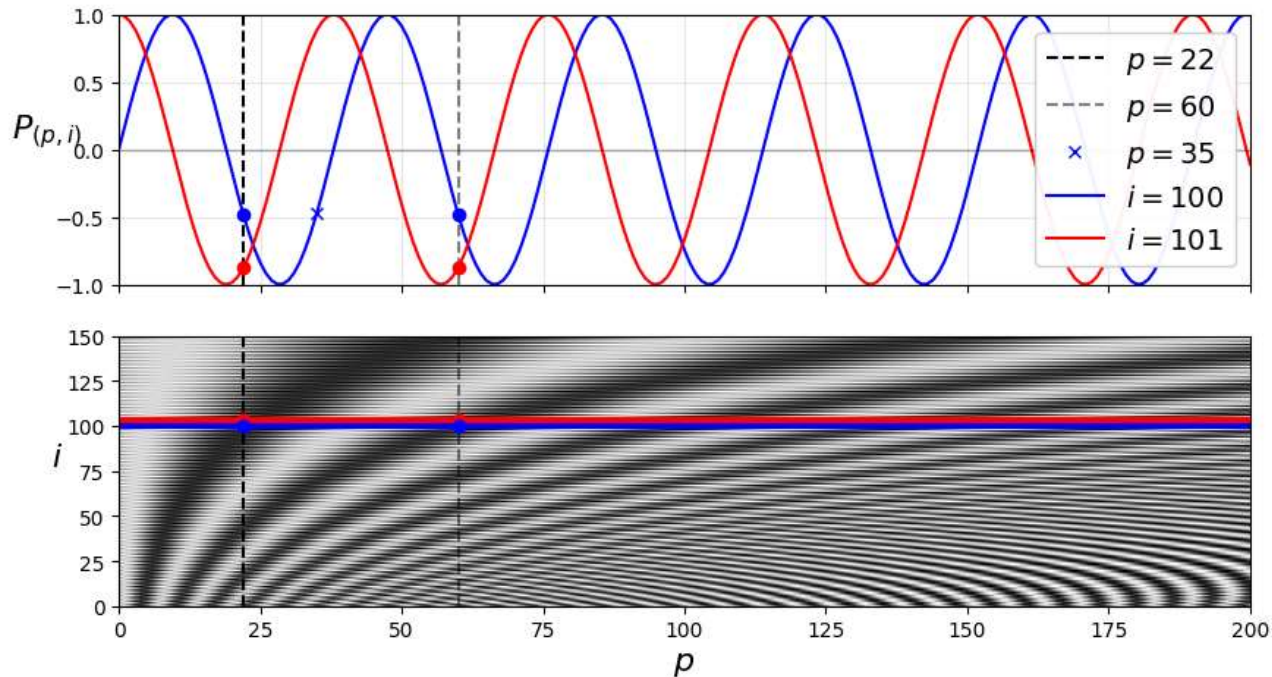
- There is an analogy with an (analogical) clock: the seconds' hand makes a full rotation with a frequency of 1/60 Hz (once every minute). Hence, every “point in time” inside a minute can be distinguished by looking at the hand, but two time instants in general can only be identified modulo 60 seconds.
- We overcome this in a clock by adding a separate hand (the minute hand) that rotates with a much slower frequency of 1/3600 Hz. Hence, by looking at the pair of coordinates (second, minute) (the “embedding” of time) we can distinguish any point inside an hour.
- Adding yet another hand with an even slower frequency (the hour hand with frequency 1/86400 Hz) we can distinguish any point inside a day.
- This can be generalized: we could design clocks with lower or higher frequencies to distinguish months, years, or milliseconds.
- A similar strategy can be applied here: we can distinguish each position  $i$  by encoding it through a set of  $d$  sines (with  $d$  a hyper-parameter) of increasing frequencies:

$$S_i = [\sin(\omega_1 i), \sin(\omega_2 i), \dots, \sin(\omega_d i)]$$

- In practice, the original proposal from [2017] uses only  $d/2$  possible frequencies, but adds both sines and cosines:

$$S_i = \sin(\omega_1 i), \cos(\omega_1 i), \dots, \sin(\omega_{d/2} i), \cos(\omega_{d/2} i)$$

# Non-trainable Positional Encoding



- This solution can give the same performance as trainable positional encodings, and it can extend to arbitrarily long sentences without adding any parameters to the model. When there is a large amount of pretraining data, trainable positional encodings are favored.
- After these positional encodings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional encoding for each position (e.g., the positional encoding for the word located at  $p=22$  position in a sentence is represented by the vertical dashed line at the top left the figure above, and you can see that it is unique to that position).
- This positional encoding makes it possible for the model to learn relative positions as well.

# Masking

- Mask all the pad tokens in the batch of sequence. It ensures that the model does not treat padding as the input. The mask indicates where pad value 0 is present: it outputs a 1 at those locations, and a 0 otherwise.

```
def create_padding_mask(seq):  
    seq = tf.cast(tf.math.equal(seq, 0), tf.float32)  
  
    # add extra dimensions to add the padding  
    # to the attention logits.  
    return seq[:, tf.newaxis, tf.newaxis, :] # (batch_size, 1, 1, seq_len)  
  
x = tf.constant([[7, 6, 0, 0, 1], [1, 2, 3, 0, 0], [0, 0, 0, 4, 5]])  
create_padding_mask(x)  
<tf.Tensor: shape=(3, 1, 1, 5), dtype=float32, numpy=  
array([[[[0., 0., 1., 1., 0.]],  
  
        [[0., 0., 0., 1., 1.]],  
  
        [[1., 1., 1., 0., 0.]])], dtype=float32)>
```

# The look-ahead mask

- The look-ahead mask is used to mask the future tokens in a sequence. In other words, the mask indicates which entries should not be used.
- This means that to predict the third word, only the first and second word will be used. Similarly, to predict the fourth word, only the first, second and the third word will be used and so on.

