# Attention and transformers

Attention mechanisms; Transformers;

Examples with ***keras*** and ***tensorflow***

*Examples adapted from F. Chollet, "Deep Learning with Python"*

## Recent developments on sequence processing

Previously, we saw two types of methods that could help in processing sequential data:

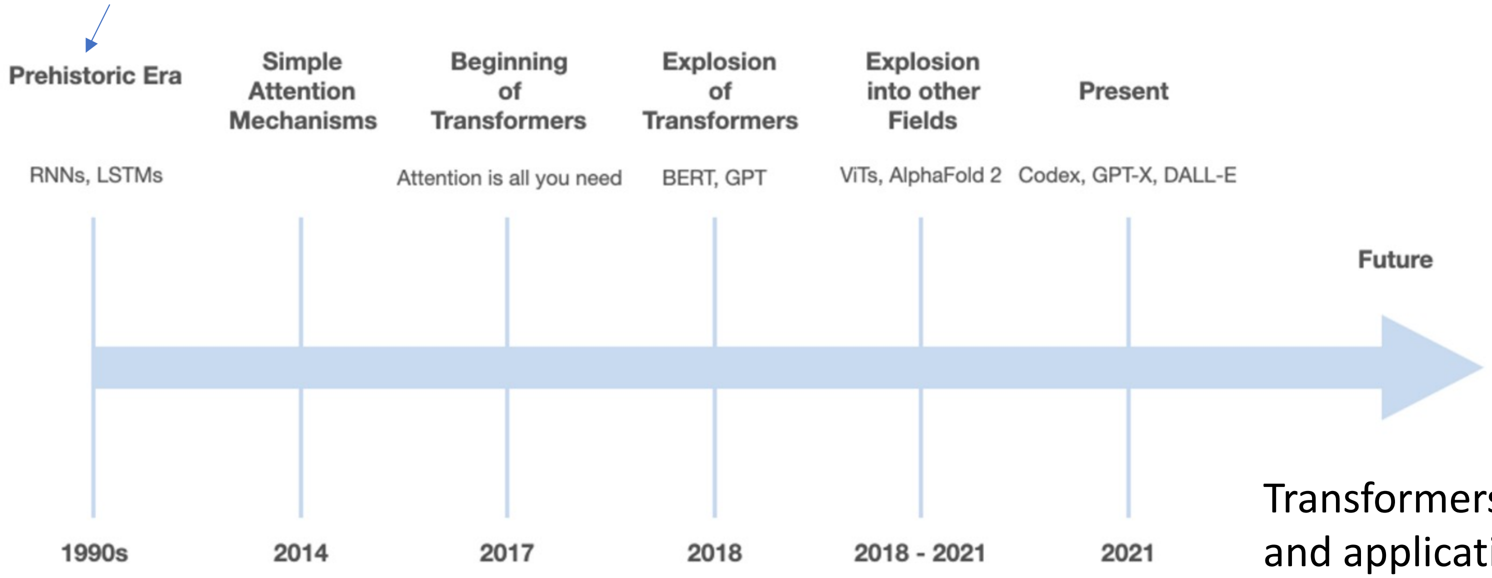
**Embeddings** – helped to create more compact representations of tokens (e.g. words) and with semantics (latent spaces with meaning); They launched the idea of representing concepts with compact number vectors by learning (with gradient descent) and showed that it can work well

**Recurrent networks** – allow to work on sequential data directly; create mechanisms of memory (in LSTMs of short and long term) to process the sequences... BUT ... can have difficulties with with long sequences and with the correct understanding of contexto in natural language.

*I was born and raised in* ***Portugal****. Later, I lived in France, Italy, Germany and the UK, and travelled to many parts of the world. I learned different cultures and habits, I met interesting people and even learned to speak some words in other languages. Still, the only language I speak fluently is* ***Portuguese****.*

## Timeline

A bit of an exaggeration... but things change fast in these fields... yet LSTMs dominated the field until at least 2018...



Prehistoric Era
Simple Attention Mechanisms
Beginning of Transformers
Explosion of Transformers
Explosion into other Fields
Present
RNNs, LSTMs
Attention is all you need
BERT, GPT
ViTs, AlphaFold 2
Codex, GPT-X, DALL-E
Future
1990s
2014
2017
2018
2018 - 2021
2021

Figure taken from the "Transformers United" course taught at Stanford
Available at [https://web.stanford.edu/class/cs25/](https://web.stanford.edu/class/cs25/)

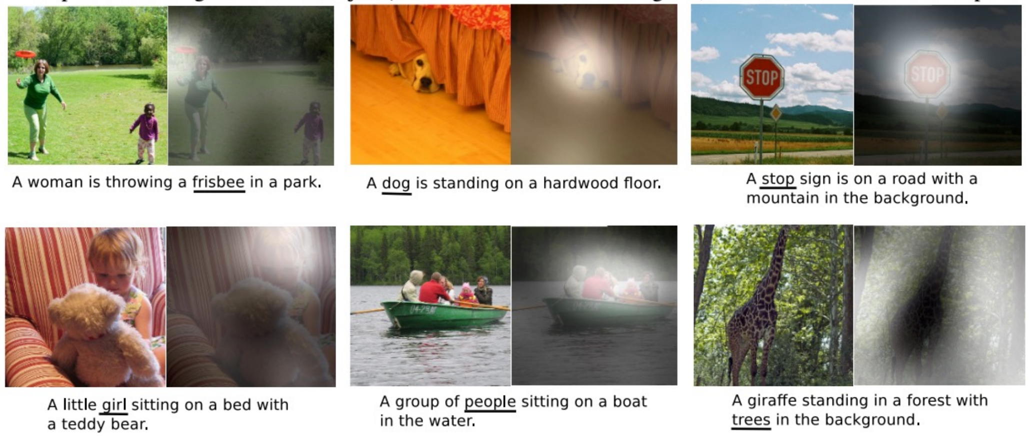
Transformers are used in many areas and applications today including vision, reinforcement learning, few-zero shot learning.
They are the basis of the latest LLMs and prompt systems such as ChatGPT and DALLE-2

## Simple attention mechanisms

Attention mechanism was proposed for images as a way to give different weights to different parts of the images, with two variants:

- ***Soft attention***: weights between [0,1] throughout the image
  - Differentiable models
  - But they involve heavy computations because they work on the whole image
- ***Hard attention***: weights only in some parts of the images
  - Non-differentiable models which made training difficult
  - But fewer calculations in inference

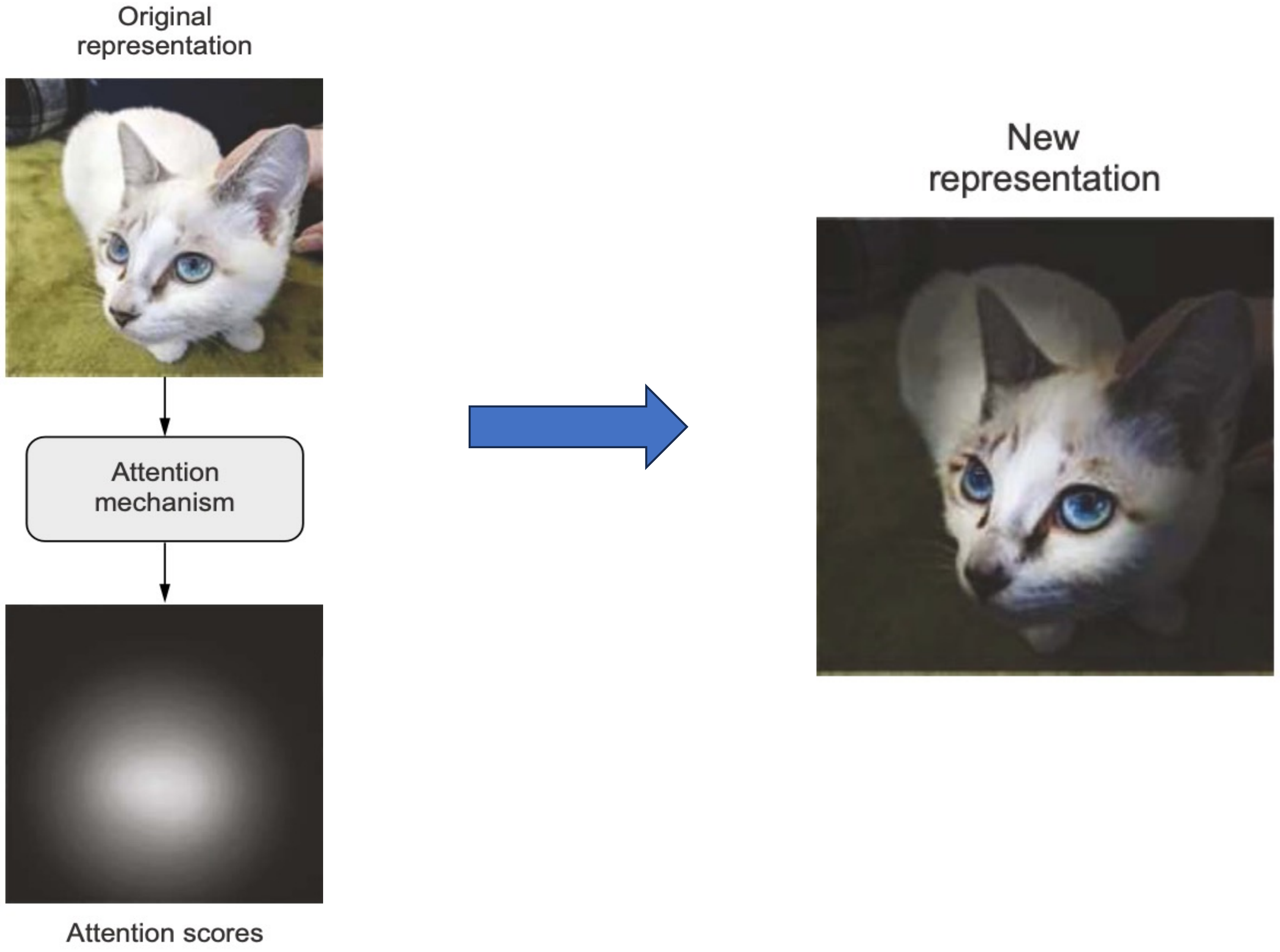
*Figure 4.* Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)


