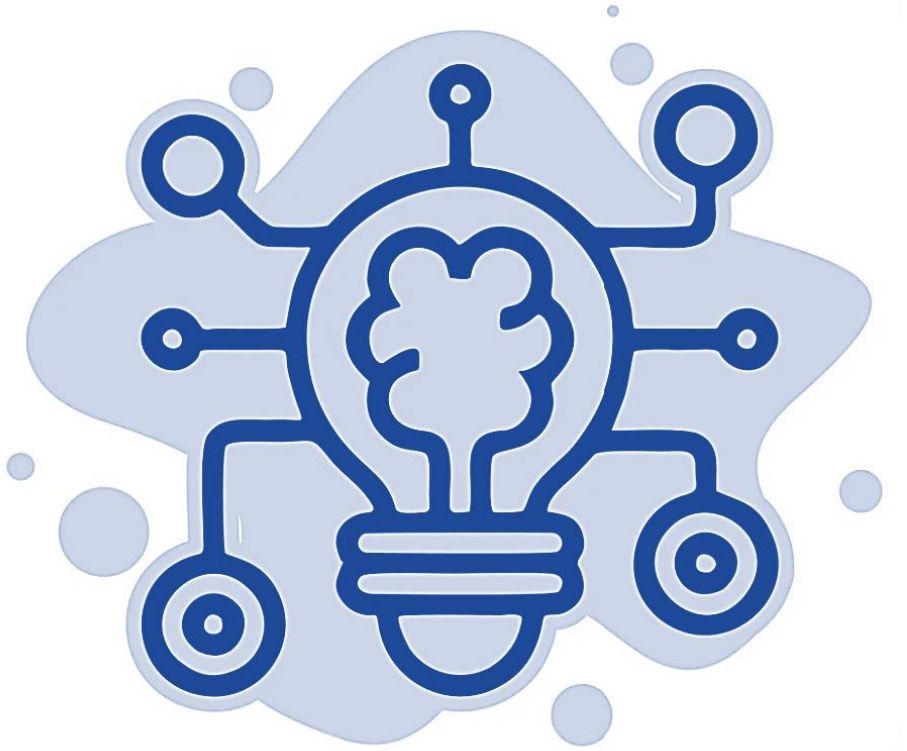


# Transformers

***Hichem Felouat***
***hichemfel@gmail.com***

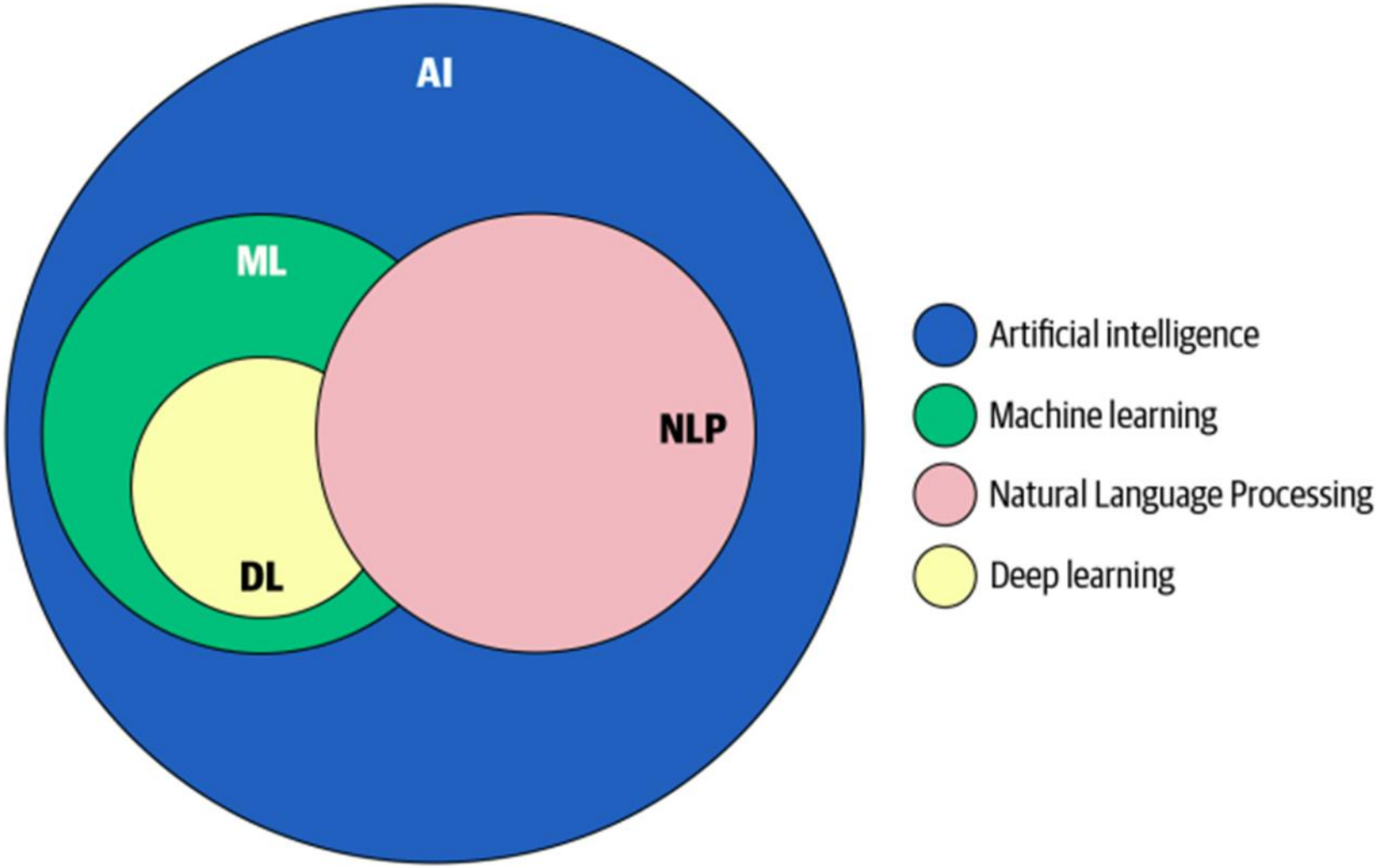
# Contents

# Natural Language Processing NLP

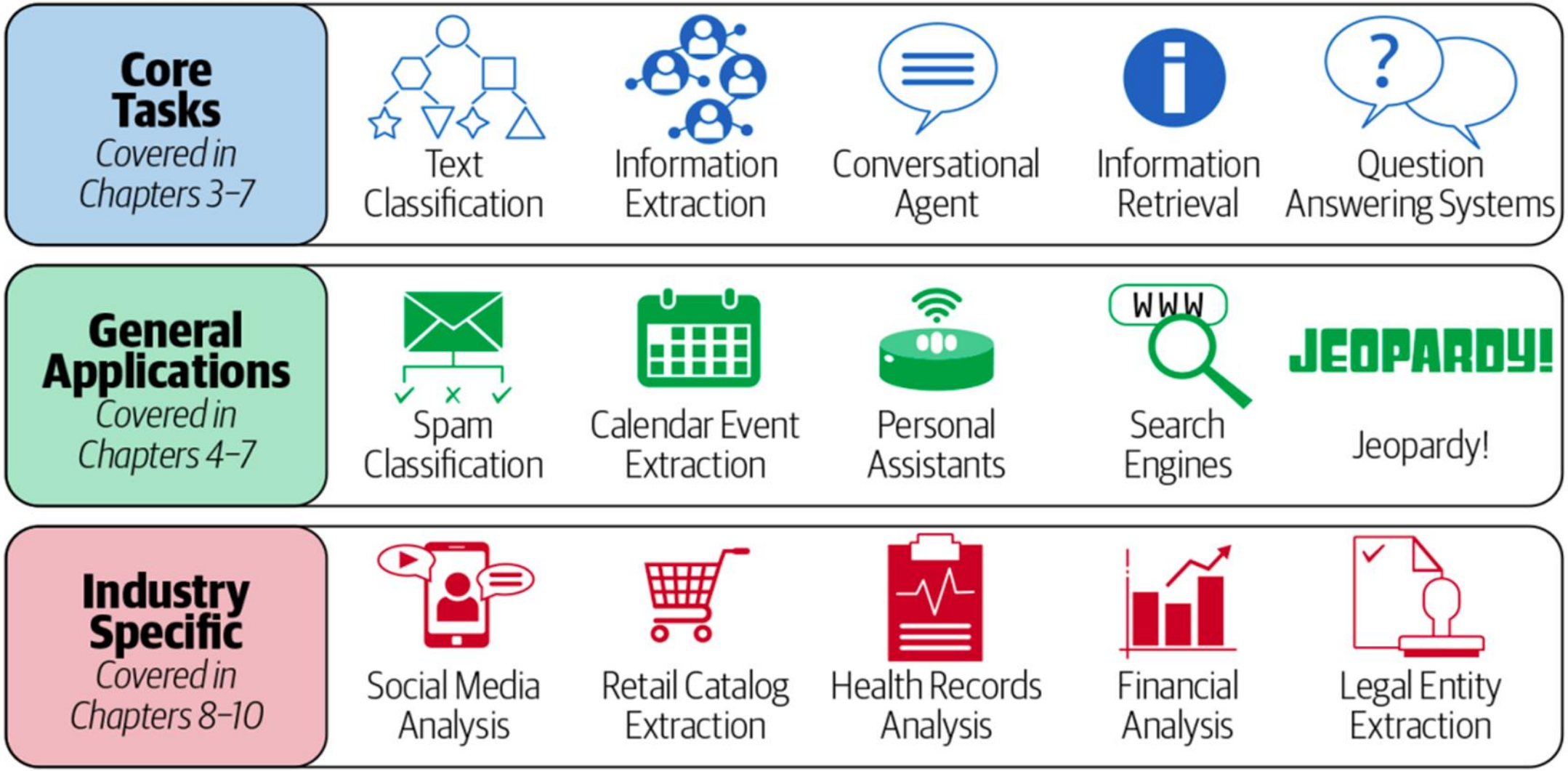
- **Natural language processing (NLP)** is a subfield of **artificial intelligence** concerned with the interactions between computers and human (natural) languages, in particular how to program computers to process and analyze large amounts of natural language data.

- **Challenges in natural language processing** frequently involve speech recognition, natural language understanding, and natural language generation.