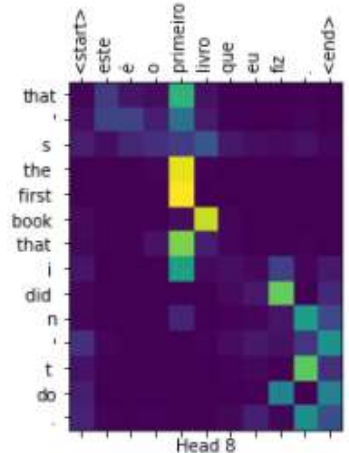
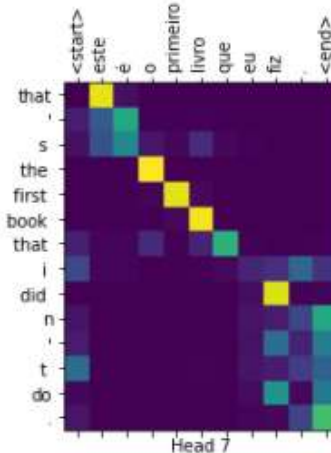
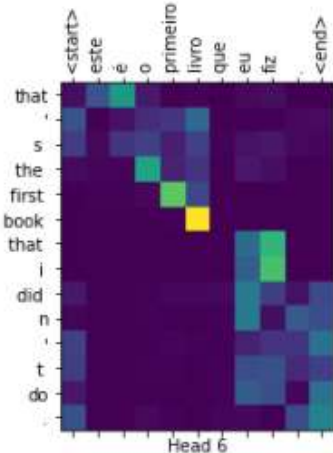
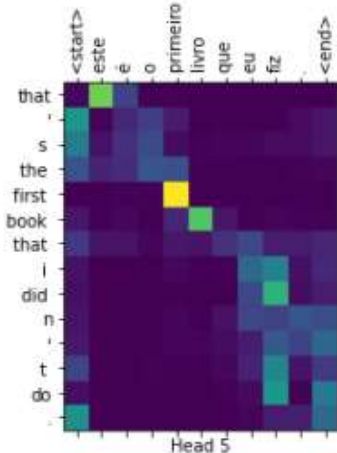
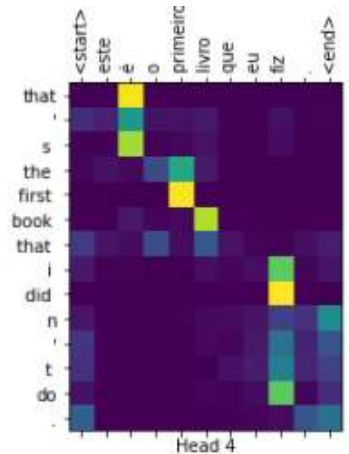
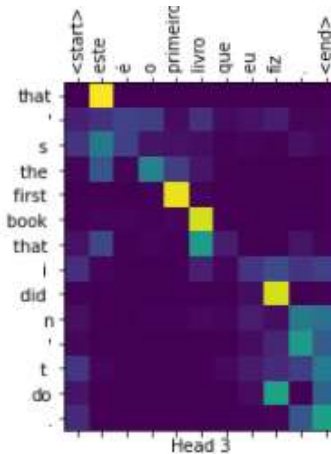
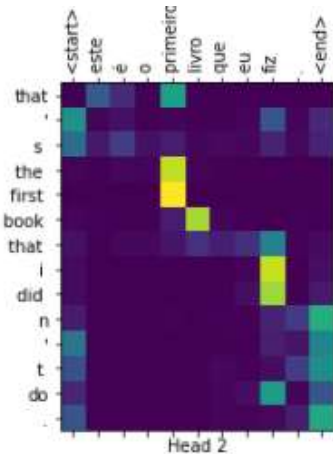
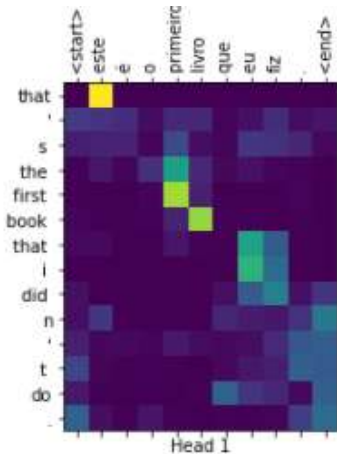
```
def create_look_ahead_mask(size):  
    mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)  
    return mask # (seq_len, seq_len)  
x = tf.random.uniform((1, 3))  
temp = create_look_ahead_mask(x.shape[1])  
temp
```

```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=  
array([[0., 1., 1.],  
       [0., 0., 1.],  
       [0., 0., 0.]], dtype=float32)>
```

# Attention in different blocks

- You can pass different layers and attention blocks of the decoder to the plot parameter.

```
translate("este é o primeiro livro que eu fiz.", plot='decoder_layer4_block2')
```