A woman is throwing a frisbee in a park.
A dog is standing on a hardwood floor.
A stop sign is on a road with a mountain in the background.
A little girl sitting on a bed with a teddy bear.
A group of people sitting on a boat in the water.
A giraffe standing in a forest with trees in the background.

https://lilianweng.github.io/posts/2018-06-24-attention

Paper: http://proceedings.mlr.press/v37/xuc15.pdf

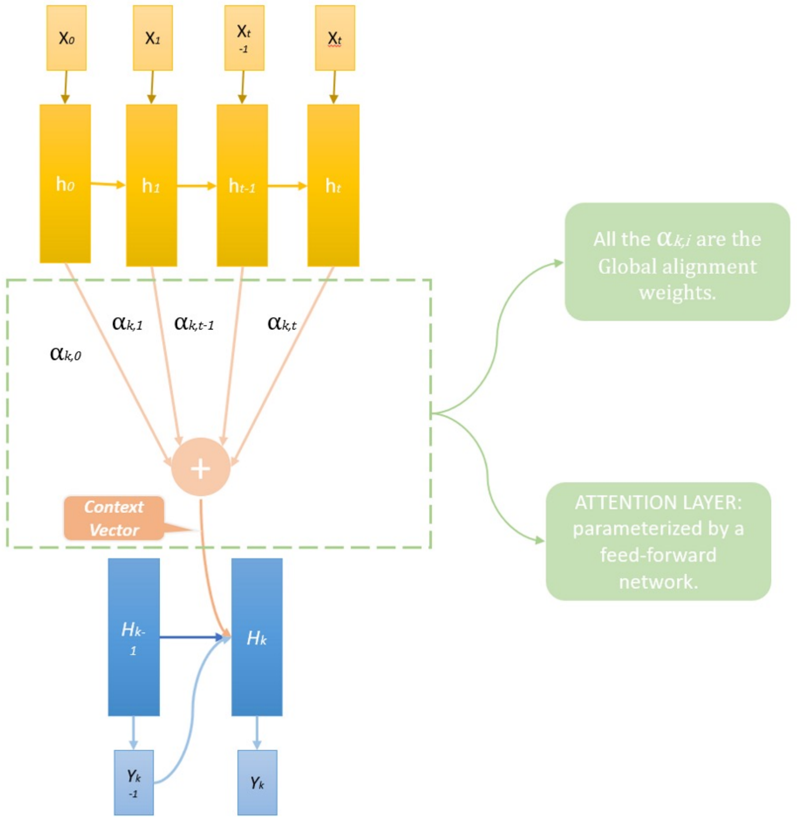
## Attention mechanism: illustration



F. Chollet

## Attention in texts: Neural machine translation

- Attention mechanism proposed for RNNs that implement machine translation as a way to overcome problems in decoders in representing the entire initial sequence processed in a single context vector
- Idea is to use the outputs of all intermediate steps in the decoder, with specific weights (learned in the training process)
- Allows to **identify more relevant parts of the sequence** when processing the input for the result, can be applied locally or globally



Paper - 2015: https://nlp.stanford.edu/pubs/emnlp15_attn.pdf
More details:
https://towardsdatascience.com/intuitive-understanding-of-attention-mechanism-in-deep-learning-6c9482aecf4f
https://towardsdatascience.com/attaining-attention-in-deep-learning-a712f93bdb1e

## Attention in texts

In texts, **attention mechanisms** are in a way similar to **TF-IDF normalization** which assigns importance scores to tokens based on how much information different tokens are likely to carry. Important tokens get boosted while irrelevant tokens get faded out.

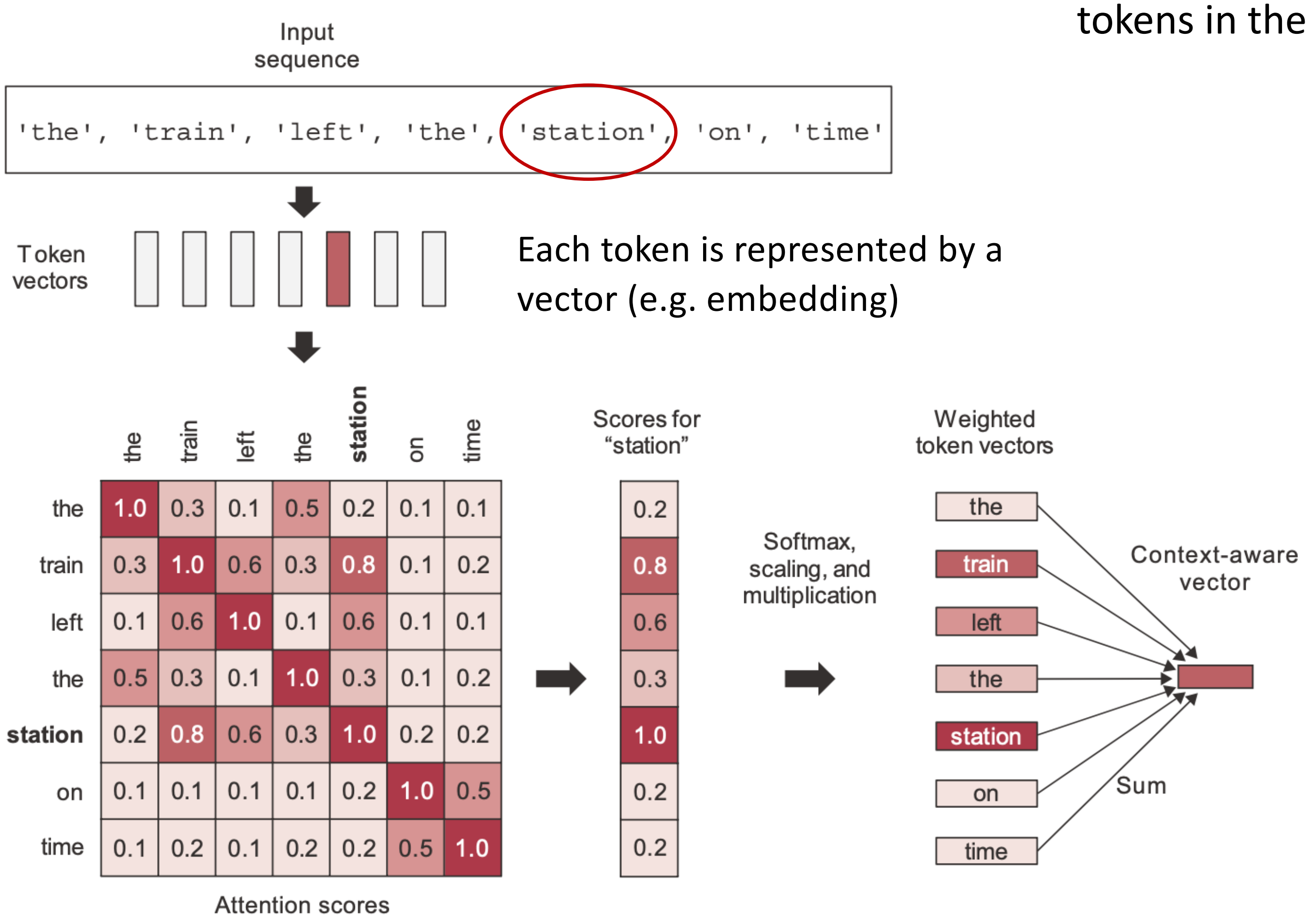
Another analogy is to look at latent representations (e.g. **embeddings**) which have a fixed numerical representation for each word. But, these do not depend on context

We can look at **attention mechanisms** in texts as **representations** that should depend on the **context** of the work in the text

*Peter and Mary went on a* ***date*** *but did not get along.*
*The project's final* ***date*** *is May 27th.*
*Peter did the like the* ***date*** *he had for desert since it was too sweet.*

# Self attention in texts

**Purpose**: produce context-aware token representations, by modulating the representation of a token by using the representations of related tokens in the sequence.



| | the | train | left | the | **station** | on | time |
|---|---|---|---|---|---|---|---|
| the | 1.0 | 0.3 | 0.1 | 0.5 | 0.2 | 0.1 | 0.1 |
| train | 0.3 | 1.0 | 0.6 | 0.3 | 0.8 | 0.1 | 0.2 |
| left | 0.1 | 0.6 | 1.0 | 0.1 | 0.6 | 0.1 | 0.1 |
| the | 0.5 | 0.3 | 0.1 | 1.0 | 0.3 | 0.1 | 0.2 |
| **station** | 0.2 | 0.8 | 0.6 | 0.3 | 1.0 | 0.2 | 0.2 |
| on | 0.1 | 0.1 | 0.1 | 0.1 | 0.2 | 1.0 | 0.5 |
| time | 0.1 | 0.2 | 0.1 | 0.2 | 0.2 | 0.5 | 1.0 |

| Scores for "station" |
|---|
| 0.2 |
| 0.8 |
| 0.6 |
| 0.3 |
| 1.0 |
| 0.2 |
| 0.2 |

Each token is represented by a vector (e.g. embedding)

**Step 1** **Step 2**

**Step 1:**
compute **attention scores –** similarity between the vector for the target word (in this case "station") and every other word in the sentence – use dot product between their representations