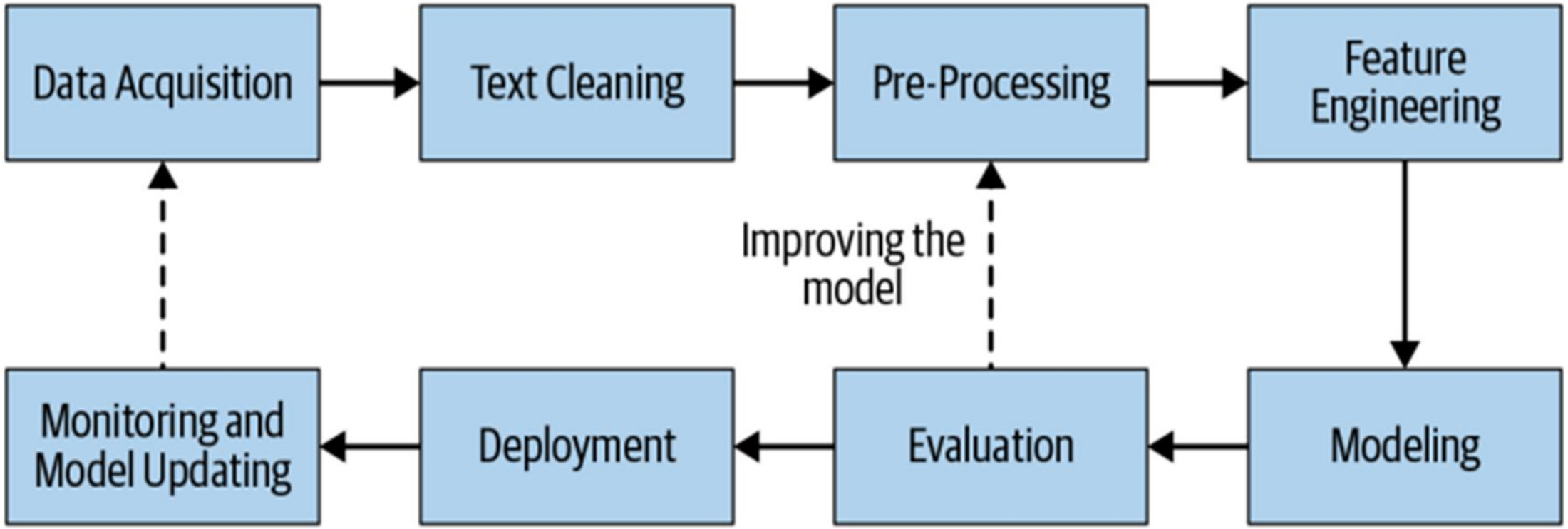
# Natural Language Processing NLP



# Natural Language Processing NLP



# Generic NLP Pipeline



# Texts to Sequence/Matrix

- In natural language processing (NLP), texts can be represented as a **sequence** or a **matrix**, depending on the task and the model type.

texts = ["I love Algeria", "machine learning", "Artificial intelligence", "AI"]

- The total number of documents: 4
- The number of distinct words (**Tokenization** ): 8
- word_index :

{'i': 1, 'love': 2, 'algeria': 3, 'machine': 4, 'learning': 5, 'artificial': 6, 'intelligence': 7, 'ai': 8}

- texts_to_sequences : input [Algeria love AI]

[3, 2, 8]

- sequences_to_texts : input [3, 4, 7, 2, 8, 1, 3]

['algeria machine intelligence love ai i algeria']

# Texts to Sequence/Matrix

- **binary**: Whether or not each word is present in the document. This is the default.
- **count** : The count of each word in the document.
- **freq** : The frequency of each word as a ratio of words within each document.
- **tfidf** : The Text Frequency-Inverse Document Frequency (TF-IDF) scoring for each word in the document.

```
texts = [
   "blue car and blue window", "black crow in the window","i see my reflection in the window"
]
```

| | and | black | blue | car | crow | in | my | reflection | see | the | window |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