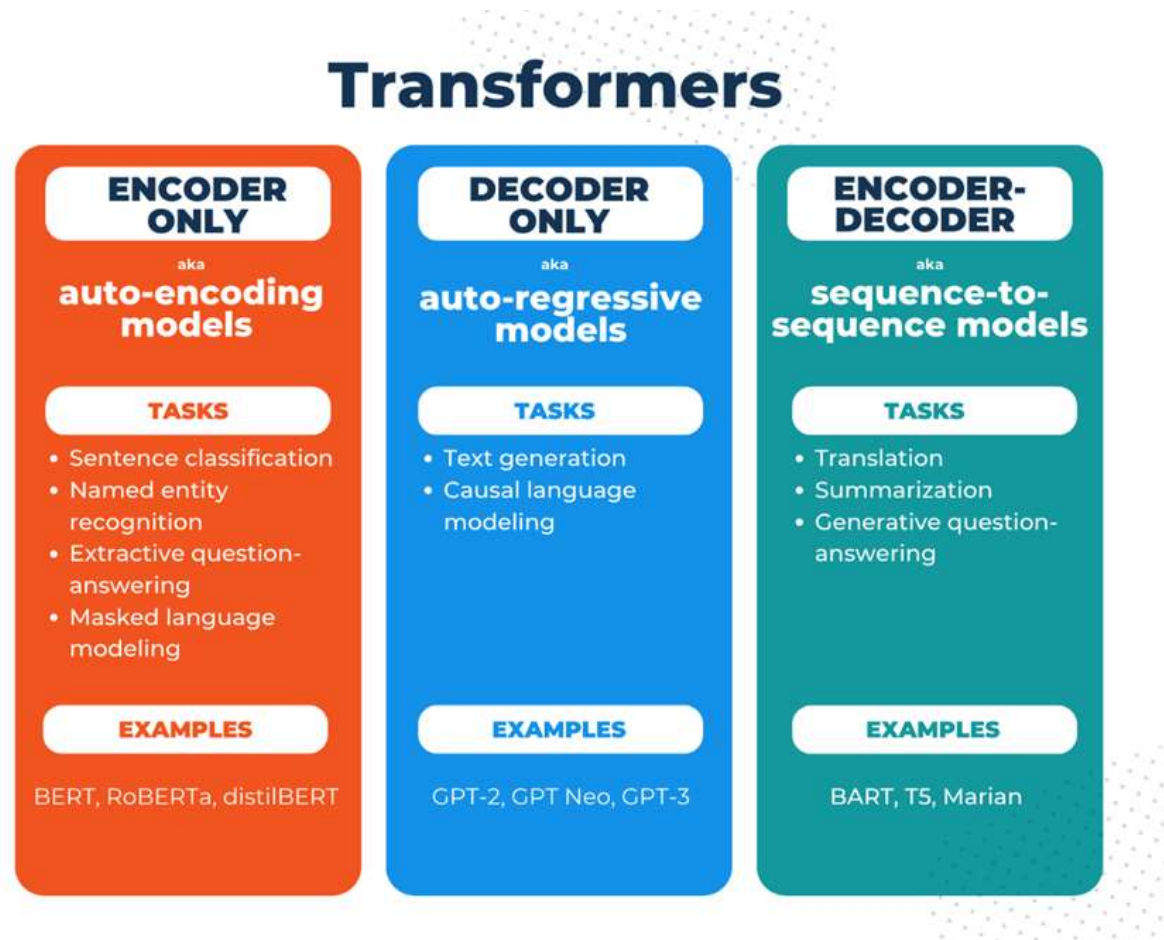


Real translation: this is the first book i've ever done.



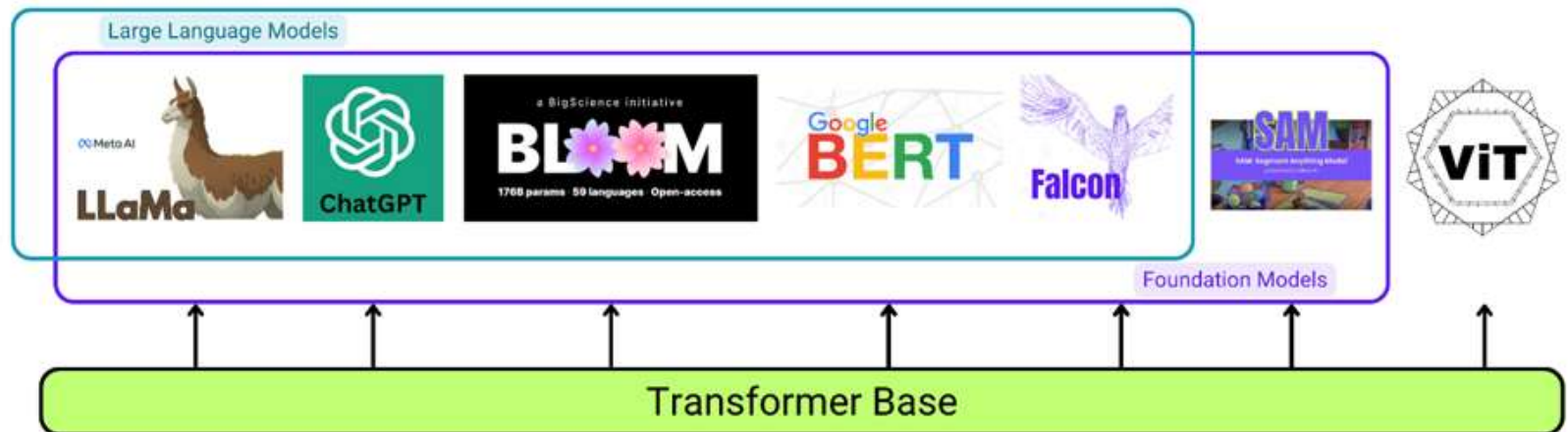
# Use Cases for Transformers

- Transformers are made up of encoders and decoders, and the tasks we can perform with them depend on whether we use either or both of these components. .



# Transformers in LLMs

- Large Language Models (LLMs) like ChatGPT or LLaMA use the transformer architecture as the fundamental building block for self-supervised learning on vast amounts of unlabelled data.
- These models are also sometimes referred to as “foundation models” because they tend to generalize well to a wide range of tasks, and in some cases are also available for more specific fine-tuning. Not all foundation models are transformer based, though the majority are.



# Transformers on HuggingFace.co

- Site HuggingFace.co contains implementations of many kinds of transformers in major APIs: Keras, PyTorch and JAX. In general, hugging Face contains large variety of code and algorithms for state-of-the-art machine learning for PyTorch, TensorFlow, and JAX.
- 🤗 Transformers provides APIs and tools to easily download and train state-of-the-art pretrained models. Using pretrained models can reduce your compute costs, carbon footprint, and save you the time and resources required to train a model from scratch. These models support common tasks in different modalities, such as:
  - 📄 Natural Language Processing: text classification, named entity recognition, question answering, language modeling, summarization, translation, multiple choice, and text generation.
  - 🖼️ Computer Vision: image classification, object detection, and segmentation.
  - 🎧 Audio: automatic speech recognition and audio classification.
  - 🌸 Multimodal: table question answering, optical character recognition, information extraction from scanned documents, video classification, and visual question answering.
- 🤗 Transformers support framework interoperability between PyTorch, TensorFlow, and JAX. This provides the flexibility to use a different framework at each stage of a model's life; train a model in three lines of code in one framework, and load it for inference in another. Models can also be exported to a format like ONNX and TorchScript for deployment in production environments.

# Translation with Sequence-to-Sequence Transformer

# English to Spanish Translation with Transformers

- In this example, we will build a sequence-to-sequence Transformer model, which we'll train on an English-to-Spanish machine translation task.
- We will learn how to:
  - Vectorize text using the Keras TextVectorization layer.
  - Implement a TransformerEncoder layer, a TransformerDecoder layer, and a PositionalEmbedding layer.
  - Prepare data for training a sequence-to-sequence model.
  - Use the trained model to generate translations of never-seen-before input sentences (sequence-to-sequence inference).
  - The code featured here is adapted from the book Deep Learning with Python, Second Edition (chapter 11: Deep learning for text).