**Step 2**:
compute the **sum** of all word vectors in the sentence, weighted by attention scores, after **normalization and scaling**; resulting vector is the new representation for the target word, incorporating context

This process is repeated for all the words in the sentence

F. Chollet

## Self attention in texts

- The process we saw before can be summarized in:

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

- The generalized version of self-attention

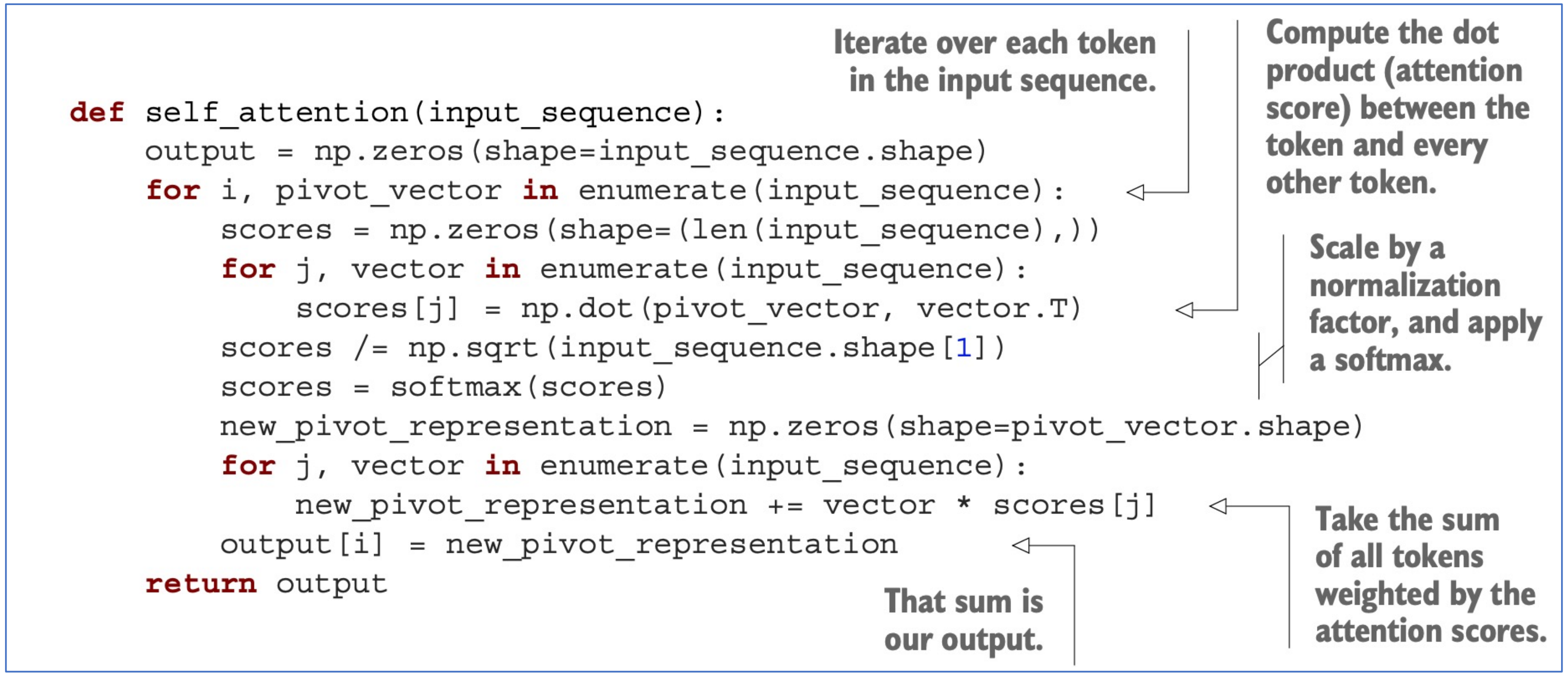
```
outputs = sum (values * pairwise_scores(query, keys))
```

For each element in the ***query***, compute how much the element is related to every ***key***, and use these scores to weight a sum of ***values***

In practice, the keys and the values are often the same sequence.

F. Chollet

## Self attention in texts - code



```python
def self_attention(input_sequence):
    output = np.zeros(shape=input_sequence.shape)
    for i, pivot_vector in enumerate(input_sequence):
        scores = np.zeros(shape=(len(input_sequence),))
        for j, vector in enumerate(input_sequence):
            scores[j] = np.dot(pivot_vector, vector.T)
        scores /= np.sqrt(input_sequence.shape[1])
        scores = softmax(scores)
        new_pivot_representation = np.zeros(shape=pivot_vector.shape)
        for j, vector in enumerate(input_sequence):
            new_pivot_representation += vector * scores[j]
        output[i] = new_pivot_representation
    return output
```

Iterate over each token in the input sequence.

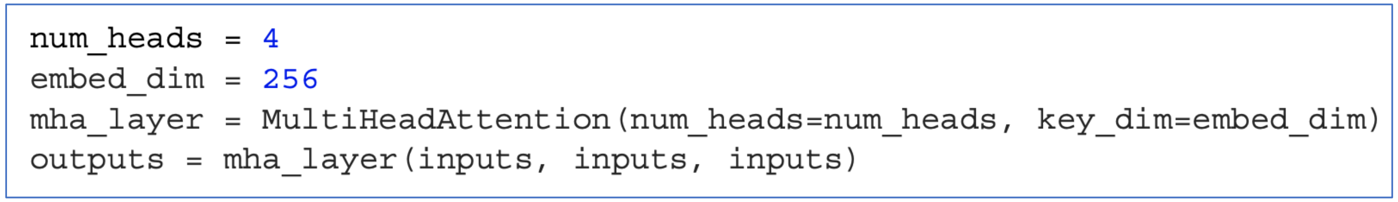
Compute the dot product (attention score) between the token and every other token.

Scale by a normalization factor, and apply a softmax.

Take the sum of all tokens weighted by the attention scores.

That sum is our output.

NumPy implementation

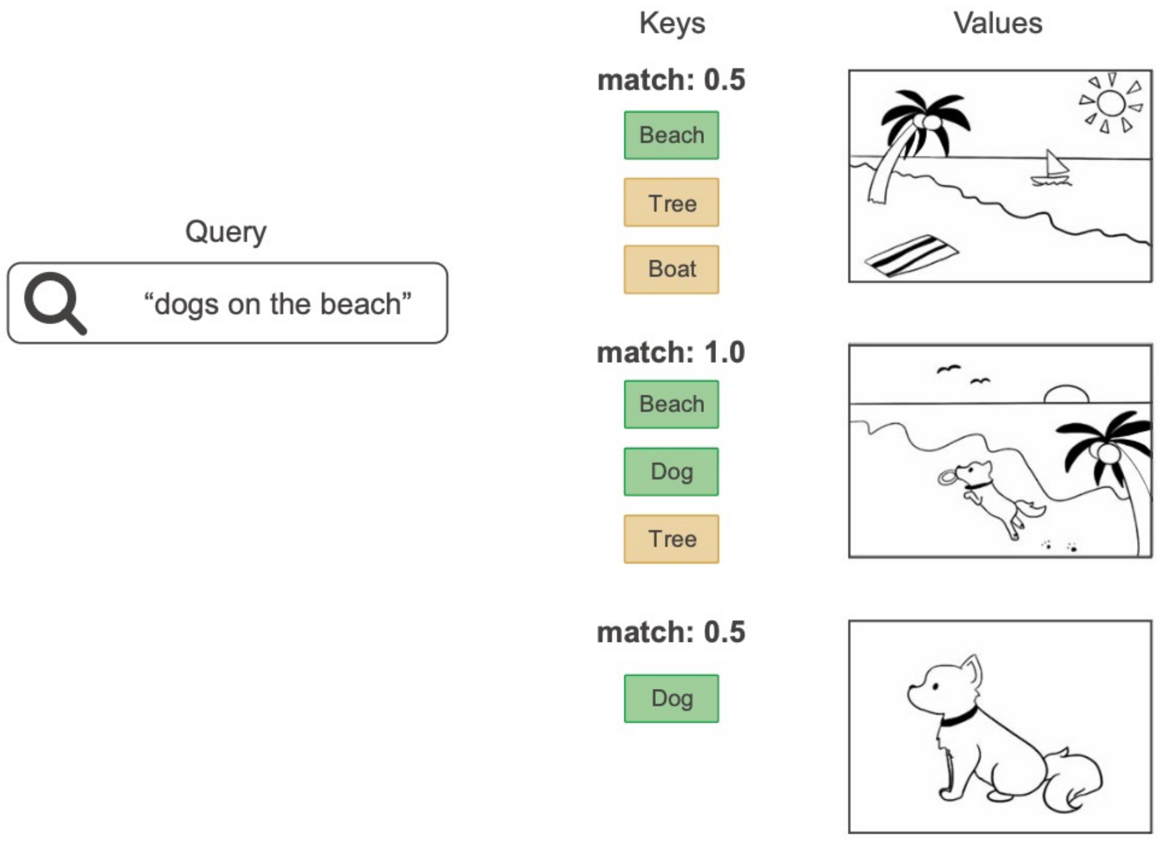


```python
num_heads = 4
embed_dim = 256
mha_layer = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
outputs = mha_layer(inputs, inputs, inputs)
```

Tensorflow/
Keras

F. Chollet

# Self attention in texts: analogy to search



Analogy to image retrieving from a database indexed by keywords (our keys).