# Texts to Sequence/Matrix

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

**TF-IDF**

Term $x$ within document $y$

$tf_{x,y}$ = frequency of $x$ in $y$

$df_x$ = number of documents containing $x$

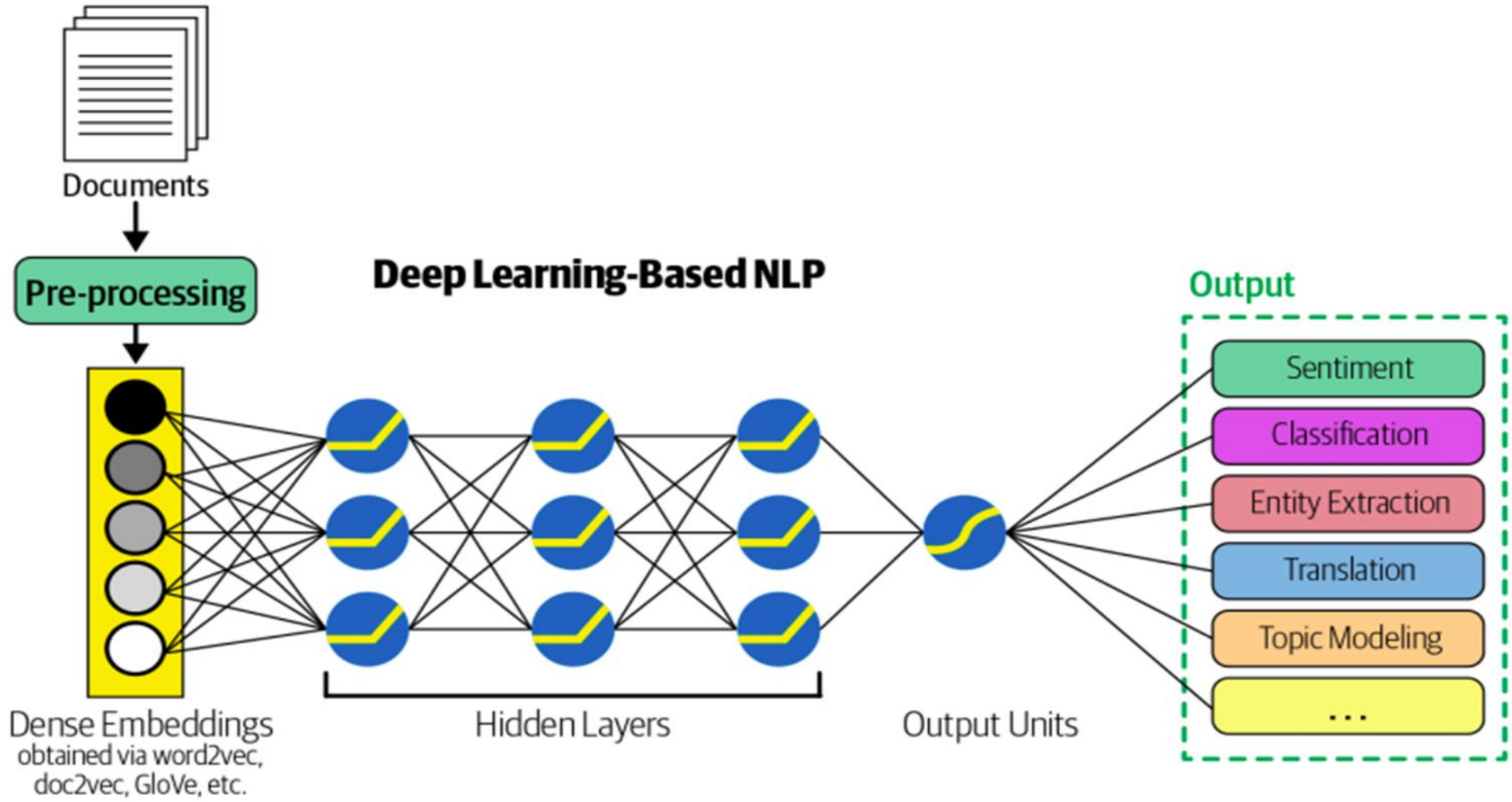
$N$ = total number of documents

# Sequence Padding

- Sequence padding is the process of **adding zeroes** or other filler tokens to sequences of variable length so that **all sequences have the same length**.
- Many machine learning models require **fixed-length inputs**, and variable-length sequences can't be fed directly into these models.

```
sequences = [ [1, 2, 3, 4], [1, 2, 3], [1] ]
maxlen= 4
result:  [[1 2 3 4]
          [0 1 2 3]
          [0 0 0 1]]
```