# Setup & Data Download

- We import the essential packages:

```
import pathlib
import random
import string
import re
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers import TextVectorization
```

- Download the English-to-Spanish translation dataset provided by [Anki](#).

```
text_file = keras.utils.get_file(
    fname="spa-eng.zip",
    origin="http://storage.googleapis.com/download.tensorflow.org/data/spa-
eng.zip",
    extract=True,
)
text_file = pathlib.Path(text_file).parent / "spa-eng" / "spa.txt"
```

# Parsing the Data

- Each line contains an English sentence and its corresponding Spanish sentence. The English sentence is the source sequence and Spanish one is the target sequence. We prepend the token "[start]" and we append the token "[end]" to the Spanish sentence.

```
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    eng, spa = line.split("\t")
    spa = "[start] " + spa + " [end]"
    text_pairs.append((eng, spa))
```

- Here's what our sentence pairs look like:

```
for _ in range(5):
    print(random.choice(text_pairs))
```

```
('How is your daughter?', '[start] ¿Cómo está tu hija? [end]')
('She had some cookies to stay her hunger until dinner.', '[start] Toma algunas galletas para calmar el hambre hasta la cena. [end]')
('Leave tomorrow.', '[start] Sal mañana. [end]')
('It may have rained a little last night.', '[start] Puede que anoche lloviera un poco. [end]')
('I'll bring you the bill immediately.', '[start] Te traigo la cuenta en un segundo. [end]')
```

# Splitting data into Training, Validation and Test set

- Next, we split the sentence pairs into a training set, a validation set, and a test set.

```
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples : num_train_samples +
num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples :]

print(f"{len(text_pairs)} total pairs")
print(f"{len(train_pairs)} training pairs")
print(f"{len(val_pairs)} validation pairs")
print(f"{len(test_pairs)} test pairs")
```

```
118964 total pairs
83276 training pairs
17844 validation pairs
17844 test pairs
```



# Vectorizing the text data

- We need two instances of the `TextVectorization` layer to vectorize the text data (one for English and one for Spanish).
- `TextVectorization` layers will turn the original string sentences into integer sequences where each integer represents the index of a word in a vocabulary.
- The English layer will use the default string standardization (strip punctuation characters) and splitting scheme (split on whitespace).
- The Spanish layer will use a custom standardization, where we add the character "¿" to the set of punctuation characters to be stripped.
- Note: in a production-grade machine translation model, it is not recommended to strip the punctuation characters in either language. Instead, we turn each punctuation character into its own token. This can be achieved by providing a custom split function to the `TextVectorization` layer.

# Vectorizing the text data

```
strip_chars = string.punctuation + ";"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

vocab_size = 15000
sequence_length = 20
batch_size = 64

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(lowercase, "[%s]" % re.escape(strip_chars), "")

eng_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
spa_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)
train_eng_texts = [pair[0] for pair in train_pairs]
train_spa_texts = [pair[1] for pair in train_pairs]
eng_vectorization.adapt(train_eng_texts)
spa_vectorization.adapt(train_spa_texts)
```

# Format the Dataset

- Next, we format our datasets.
- At each training step, the model will seek to predict target words  $N+1$  (and beyond) using the source sentence and the target words 0 to  $N$ .
- As such, the training dataset will yield a tuple (`inputs`, `targets`), where:
- `inputs` is a dictionary with the keys `encoder_inputs` and `decoder_inputs`.
- `encoder_inputs` is the vectorized source sentence and `decoder_inputs` is the target sentence "so far", that is to say, the words 0 to  $N$  used to predict word  $N+1$  (and beyond) in the target sentence.
- `target` is the target sentence offset by one step: it provides the next words in the target sentence -- what the model will try to predict.

# Format the Dataset

```
def format_dataset(eng, spa):
    eng = eng_vectorization(eng)
    spa = spa_vectorization(spa)
    return (
        {
            "encoder_inputs": eng,
            "decoder_inputs": spa[:, :-1],
        },
        spa[:, 1:],
    )

def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset)
    return dataset.shuffle(2048).prefetch(16).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

# Shapes of Datasets

- Let's take a quick look at the sequence shapes (we have batches of 64 pairs, and all sequences are 20 steps long):

```
for inputs, targets in train_ds.take(1):  
    print(f'inputs["encoder_inputs"].shape:  
{inputs["encoder_inputs"].shape}')  
    print(f'inputs["decoder_inputs"].shape:  
{inputs["decoder_inputs"].shape}')  
    print(f"targets.shape: {targets.shape}")
```

```
inputs["encoder_inputs"].shape: (64, 20)  
inputs["decoder_inputs"].shape: (64, 20)  
targets.shape: (64, 20)
```

# Building the Model

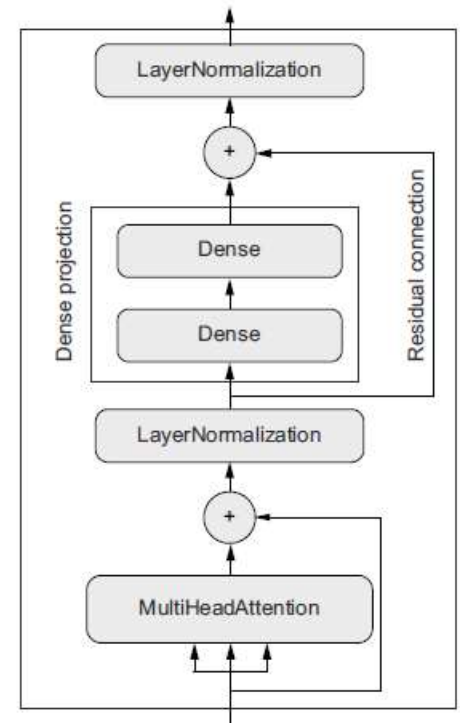
- Our sequence-to-sequence Transformer consists of a `TransformerEncoder` and a `TransformerDecoder` chained together. To make the model aware of word order, we also use a `PositionalEmbedding` layer.
- The source sequence will be pass to the `TransformerEncoder`, which produces a new representation of the sequence. This new representation is passed to the `TransformerDecoder`, together with the target sequence so far (target words 0 to N).
- The `TransformerDecoder` seeks to predict the next words in the target sequence (N+1th and beyond).
- A key detail that makes this possible is `causal masking` (`use_causal_mask=True` in the first attention layer of the `TransformerDecoder`). The `TransformerDecoder` sees the entire sequences at once, and thus we must make sure that it only uses information from target tokens 0 to N when predicting token N+1 (otherwise, it could use information from the future, which would result in a model that cannot be used at inference time).
- Both `TransformerEncoder` and `TransformerDecoder` are implemented as subclasses of class `layers.Layer`

# class TransformerEncoder