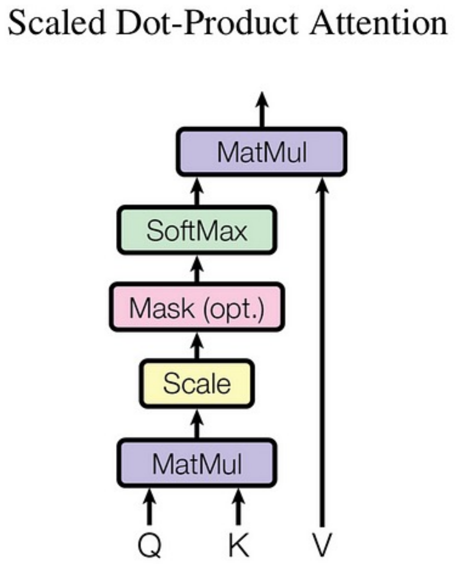
The query is compared to these keys and the matching scores are used to rank images ("values")

F. Chollet

## Self-attention in transformers

In transformers, the mechanism of self-attention works with three vectors, which are all obtained from the input multiplying by 3 **different matrices**:

- Q – query – vector representation of each word in the sequence
- K – keys – vector representations of all words in the sequence
- V – values – identical to K, but it will be multiplied by the weights of each word

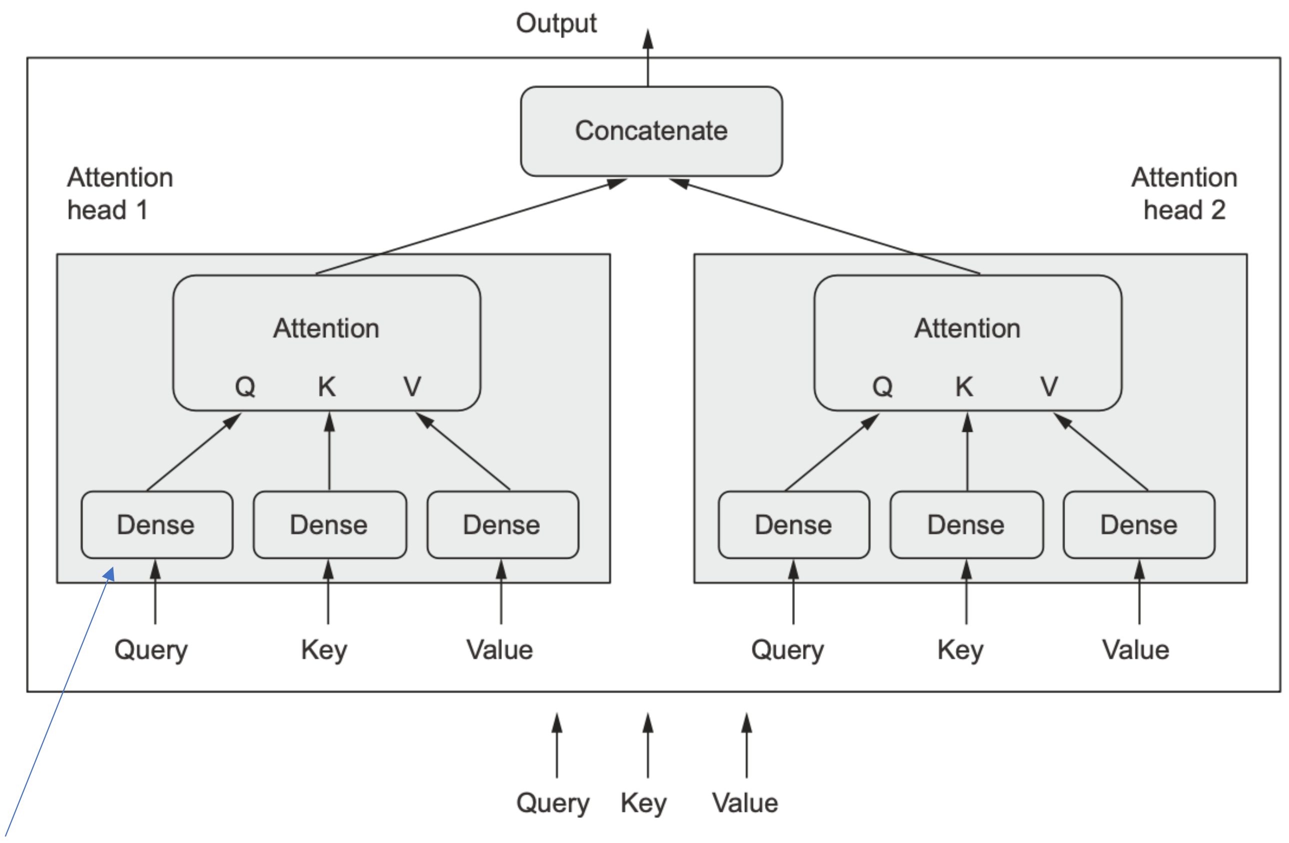


$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Multiplication of Q by $K^T$ aims to identify similarities and calculate weights (attention scores)

Multiply the result (scaled and normalized) by V applies the weights to the vector V to recover the values

# Multi-head attention



The output of the self-attention layer gets factored into a set of independent sub-spaces, learned separately

The Q, K, V are sent through three independent sets of dense weights, resulting in three separate vectors.

Each vector is processed via attention, and the three outputs are concatenated back together into a single output sequence.

Dense layers enable the attention layer to actually learn something, as opposed to being a purely stateless transformation

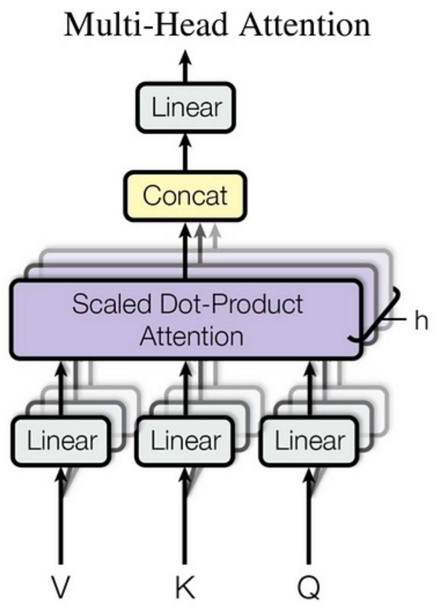
F. Chollet

# Multi-head attention

Several layers of attention are placed in parallel in a module multi-head attention; here, K,Q, V are multiplied by learned weights (linear projections)

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.

Detailed description of the processes:
https://e2eml.school/transformers.html



# Transformers

- Transformers are models that use the attention mechanism explained above (***self attention***)
- Implement Seq2Seq models (convert sequences into new sequences, using an Encoder and a Decoder), **without using recurrent networks** but only distinct feed forward layers
- Encoder – encodes text into latent representations (numerical)
- Decoder – Decodes from latent space to text

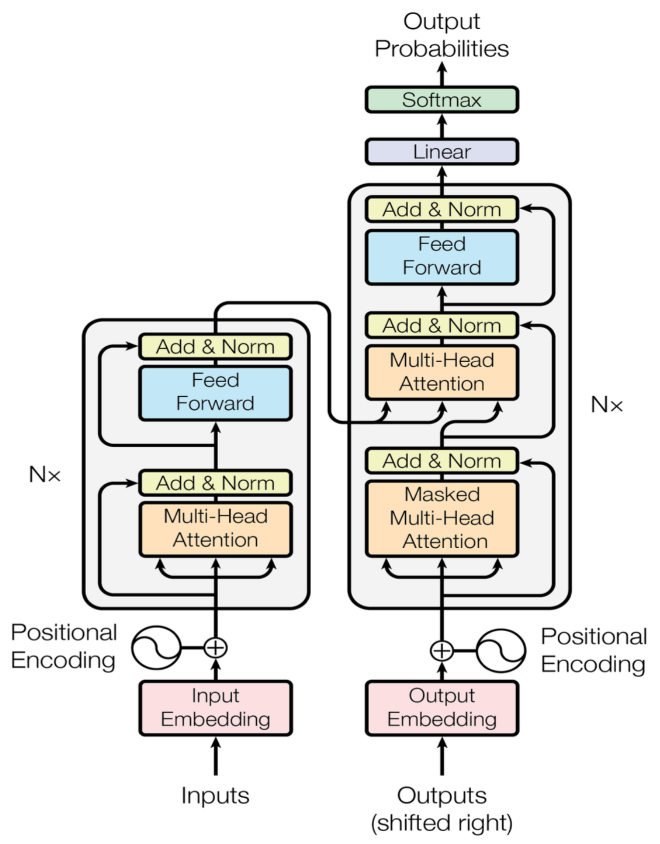


Figure 1: The Transformer - model architecture.