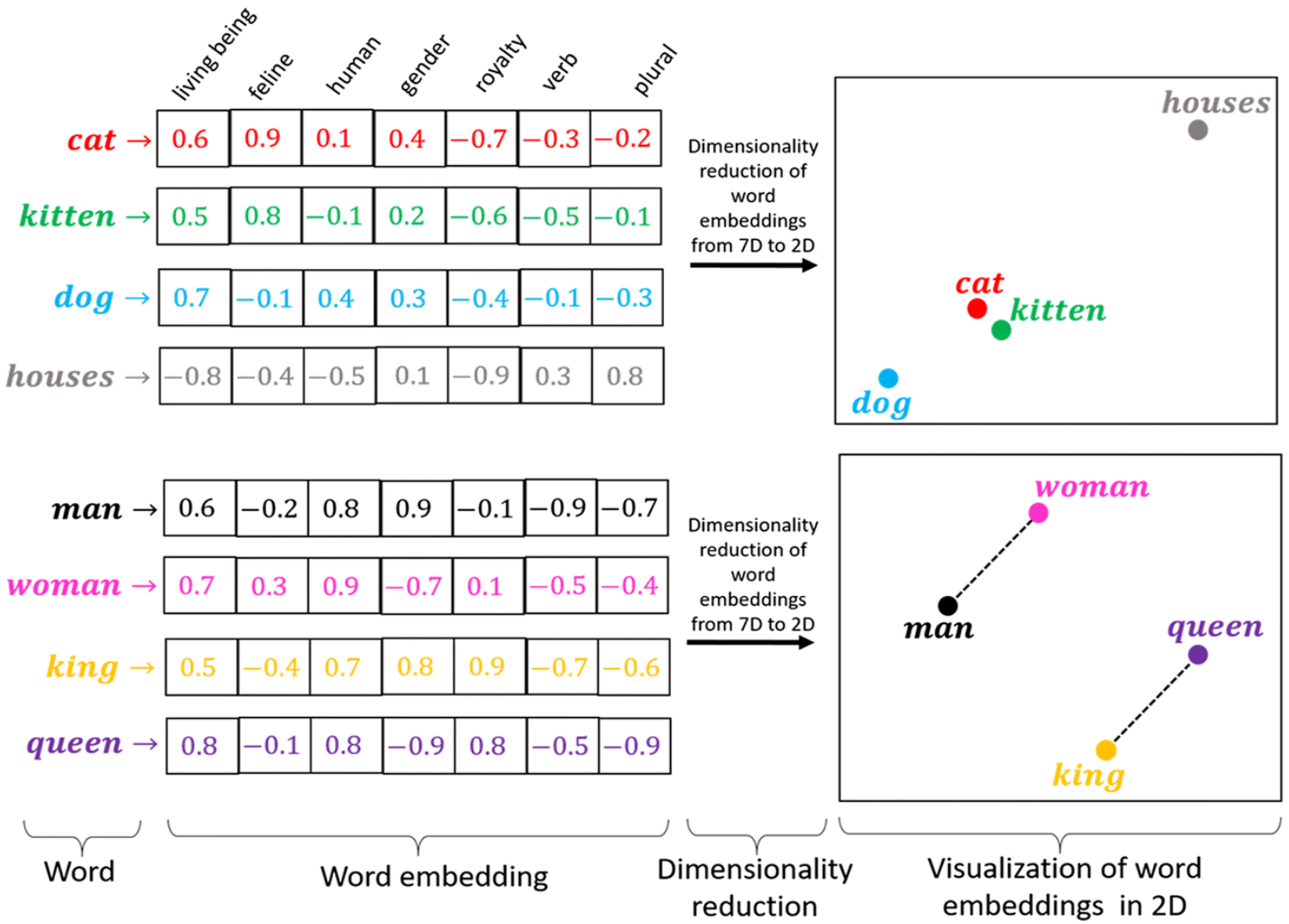
# Deep Learning-Based NLP



# Word Embedding

- Word embedding is a technique used in NLP to represent words as **numerical vectors** in a **high-dimensional space**.
- Word embedding aims to **capture** the **meaning** and **context** of words in a way that is useful for downstream NLP tasks, such as text classification, sentiment analysis, and machine translation.
- There are several popular algorithms for creating word embeddings, such as **Word2Vec**, **GloVe**, and **fastText**.