```
class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim # size of the embedding vectors
        self.dense_dim = dense_dim # size of the inner dense layer
        self.num_heads = num_heads # number of attention heads
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.dense_proj = keras.Sequential(
            [
                layers.Dense(dense_dim, activation="relu"),
                layers.Dense(embed_dim),
            ]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.supports_masking = True

    def call(self, inputs, mask=None): # Computation goes in call()
        if mask is not None:
            # The mask generated by Embedding layer is 2D. The attention
            mask = mask[:, tf.newaxis, :] # layer expect 3D or 4D, so we expand its rank
        attention_output = self.attention(
            inputs, inputs, attention_mask=mask
        )
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self): # Implement serialization s we can save the model
        config = super().get_config()
        config.update(
            {
                "embed_dim": self.embed_dim,
                "dense_dim": self.dense_dim,
                "num_heads": self.num_heads,
            }
        )
        return config
```



# class PositionalEmbedding

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(
            input_dim=vocab_size, output_dim=embed_dim
        )
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=embed_dim
        )
        self.sequence_length = sequence_length
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return tf.math.not_equal(inputs, 0)

    def get_config(self):
        config = super().get_config()
        config.update(
            {
                "sequence_length": self.sequence_length,
                "vocab_size": self.vocab_size,
                "embed_dim": self.embed_dim,
            }
        )
        return config
```



## Note: Saving Custom Layers

- When you write custom layers, make sure to implement the `get_config` method: this enables the layer to be reinstantiated from its config dict, which is useful during model saving and loading. The method should return a Python dict that contains the values of the constructor arguments used to create the layer.

- All Keras layers can be serialized and deserialized as follows:

```
config = layer.get_config()
new_layer = layer.__class__.from_config(config)
```

- The config does not contain weight values, so all weights in the layer get initialized from scratch.

- For instance:

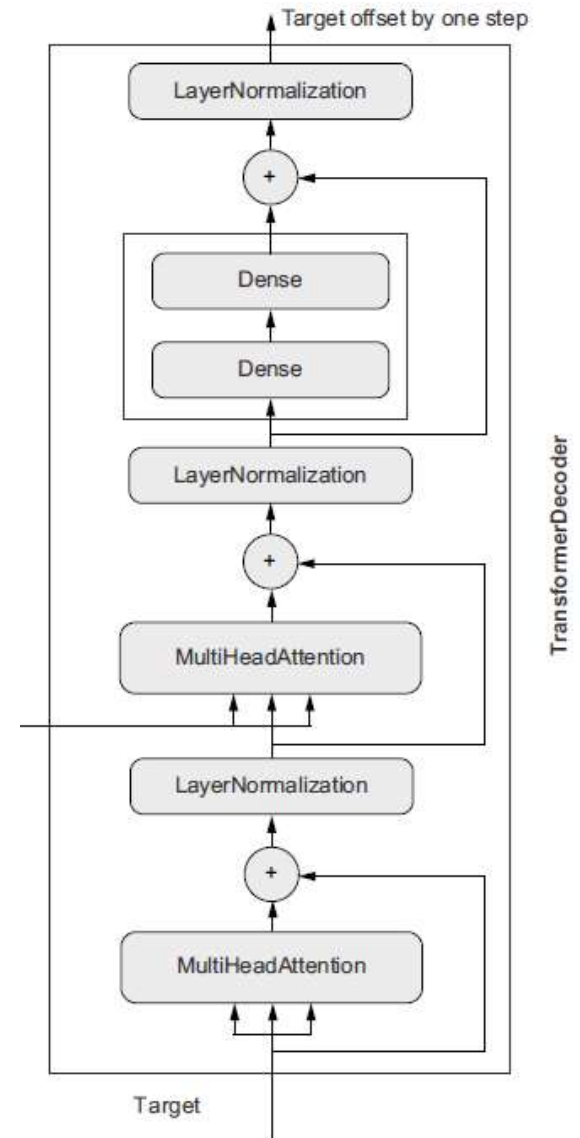
```
layer = PositionalEmbedding(sequence_length, input_dim,
output_dim)
config = layer.get_config()
new_layer = PositionalEmbedding.from_config(config)
```

- When saving a model that contains custom layers, the savefile will contain these config dicts. When loading the model from the file, you should provide the custom layer classes to the loading process, so that it can make sense of the config objects:

```
model = keras.models.load_model(
filename, custom_objects={"PositionalEmbedding": PositionalEmbedding})
```

# class TransformerDecoder

- The `TransformerDecoder` is similar to the `TransformerEncoder`, except it features an additional attention block where the keys and values are the source sequence encoded by the `TransformerEncoder`.
- Together, the encoder and the decoder form an end-to-end Transformer.



# class TransformerDecoder

```
class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, latent_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.latent_dim = latent_dim
        self.num_heads = num_heads
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.dense_proj = keras.Sequential(
            [
                layers.Dense(latent_dim, activation="relu"),
                layers.Dense(embed_dim),
            ]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.add = layers.Add() # instead of `+` to preserve mask
        self.supports_masking = True
```

- The last attribute `supports_masking` ensures that the layer will propagate its input mask to its outputs; masking in Keras is explicitly opt-in. If you pass a mask to a layer that doesn't implement `compute_mask()` and that doesn't expose this `supports_masking` attribute, that's an error.

# class TransformerDecoder, continued

```
def call(self, inputs, encoder_outputs, mask=None):
    attention_output_1 = self.attention_1(
        query=inputs, value=inputs, key=inputs, use_causal_mask=True
    )
    out_1 = self.layer_norm_1(self.add([inputs, attention_output_1]))

    attention_output_2 = self.attention_2(
        query=out_1,
        value=encoder_outputs,
        key=encoder_outputs,
    )
    out_2 = self.layer_norm_2(self.add([out_1, attention_output_2]))

    proj_output = self.dense_proj(out_2)
    return self.layer_norm_3(self.add([out_2, proj_output]))

def get_config(self):
    config = super().get_config()
    config.update(
        {
            "embed_dim": self.embed_dim,
            "latent_dim": self.latent_dim,
            "num_heads": self.num_heads,
        }
    )
    return config
```