More details:
https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04
Papers: https://arxiv.org/abs/1706.03762

# Transformers: encoder/ decoder

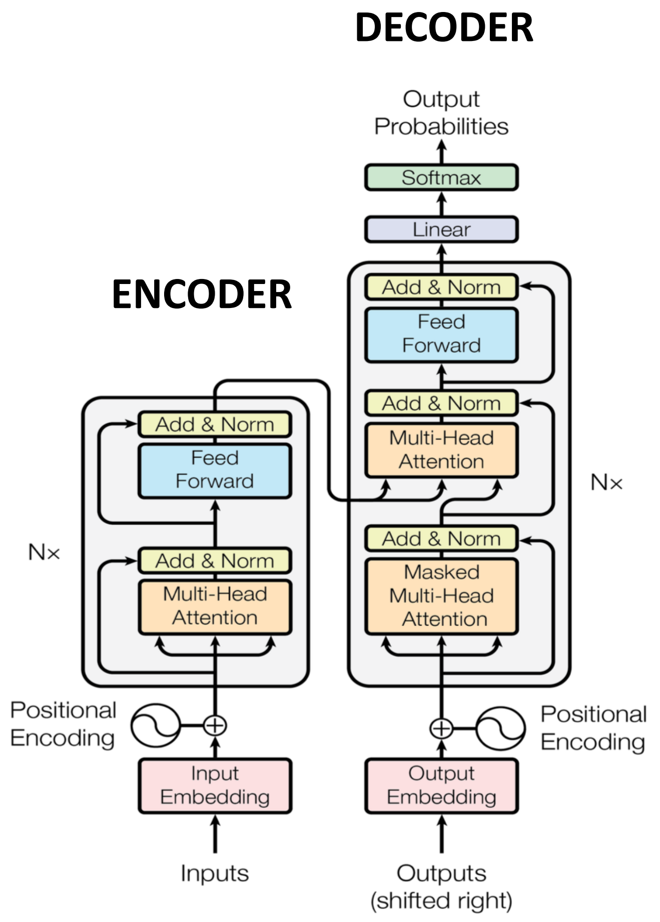
Both encoder and decoder are composed of several modules (Nx in the figure)

Modules are:

- Multi-Head Attention – the building blocks that bring innovation to the Transformers
- Feed-forward (dense) layers
- Embeddings (for inputs and outputs)

*Positional encoding* – “replaces” the position of the word (token) in the sequence; added to the word embeddings as values in interval [-1; 1] through cosine functions over the index

Some interesting patterns shared by other topologies, such as residual connections, normalization layers are also added



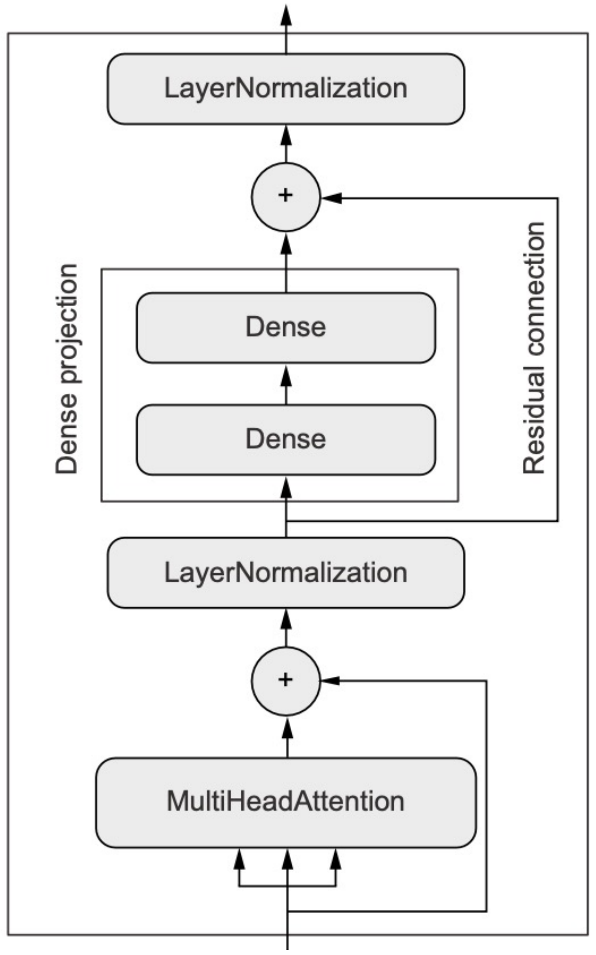
## Transformers: example of encoder

A good example of the implementation of the encoder of a transformer, and its use to build a text classifier is provided in (from F. Chollet book):

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part03_transformer.ipynb

First version uses transformer without positional encoding (does not take into account order of words)

Sugestão: correr o notebook no Google Colab
https://colab.research.google.com/



Transformer encoder implementation from F. Cholett

## Transformers for text classification

- Transformers (encoders) may be used to support text classification
- Models typically start with embeddings, followed by a transformer encoder, and by one (or more) Dense layers

Example with the IMDB dataset

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part03_transformer.ipynb

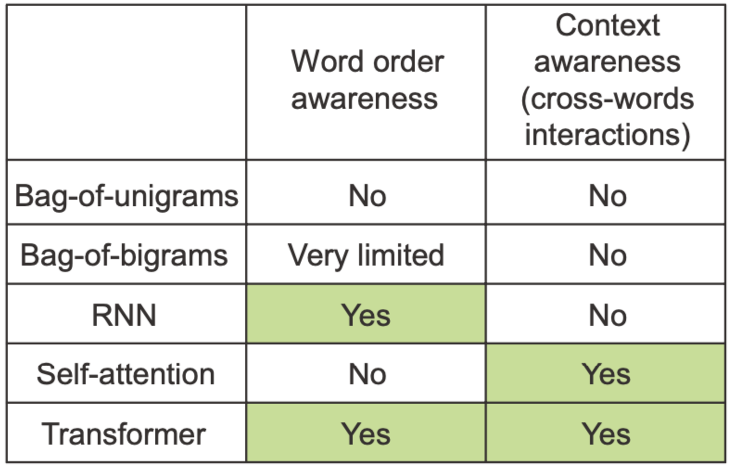
```
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

**Since TransformerEncoder returns full sequences, we need to reduce each sequence to a single vector for classification via a global pooling layer.**

# Positional encoding

Previous models did not take into account the position of the tokens in the sequence (neither the Dense layer nor the self attention which is a set-processing mechanism)



| | Word order awareness | Context awareness (cross-words interactions) |
|---|---|---|
| Bag-of-unigrams | No | No |
| Bag-of-bigrams | Very limited | No |
| RNN | Yes | No |
| Self-attention | No | Yes |
| Transformer | Yes | Yes |

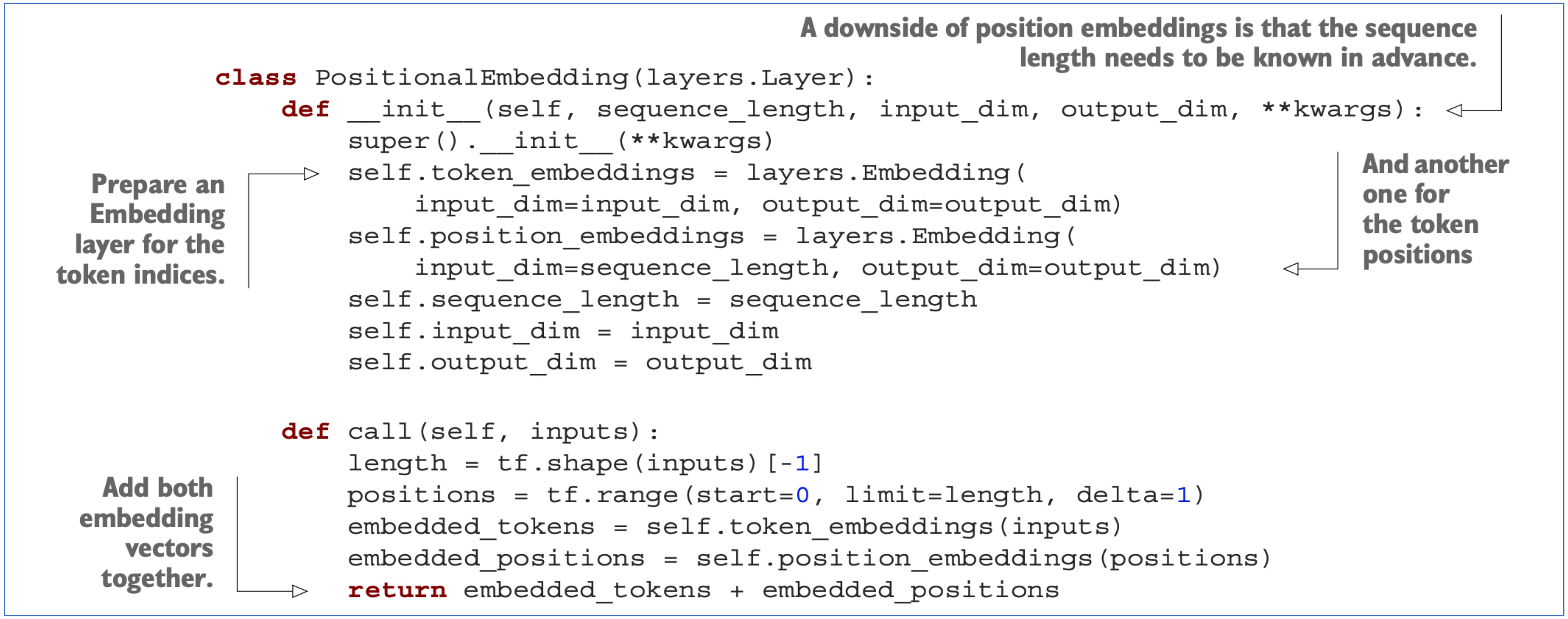
A mechanism needs to be used to consider the information about position of the tokens – **positional encoding**

Idea: to add the token's position to each word embedding
So, input word embeddings consider the word vector and the positional encoding

The original paper added this in the form of a value in the range [-1,1] that varied cyclically depending on the position (using the cosine function)

## Positional embedding

An alternative to the previous scheme is to consider positional embeddings, i.e. learn the values in the same way we learn word indexes



A downside of position embeddings is that the sequence length needs to be known in advance.

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

Prepare an Embedding layer for the token indices.

And another one for the token positions

Add both embedding vectors together.

## Transformers for text classification – v2

Now with Positional Embeddings

```
vocab_size = 20000
sequence_length = 600
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("full_transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20,
     callbacks=callbacks)
model = keras.models.load_model(
    "full_transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder,
                    "PositionalEmbedding": PositionalEmbedding})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

**Look here!**

Example with the IMDB dataset

[https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part03_transformer.ipynb](https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part03_transformer.ipynb)