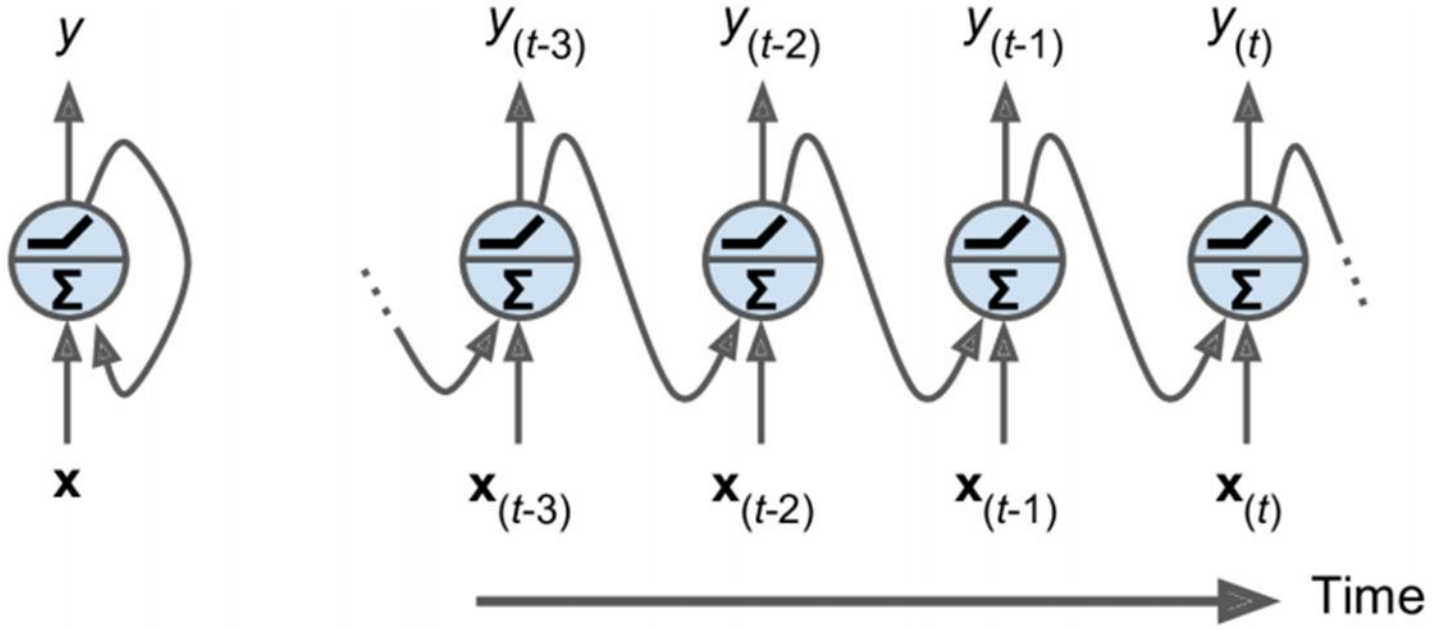
# Word Embedding



# Recurrent Neural Network(RNN)

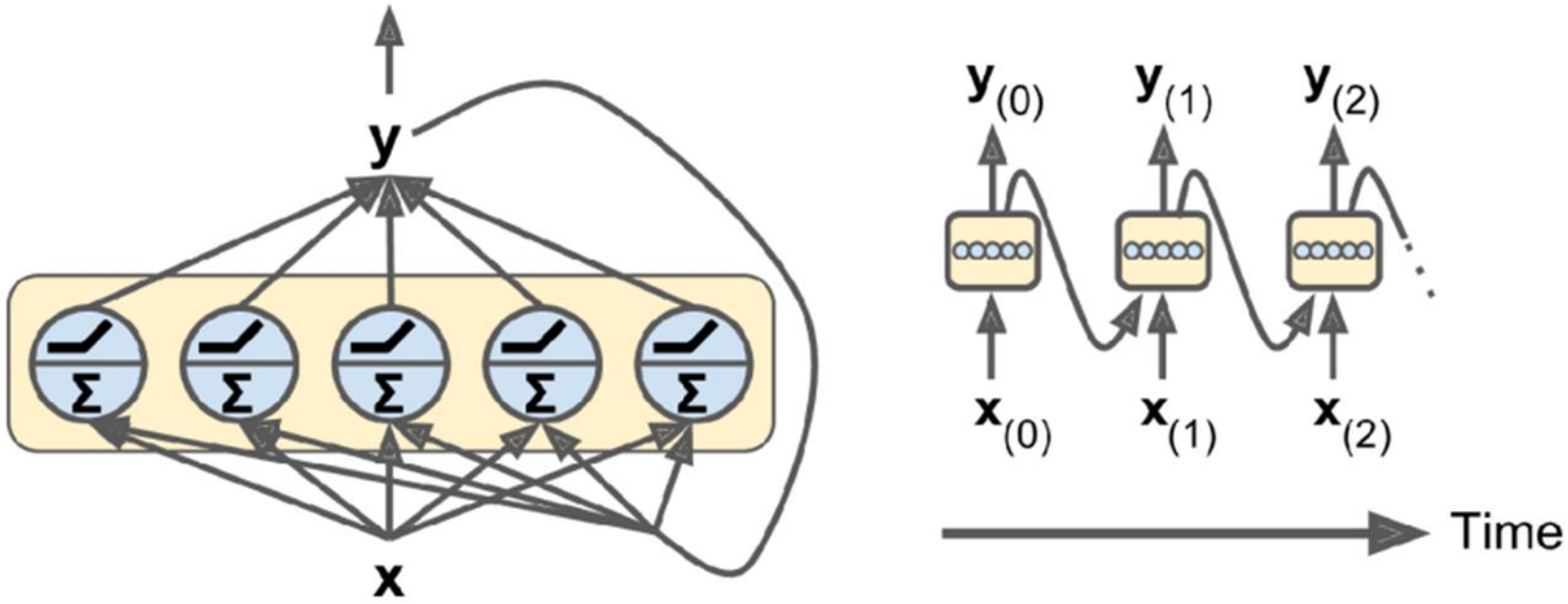
- The **simplest** possible RNN composed of **one neuron** receiving inputs, producing an output, and **sending that output back to itself** (figure -left).

- We can represent this tiny network against the time axis, as shown in (figure -right). This is called **unrolling the network through time**.