# Assemble the Model

- Next, we assemble the end-to-end model.

```
embed_dim = 256
latent_dim = 2048
num_heads = 8
```

```
encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="encoder_inputs")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, latent_dim, num_heads)(x)
encoder = keras.Model(encoder_inputs, encoder_outputs)
```

```
decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="decoder_inputs")
encoded_seq_inputs = keras.Input(shape=(None, embed_dim),
name="decoder_state_inputs")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, encoded_seq_inputs)
x = layers.Dropout(0.5)(x)
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
decoder = keras.Model([decoder_inputs, encoded_seq_inputs], decoder_outputs)
```

```
decoder_outputs = decoder([decoder_inputs, encoder_outputs])
transformer = keras.Model(
    [encoder_inputs, decoder_inputs], decoder_outputs, name="transformer"
)
```

# call() Method

- The `call()` method is almost a straightforward rendering of the connectivity diagram from slide 58.
- There is an additional detail we need to take into account: causal padding. Causal padding is absolutely critical to successfully training a sequence-to-sequence Transformer. Unlike an RNN, which looks at its input one step at a time, and thus will only have access to steps  $0 \dots N$  to generate output step  $N$  (which is token  $N+1$  in the target sequence), the `TransformerDecoder` is order-agnostic: it looks at the entire target sequence at once.
- If it were allowed to use its entire input, it would simply learn to copy input step  $N+1$  to location  $N$  in the output. The model would thus achieve perfect training accuracy, but of course, when running inference, it would be completely useless, since input steps beyond  $N$  are not available.
- The fix is simple: we'll mask the upper half of the pairwise attention matrix to prevent the model from paying any attention to information from the future—only information from tokens  $0 \dots N$  in the target sequence should be used when generating target token  $N+1$ . To do this, we add a `get_causal_attention_mask(self, inputs)` method to our `TransformerDecoder` to retrieve an attention mask that we can pass to our `MultiHeadAttention` layers.

## TransformerDecoder.get\_causal\_attention\_mask()

```
def get_causal_attention_mask(self, inputs):
    input_shape = tf.shape(inputs)
    batch_size, sequence_length = input_shape[0], input_shape[1]
    i = tf.range(sequence_length)[:, tf.newaxis]
    j = tf.range(sequence_length)
    # Generate matrix of shape (sequence_length, sequence_length)
    # with 1s in one half and 0s in the other.
    mask = tf.cast(i >= j, dtype="int32")
    # Replicate it along the batch axis to get a matrix
    # of shape (batch_size, sequence_length, sequence_length).
    mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
    mult = tf.concat(
        [tf.expand_dims(batch_size, -1),
         tf.constant([1, 1], dtype=tf.int32)], axis=0)
    return tf.tile(mask, mult)
```

# Full call() method

```
def call(self, inputs, encoder_outputs, mask=None):  
    # Retrieve the casual mask  
    causal_mask = self.get_causal_attention_mask(inputs)  
    # Preparing the input mask describing padding locations in  
    # the target sequence  
    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,  
        # Pass the casual mask to the first attention layer, which performs  
        # self-attention over the target sequences  
        attention_mask=causal_mask)  
    attention_output_1 = self.layer_norm_1(inputs + attention_output_1)  
    attention_output_2 = self.attention_2(  
        query=attention_output_1,  
        value=encoder_outputs,  
        key=encoder_outputs, #Pass the combined mask to the second attention layer,  
        attention_mask=padding_mask, #which relates the source sequence to the target sequence  
    )  
    attention_output_2 = self.layer_norm_2(  
        attention_output_1 + attention_output_2)  
    proj_output = self.dense_proj(attention_output_2)  
    return self.layer_norm_3(attention_output_2 + proj_output)
```



# Training on a PC

- On my PC with TF 2.10 and a modest GPU card each epoch took some 60+ seconds.

```
Epoch 1/40
1302/1302 [=====] - 63s 46ms/step - loss: 0.8837 -
accuracy: 0.7236 - val_loss: 1.0336 - val_accuracy: 0.6504
Epoch 2/40
1302/1302 [=====] - 60s 46ms/step - loss: 0.8649 -
accuracy: 0.7290 - val_loss: 1.0135 - val_accuracy: 0.6604
Epoch 3/40
1302/1302 [=====] - 60s 46ms/step - loss: 0.8572 -
accuracy: 0.7318 - val_loss: 1.0219 - val_accuracy: 0.6581
Epoch 4/40
1302/1302 [=====] - 60s 46ms/step - loss: 0.8515 -
accuracy: 0.7343 - val_loss: 1.0313 - val_accuracy: 0.6603
0.6632
. . . . .
. . . . .
Epoch 37/40
1302/1302 [=====] - 60s 46ms/step - loss: 0.7269 -
accuracy: 0.7773 - val_loss: 1.1071 - val_accuracy: 0.6620
Epoch 38/40
1302/1302 [=====] - 60s 46ms/step - loss: 0.7240 -
accuracy: 0.7781 - val_loss: 1.1331 - val_accuracy: 0.6580
Epoch 39/40
1302/1302 [=====] - 60s 46ms/step - loss: 0.7223 -
accuracy: 0.7791 - val_loss: 1.1194 - val_accuracy: 0.6555
Epoch 40/40
1302/1302 [=====] - 60s 46ms/step - loss: 0.7196 -
accuracy: 0.7795 - val_loss: 1.1239 - val_accuracy: 0.6607
```

# Decoding Test Sentences

- Let us demonstrate how to translate brand new English sentences.
- We simply feed into the model the vectorized English sentence as well as the target token "[start]", then we repeatedly generated the next token, until we hit the token "[end]"

```
spa_vocab = spa_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = eng_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = spa_vectorization([decoded_sentence])[:, :-1]
        predictions = transformer([tokenized_input_sentence,
tokenized_target_sentence])

        sampled_token_index = np.argmax(predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token

    if sampled_token == "[end]":
        break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(30):
    input_sentence = random.choice(test_eng_texts)
    translated = decode_sequence(input_sentence)
```

# Results for trained model

- With a model trained with 30 epochs we would get translations similar to the following:

She handed him the money. [start] ella le pasó el dinero [end]

Tom has never heard Mary sing. [start] tom nunca ha oído cantar a mary [end]

Perhaps she will come tomorrow. [start] tal vez ella vendrá mañana [end]

I love to write. [start] me encanta escribir [end]

His French is improving little by little. [start] su francés va a [UNK] sólo un poco [end]

My hotel told me to call you. [start] mi hotel me dijo que te [UNK] [end]

## Appendix: Beam Search

# Statistical machine translation

- One could construct decoders relying on ideas from statistical machine translation.
- Consider a model that computes the probability  $P(\bar{s} | s)$  of a translation  $\bar{s}$  given the original sentence  $s$ . We want to pick the translation that has the best probability. In other words, we want

$$\bar{s} = \operatorname{argmax}_{\bar{s}} (P(\bar{s} | s))$$

- As the search space can be huge, we need to shrink its size.
- Here is a list of sequence model decoders (both good ones and bad ones).