## Transformers - BERT family

- **BERT** (Bidirectional Encoder Representations from Transformers) was created at Google (2018) for word processing and quickly became popular, giving rise to several variants, improving the original and adapting to various types of tasks:
- BERT only uses the Encoder part of the transformers, encoding the entire text sequence at once
- It is trained using a token-masking technique in the training data at random positions

Paper: https://arxiv.org/pdf/1810.04805.pdf
https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270
https://www.youtube.com/watch?v=h_U27jBNYI4

## Transformers - BERT family – learning task

**Masked Language Modeling** (Masked LM)

- The objective of this task is to guess the masked tokens.
- Example:

  That's [mask] she [mask] -> That's what she said

**Next Sentence Prediction** (NSP)

- Given a pair of two sentences, the task is to say whether or not the second follows the first (binary classification).
- Example:

  *Input* = [CLS] That's [mask] she [mask]. [SEP] Hahaha, nice! [SEP]

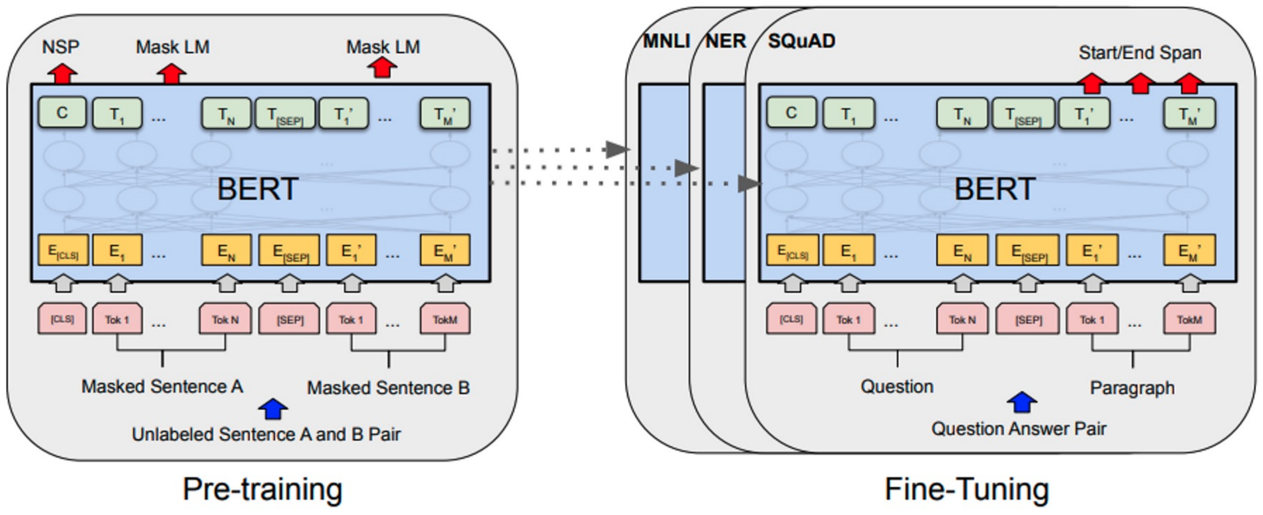
  *Label* = *IsNext*

  *Input* = [CLS] That's [mask] she [mask]. [SEP] Dwight, you ignorant [mask]! [SEP]

  *Label* = *NotNext*

# Transformers - BERT family

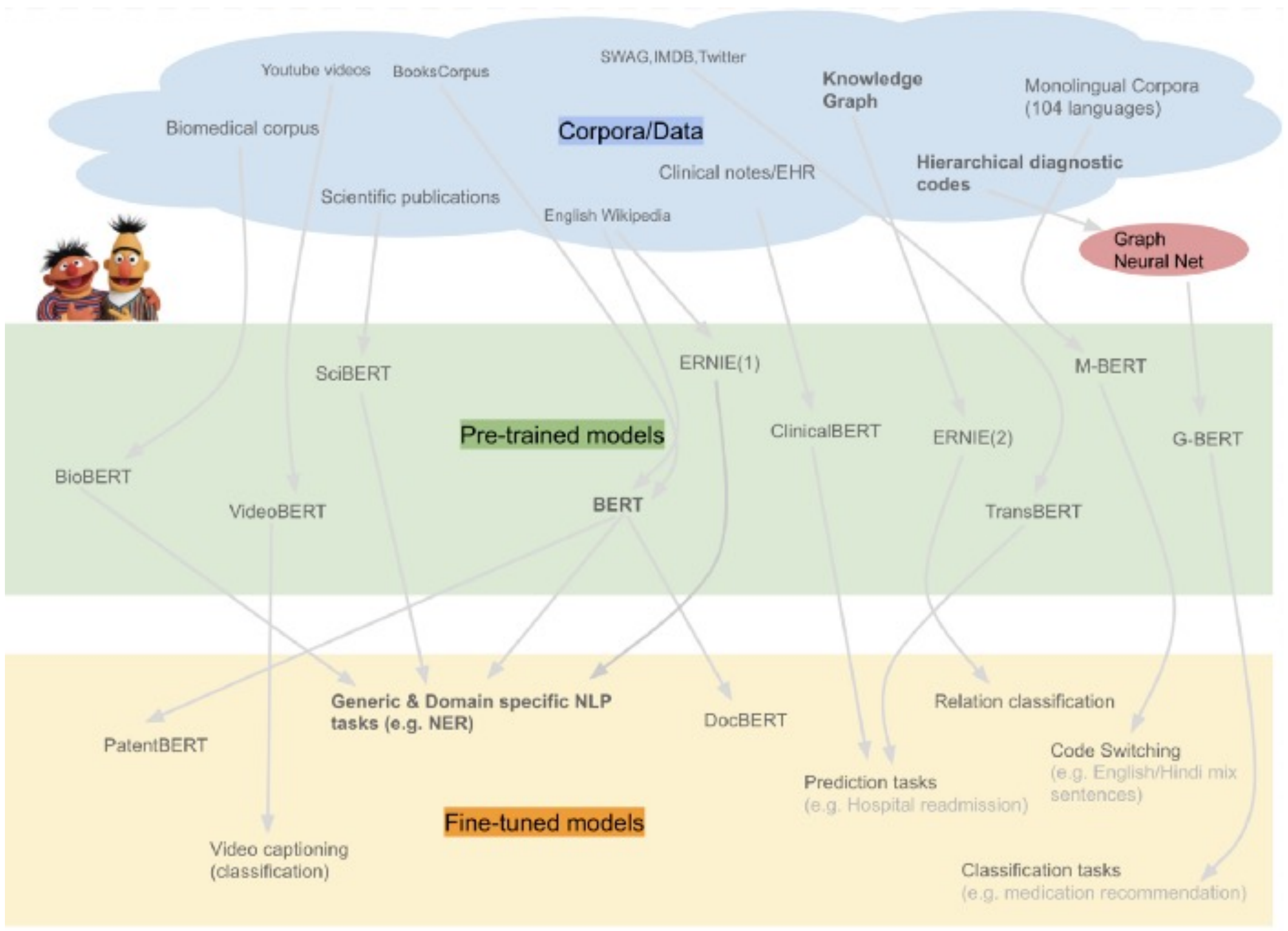
- **BERT** and several of its successors have been used for numerous text mining applications, using BERT (and variants) to encode the text and then have a model that uses the BERT output to predict other variables



- Sentiment analysis
- NER
- Question answering
- Document classification
- …

https://kgptalkie.com/sentiment-classification-using-bert/

## BERT family


Youtube videos
BooksCorpus
SWAG,IMDB,Twitter
Knowledge Graph
Monolingual Corpora (104 languages)
Biomedical corpus
Corpora/Data
Clinical notes/EHR
Hierarchical diagnostic codes
Scientific publications
English Wikipedia
Graph Neural Net
SciBERT
ERNIE(1)
M-BERT
Pre-trained models
ClinicalBERT
ERNIE(2)
G-BERT
BioBERT
VideoBERT
BERT
TransBERT
Generic & Domain specific NLP tasks (e.g. NER)
DocBERT
Relation classification
PatentBERT
Code Switching (e.g. English/Hindi mix sentences)
Prediction tasks (e.g. Hospital readmission)
Fine-tuned models
Video captioning (classification)
Classification tasks (e.g. medication recommendation)

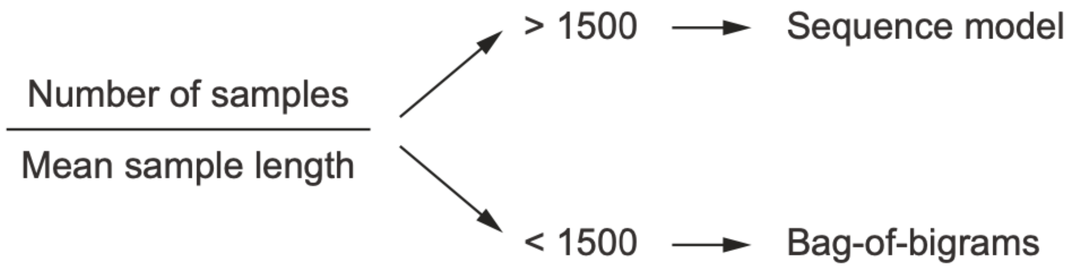
## BERT and variants for sentiment analysis (IMDB)

Several examples showing how BERT and variants can be used for the IMDB dataset

- https://www.tensorflow.org/text/tutorials/classify_text_with_bert
- https://www.youtube.com/watch?v=sZdIybqppqQ
  https://kgptalkie.com/sentiment-classification-using-bert/
- https://www.kaggle.com/code/atulanandjha/bert-testing-on-imdb-dataset-extensive-tutorial (pytorch)
- https://github.com/jlealtru/website_tutorials/blob/main/notebooks/RoBERTA%20with%20IMDB.ipynb