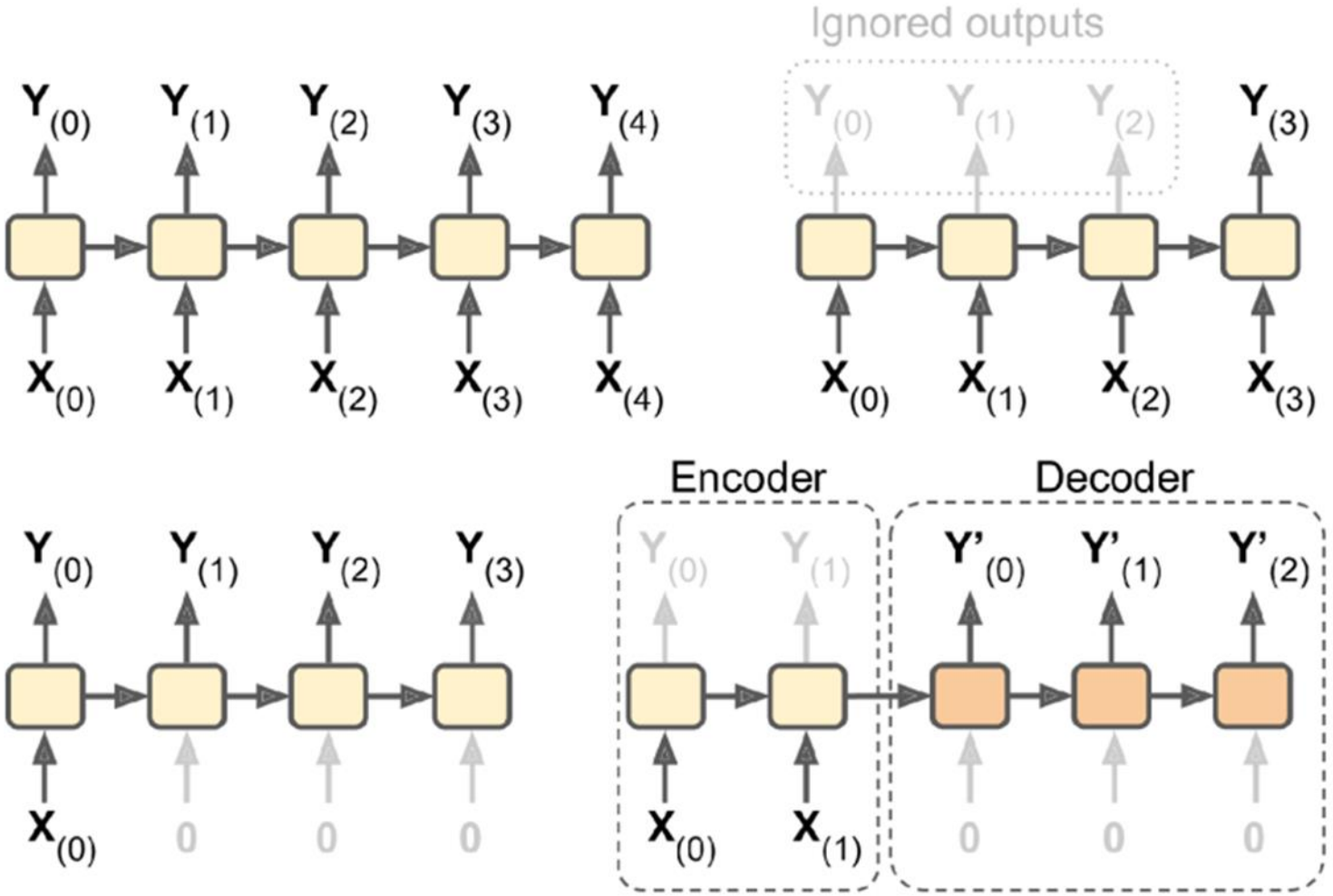
# Recurrent Neural Network(RNN)

- You can easily create a **layer** of recurrent neurons. At each time step **t**, every neuron receives both the input vector **x(t)** and the **output vector** from the previous time step **y(t–1)**.