**Exhaustive search** : this is the simplest idea.

- We compute the probability of every possible sequence, and we chose the sequence with the highest probability. However, this technique does not scale at all to large outputs as the search space is exponential in the size of the input. Decoding in this case is an NP-complete problem.

**Greedy Search** : At each time step, we pick the most probable token. In other words

$$x_t = \operatorname{argmax}_{\tilde{x}_t} P(\tilde{x}_t | x_1, \dots, x_n)$$

- This technique is efficient and natural, however it explores a small part of the search space and if we make a mistake at one time step, the rest of the sentence could be heavily impacted.

**Beam Search** described on the following slides is most effective and most frequently used.

# Beam Search

- Another popular technique that can greatly improve the performance of a translation model at inference time is beam search.
- Suppose you have trained an encoder–decoder model, and you use it to translate the sentence “I like soccer” to Spanish. You are hoping that it will output the proper translation “me gusta el fútbol”, but unfortunately it outputs “me gustan los jugadores”, which means “I like the players”. Looking at the training set, you notice many sentences such as “I like cars”, which translates to “me gustan los autos”, so it wasn’t absurd for the model to output “me gustan los” after seeing “I like”. In this case it was a mistake since “soccer” is singular.
- The model could not go back and fix it, so it tried to complete the sentence as best it could, in this case using the word “jugadores”.
- How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is beam search: it keeps track of a short list of the  $k$  most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the  $k$  most likely sentences. The parameter  $k$  is called the beam width.

# Beam Search, continued

- You translate the sentence "I like soccer" using beam search with a beam width of 3 (see Figure below). At the first decoder step, the model will output a probability for each possible first word in the translation. Let the top 3 words be "me" (75% estimated probability), "a" (3%), and "como" (1%). Next, the model finds the next word for each sentence. For the first sentence ("me"), perhaps the model outputs 36% for the word "gustan", 32% for the word "gusta", 16% for the word "encanta", and so on. These are conditional probabilities, given that the sentence starts with "me". For the second sentence ("a"), the model might give 50% for the word "mi", and so on. If vocabulary has 1,000 words, we end up with 1,000 probabilities per sentence.
- Next, we compute the probabilities of each of the 3,000 two-word sentences we considered ( $3 \times 1,000$ ). The estimated probability of the sentence "me" was 75%, while the estimated conditional probability of the word "gustan" (given that the first word is "me") was 36%, so the estimated probability of the sentence "me gustan" is  $75\% \times 36\% = 27\%$ . After computing the probabilities of 3,000 two-word sentences, we keep only the top 3. In this example they all start with the word "me": "me gustan" (27%), "me gusta" (24%), and "me encanta" (12%). Right now, the sentence "me gustan" is winning, but "me gusta" has not been eliminated.



## Beam Search, continued

- Then we repeat the same process: we use the model to predict the next word in each of these three sentences, and we compute the probabilities of all 3,000 three-word sentences we considered. Perhaps the top three are now “me gustan los” (10%), “me gusta el” (8%), and “me gusta mucho” (2%). At the next step we may get “me gusta el fútbol” (6%), “me gusta mucho el” (1%), and “me gusta el deporte” (0.2%). Notice that “me gustan” was eliminated, and the correct translation is now ahead. We boosted our encoder–decoder model’s performance without any extra training, simply by using it more wisely.
- We iterate over and over until they all candidate finish with an EOS token. The top translation is then returned (after removing its EOS token).
- Note: If  $p(S)$  is the probability of sentence  $S$ , and  $p(W|S)$  is the conditional probability of the word  $W$  given that the translation starts with  $S$ , then the probability of the sentence  $S' = \text{concat}(S, W)$  is  $p(S') = p(S) p(W|S)$ . As we add more words, the probability gets smaller and smaller. To avoid the risk of it getting too small, which could cause floating point precision errors, the function keeps track of log probabilities instead of probabilities: recall that  $\log(a \text{ and } b) = \log(a) + \log(b)$ , therefore  $\log(p(S')) = \log(p(S)) + \log(p(W|S))$ .



# Implementation of Beam Search

```
# A basic implementation of beam search

def beam_search(sentence_en, beam_width, verbose=False):
    X = np.array([sentence_en]) # encoder input
    X_dec = np.array(["startofseq"]) # decoder input
    y_proba = model.predict((X, X_dec))[0, 0] # first token's probas
    top_k = tf.math.top_k(y_proba, k=beam_width)
    top_translations = [ # list of best (log_proba, translation)
        (np.log(word_proba), text_vec_layer_es.get_vocabulary()[word_id])
        for word_proba, word_id in zip(top_k.values, top_k.indices)
    ]

    # Displays the top first words in verbose mode
    if verbose:
        print("Top first words:", top_translations)

    for idx in range(1, max_length):
        candidates = []
        for log_proba, translation in top_translations:
            if translation.endswith("endofseq"):
                candidates.append((log_proba, translation))
                continue # translation is finished, so don't try to extend it
            X = np.array([sentence_en]) # encoder input
            X_dec = np.array(["startofseq " + translation]) # decoder input
            y_proba = model.predict((X, X_dec))[0, idx] # last token's proba
            for word_id, word_proba in enumerate(y_proba):
                word = text_vec_layer_es.get_vocabulary()[word_id]
                candidates.append((log_proba + np.log(word_proba),
                                    f"{translation} {word}"))
        top_translations = sorted(candidates, reverse=True)[:beam_width]

    # Displays the top translation so far in verbose mode
    if verbose:
        print("Top translations so far:", top_translations)

    if all([tr.endswith("endofseq") for _, tr in top_translations]):
        return top_translations[0][1].replace("endofseq", "").strip()
```