## Text classification – which is the best model?

- Although transformers have greatly improved the results on many NLP tasks, more traditional models (e.g. n-grams) may still be the best option in some cases, specially with less data
- Here two factors are relevant: the number of examples (samples) and the mean length (number of words/ tokens)



Rule of thumb from a study conducted by F. Chollet and collaborators

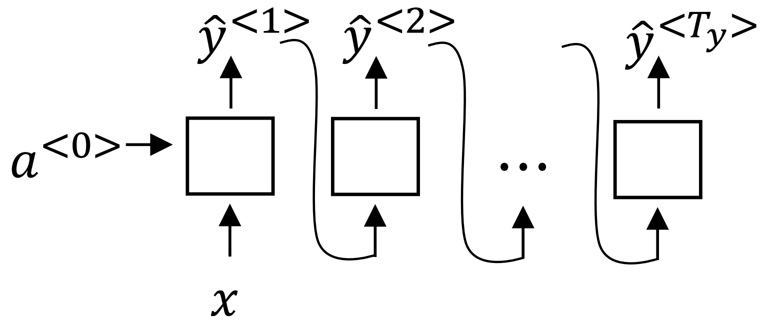
While many other studies have and may be performed, the performance of deep learning models/ transformers is typically more effective when more data are available

# What is a Language Model

- **Language model (LM)** - prediction engine that takes a sequence of words and predicts the most likely sequence to come after that sequence. It does this by assigning a probability to likely next sequences and then samples from those to choose one. The process repeats until some stopping criteria is met.
- LMs learn these probabilities by training on large corpora of text.
  - models will be better in some tasks than others.
  - LMs may generate statements that seem plausible, but are actually just random without being grounded in reality.

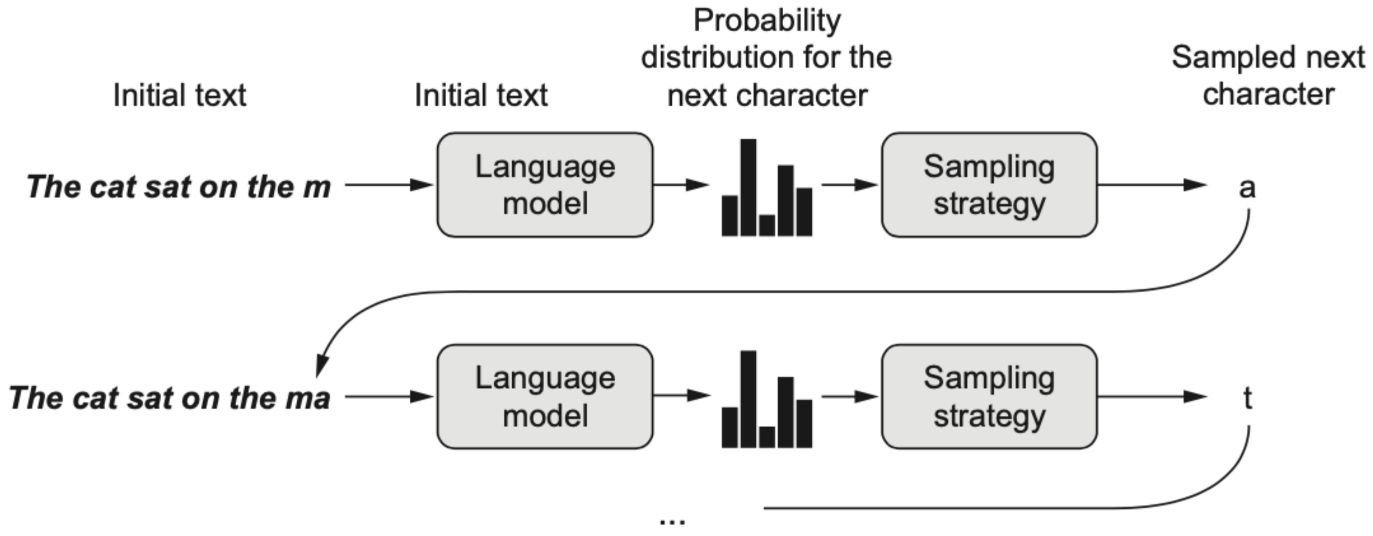
## Generating sequences with recurrent NNs

- Recurrent NNs (e.g. RNNs, LSTMs) can be used to generate different types of sequences, through language models, learned from natural language texts
- The idea is to learn probabilities associated to the different symbols of the alphabet (letters, words) from training samples
- When generating, an input sequence is given and the model predicts the next symbol



## Generating sequences with LSTMs- example

- Example of text generating letter by letter using LSTMs
- Input – strings of N charaters
- Output – the character N+1 (using a *softmax* layer)



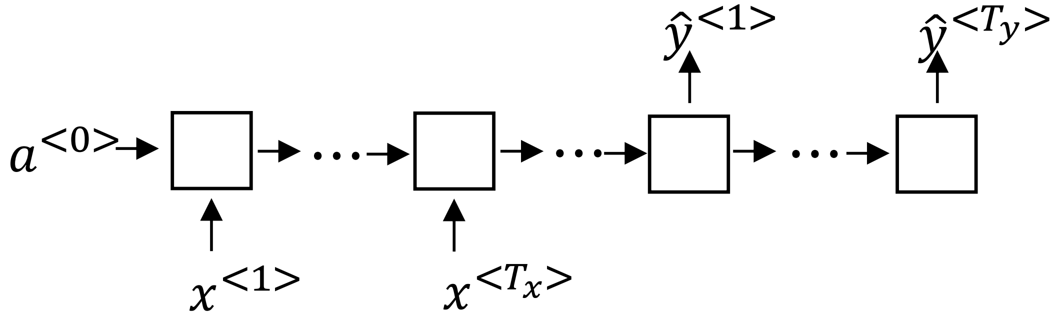
# Sequence to sequence models

A **sequence-to-sequence** model takes a sequence as input and translates it into a different sequence

Many different applications:

- Translation
- Text summarization
- Question answering
- Chatbots

Seq to seq models can be implemented by recurrent models (RNNs / LSTMs, etc)



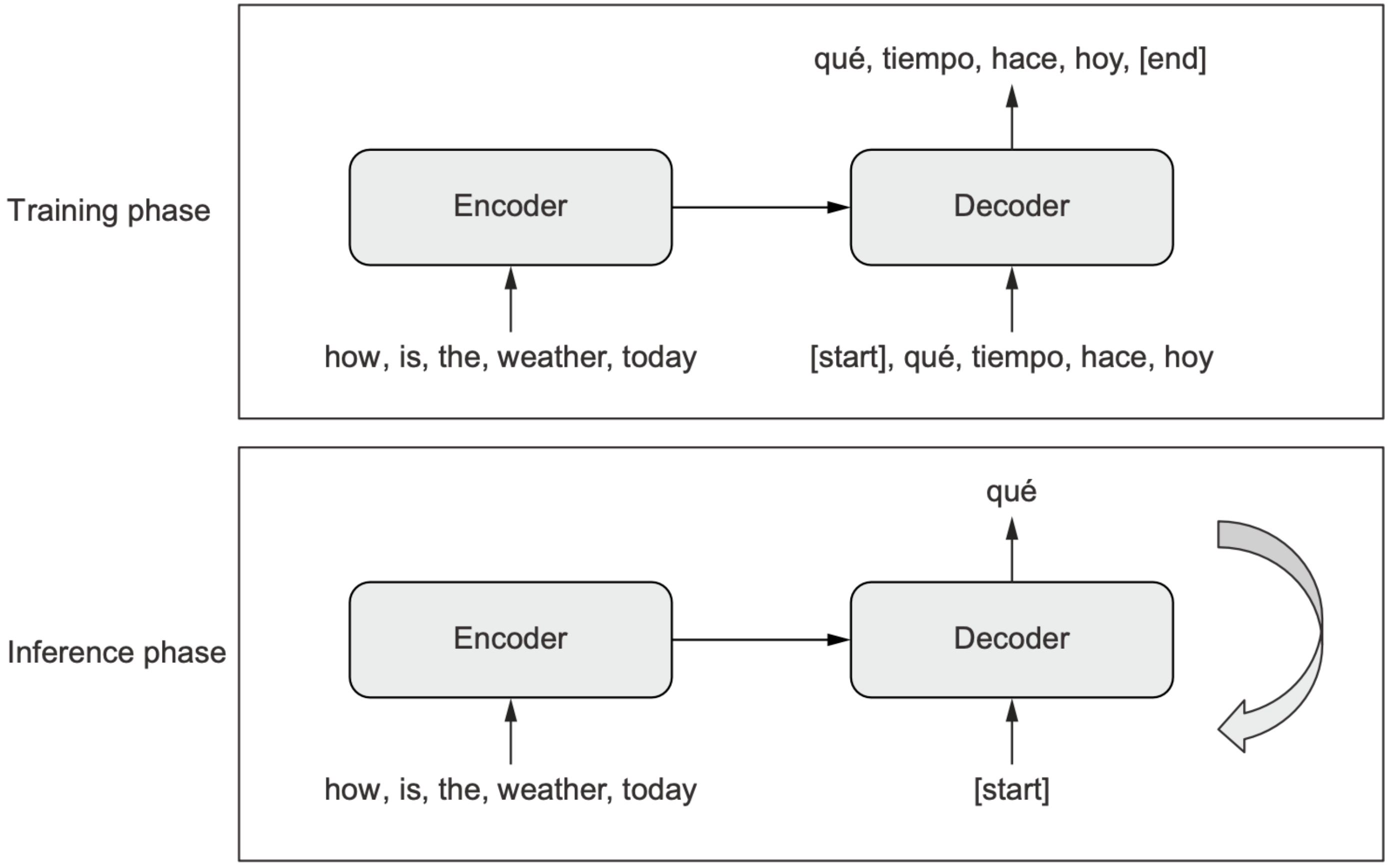
And more recently by transformers

# Sequence to sequence models: template

## Encoder – decoder architecture

**Encoder** – converts the input sequence into an intermediate representation (latent space)

**Decoder** – trained to predict the next token (language model) in the target sequence looking at previous items of the target and the input sequence