# Demonstration of Beam Search

- We first run a bidirectional LSTM model without beam search:

```
# extra code - shows how the model making an error
sentence_en = "I love cats and dogs"
translate(sentence_en)
'me [UNK] los perros y los gatos'
# beam search can help
beam_search(sentence_en, beam_width=3, verbose=True)
1/1 [=====] - 0s 22ms/step
Top first words: [(-0.043498553, 'me'), (-3.8191085, '[UNK]'), (-5.126285, 'odio')]
1/1 [=====] - 0s 21ms/step
1/1 [=====] - 0s 21ms/step
1/1 [=====] - 0s 21ms/step
Top translations so far: [(-0.49746862, 'me [UNK]'), (-1.6989571, 'me encanta'), (-1.8915125, 'me
gustan')]
1/1 [=====] - 0s 22ms/step
. . . . .
Top translations so far: [(-0.99586815, 'me [UNK] los perros y los'), (-2.486811, 'me gustan los perros y
los'), (-2.559771, 'me [UNK] los perros como el')]
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 21ms/step
1/1 [=====] - 0s 21ms/step
Top translations so far: [(-1.0827374, 'me [UNK] los perros y los gatos'), (-2.7267046, 'me gustan los
perros y los gatos'), (-3.7065737, 'me [UNK] los perros como el perro')]
1/1 [=====] - 0s 21ms/step
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 21ms/step
Top translations so far: [(-1.0828006, 'me [UNK] los perros y los gatos endofseq'), (-2.7270784, 'me
gustan los perros y los gatos endofseq'), (-3.7065864, 'me [UNK] los perros como el perro endofseq')]
'me [UNK] los perros y los gatos'
```

- The correct translation is in the top 3 sentences found by beam search, but it's not the first. Since we're using a small vocabulary, the [UNK] token is quite frequent, so you may want to penalize it (e.g., divide its probability by 2 in the beam search function): this will discourage beam search from using it too much.

## Appendix: Convert Unicode to ASCII

# Download and Prepare the Dataset

- We'll use a language dataset provided by <http://www.manythings.org/anki/>
- English-Spanish pair is under Spanish-English line:

`http://www.manythings.org/anki/spa-eng.zip`

- This dataset contains language translation pairs in the format:

`May I borrow this book? ¿Puedo tomar prestado este libro?`

- After downloading the dataset, we take these steps to prepare the data:
  1. Add a *start* and *end* token to each sentence.
  2. Clean the sentences by removing special characters.
  3. Create a word index and reverse word index (dictionaries mapping from `word` → `id` and `id` → `word`).
  4. Pad each sentence to a maximum length

```
# Download the file
path_to_zip = tf.keras.utils.get_file( 'spa-eng.zip',
                                      origin='http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip',
                                      extract=True)

path_to_file = os.path.dirname(path_to_zip)+"/spa-eng/spa.txt"
```

# Covert Unicode Files to ASCII

- Run the following:

```
import unicodedata
def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s)
                   if unicodedata.category(c) != 'Mn')
```

This is rather mysterious.

Python documentation on Unicode states:

- The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).
- For each character, there are two normal forms: normal form C and normal form D. Normal form D (**NFD**) is also known as canonical decomposition and translates each character into its decomposed form. Normal form C (**NFC**) first applies a canonical decomposition, then composes pre-combined characters again.
- In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (e.g. gb2312).
- The normal form KD (NFKD) will apply the compatibility decomposition, i.e., replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.
- Even if two Unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn't, they may not compare as equal.

<https://www.compart.com/en/unicode/category>

Number of Entries:

29 [\[1\]](#)

Total Number of Characters:

137,439

Key	Name	Characters
Cc	Control	65
Cf	Format	152
Co	Private Use	0
Cs	Surrogate	0
Ll	Lowercase Letter	2145
Lm	Modifier Letter	250
Lo	Other Letter	121212
Lt	Titlecase Letter	31
Lu	Uppercase Letter	1781
Mc	Spacing Mark	415
Me	Enclosing Mark	13
Mn	Nonspacing Mark	1805
Nd	Decimal Number	610
Nl	Letter Number	236
No	Other Number	807

Key	Name	Characters
	Connector	
Pc	Punctuation	10
Pd	Dash Punctuation	24
Pe	Close Punctuation	73
Pf	Final Punctuation	10
Pi	Initial Punctuation	12
Po	Other Punctuation	584
Ps	Open Punctuation	75
Sc	Currency Symbol	57
Sk	Modifier Symbol	121
Sm	Math Symbol	948
So	Other Symbol	5984
Zl	Line Separator	1
Zp	Paragraph Separator	1
Zs	Space Separator	17

# Preprocess Sentences

```
def preprocess_sentence(w):
    w = unicode_to_ascii(w.lower().strip())
    # creating a space between a word and the punctuation following it
    # eg: "he is a boy." => "he is a boy ."
    w = re.sub(r"([?!.,:])", r" \1 ", w)
    w = re.sub(r'[" "]+' , " ", w)

    # replacing everything with space except (a-z, A-Z, ".", "?", "!", ",", ")
    w = re.sub(r"^[a-zA-Z?!.,:]+", " ", w)

    w = w.rstrip().strip()

    # adding a start and an end token to the sentence
    # so that the model know when to start and stop predicting.
    w = '<start> ' + w + ' <end>'
    return w

en_sentence = u"May I borrow this book?"
sp_sentence = u"¿Puedo tomar prestado este libro?"
print(preprocess_sentence(en_sentence))
print(preprocess_sentence(sp_sentence).encode('utf-8'))

<start> may i borrow this book ? <end>
<start> ¿ puedo tomar prestado este libro ? <end>
# 1. Remove the accents
# 2. Clean the sentences
# 3. Return word pairs in the format: [ENGLISH, SPANISH]

def create_dataset(path, num_examples):
    lines = io.open(path, encoding='UTF-8').read().strip().split('\n')

    word_pairs = [[preprocess_sentence(w) for w in l.split('\t')] for l in lines[:num_examples]]

    return zip(*word_pairs)

en, sp = create_dataset(path_to_file, None)
print(en[-1])
print(sp[-1])
```

<start> if you want to sound like a native speaker , you must be willing to practice saying the same sentence over and over in the same way that banjo players practice the same phrase over and over until they can play it correctly and at the desired tempo .  
<end>

<start> si quieres sonar como un hablante nativo , debes estar dispuesto a practicar diciendo la misma frase una y otra vez de la misma manera en que un musico de banjo practica el mismo fraseo una y otra vez hasta que lo puedan tocar correctamente y en el tiempo esperado . <end>