Training phase
qué, tiempo, hace, hoy, [end]
Encoder
Decoder
how, is, the, weather, today
[start], qué, tiempo, hace, hoy
Inference phase
qué
Encoder
Decoder
how, is, the, weather, today
[start]

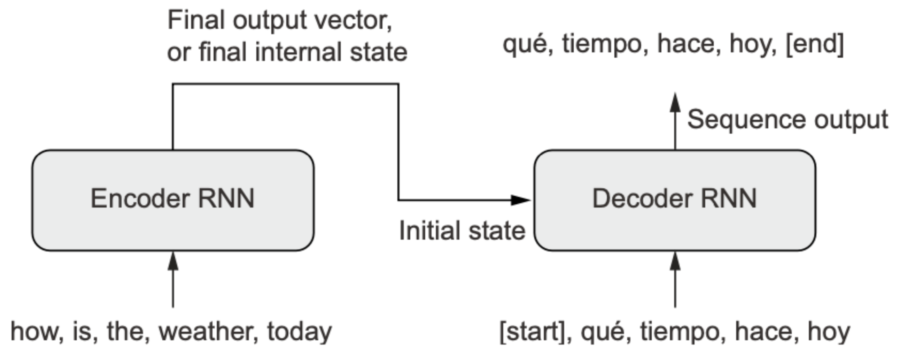
## Example of language models for translation

- Example of LMs for a simple English to Spanish translation problem
- The code is available in:

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part04_sequence-to-sequence-learning.ipynb

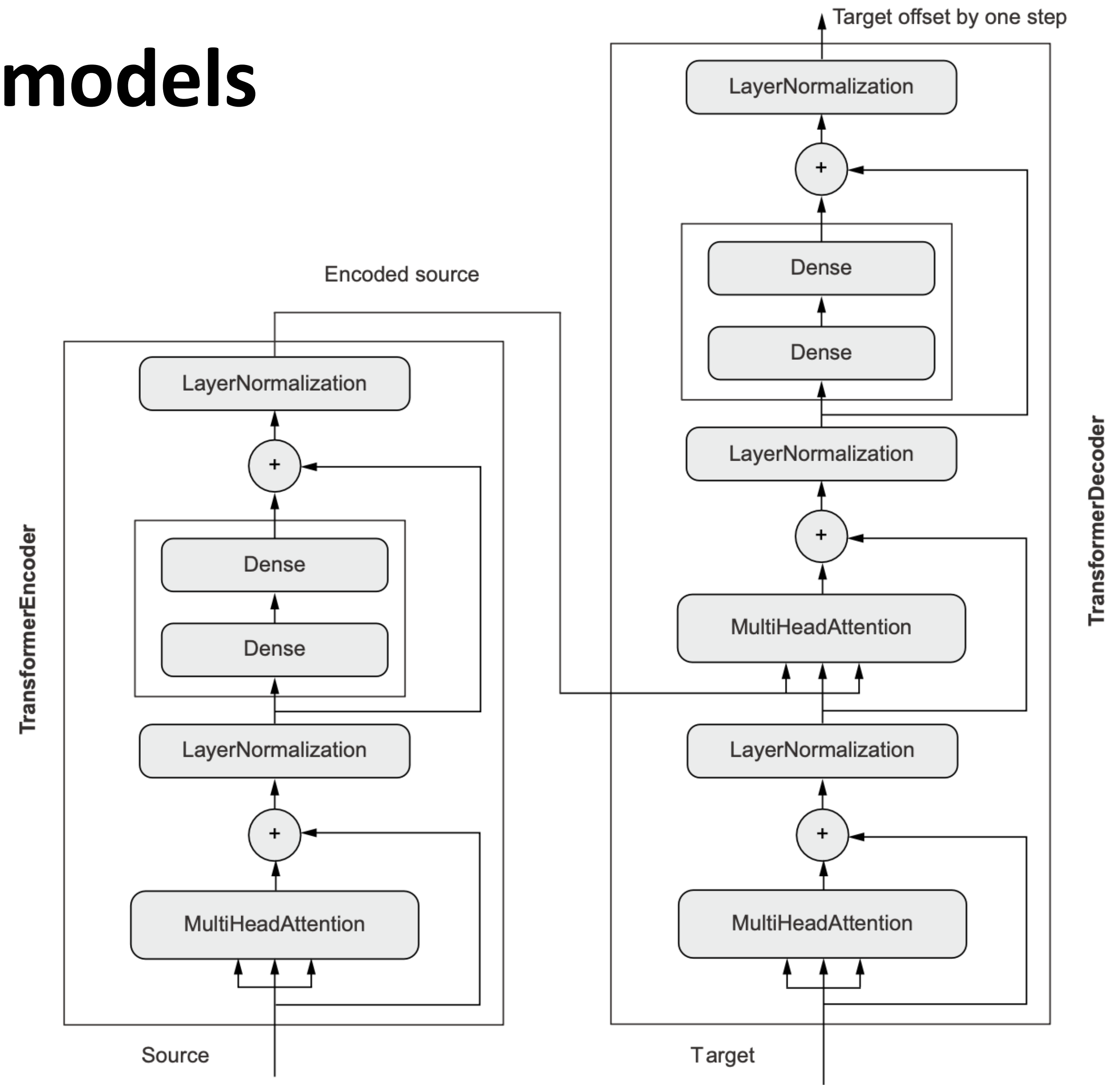
- First approach is based on the use of recurrent NNs (GRUs in the case) as encoder and decoder

Dataset available at www.manythings.org/anki/



## Example of language models for translation

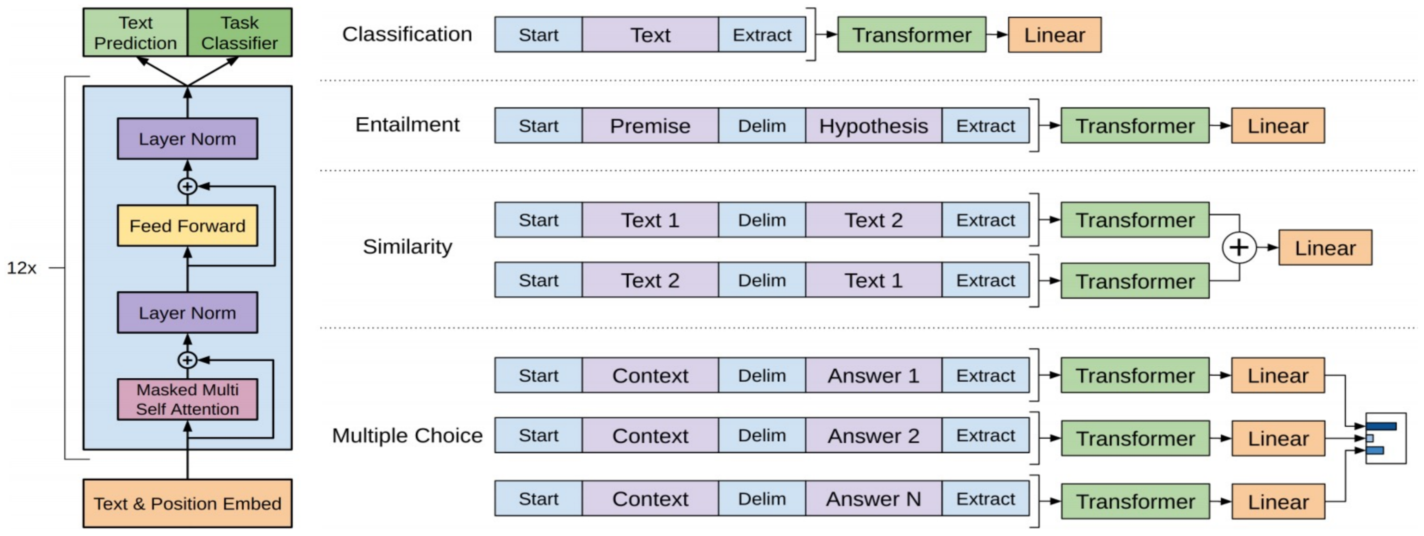
- The second approach uses a Encoder-Decoder Transformer architecture



# GPT

- **GPT** (Generative Pretrained Transformer) is an LLM based on the architecture of transformers developed by OpenAI since 2018, being one of BERT's competitors
- It already has several versions of which the last are GPT3.5 and GPT4 (2023), with several hundred billion parameters
- Only uses the **decoders** part of the transformers – input sequences are fed directly
- It can be used in various word processing and classification applications

Its greatest capacity is the use of the decoder as a generative model, generating text from prompts !



Text Prediction
Task Classifier
Layer Norm
Feed Forward
Layer Norm
Masked Multi Self Attention
12x
Text & Position Embed
Classification
Start
Text
Extract
Transformer
Linear
Entailment
Start
Premise
Delim
Hypothesis
Extract
Transformer
Linear
Similarity
Start
Text 1
Delim
Text 2
Extract
Transformer
Start
Text 2
Delim
Text 1
Extract
Transformer
Linear
Multiple Choice
Start
Context
Delim
Answer 1
Extract
Transformer
Linear
Start
Context
Delim
Answer 2
Extract
Transformer
Linear
Start
Context
Delim
Answer N
Extract
Transformer
Linear