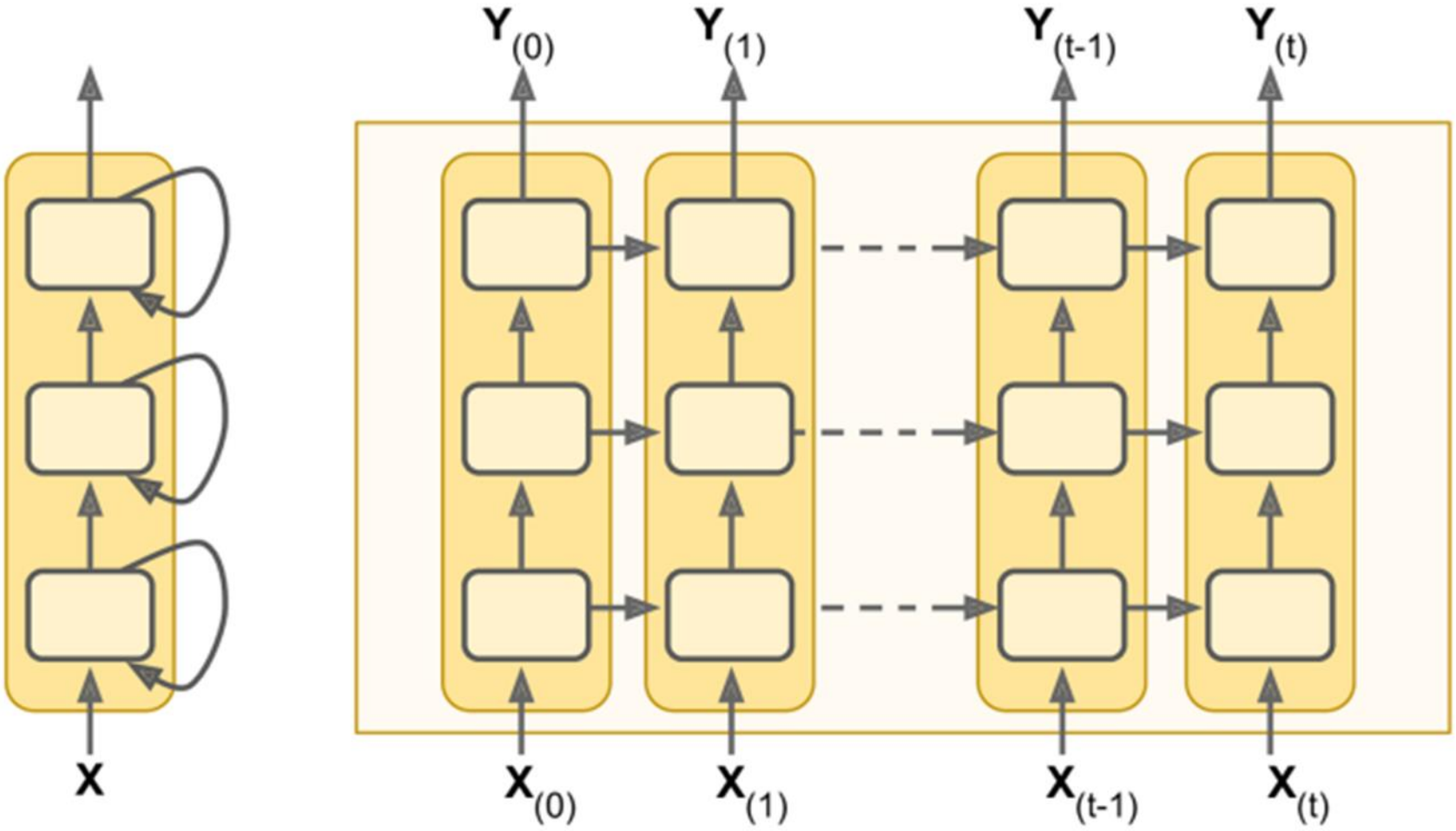

A recurrent neuron (left) unrolled through time (right)

# Recurrent Neural Network(RNN)




- Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks.

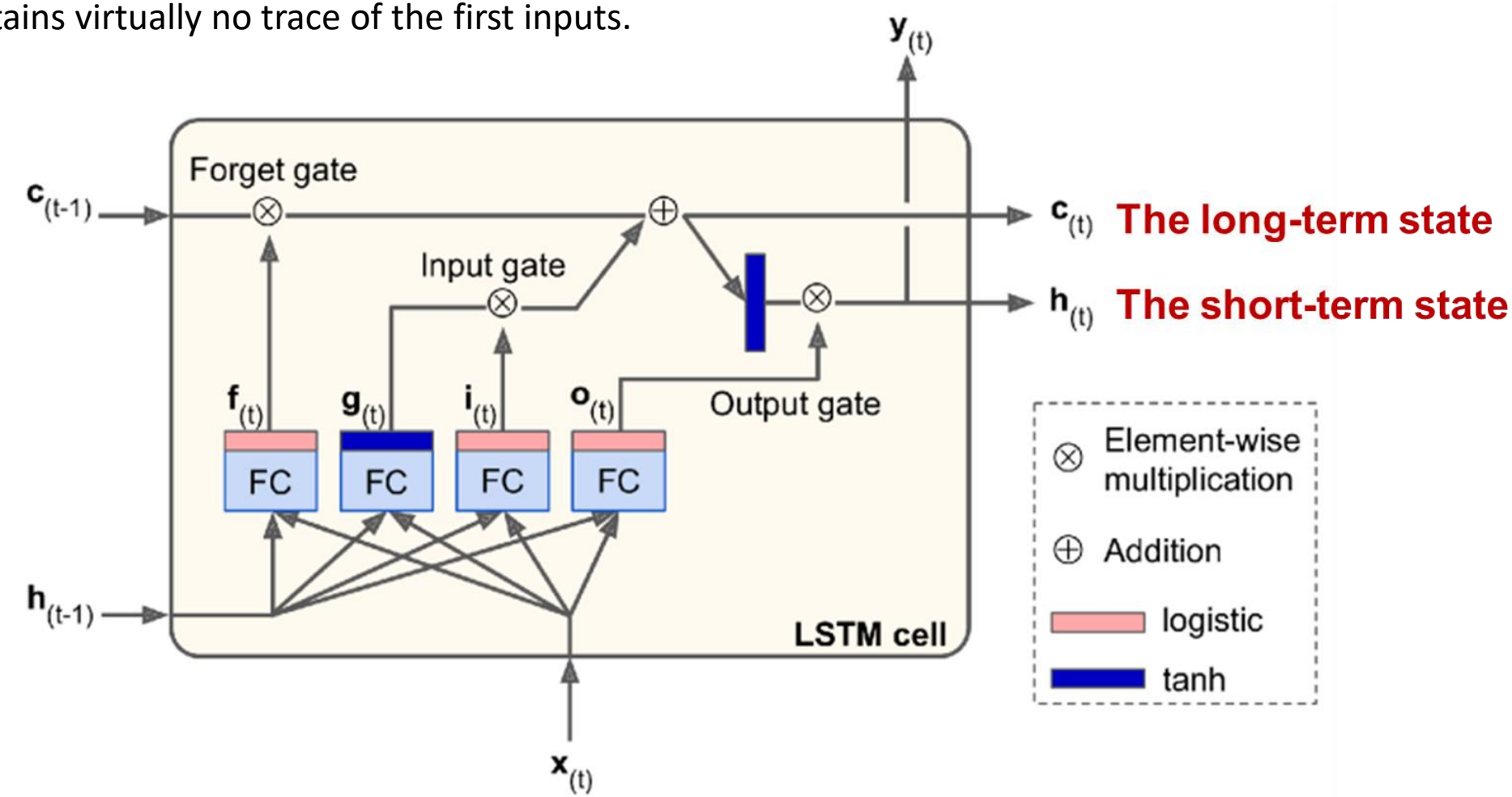
# Recurrent Neural Network(RNN)



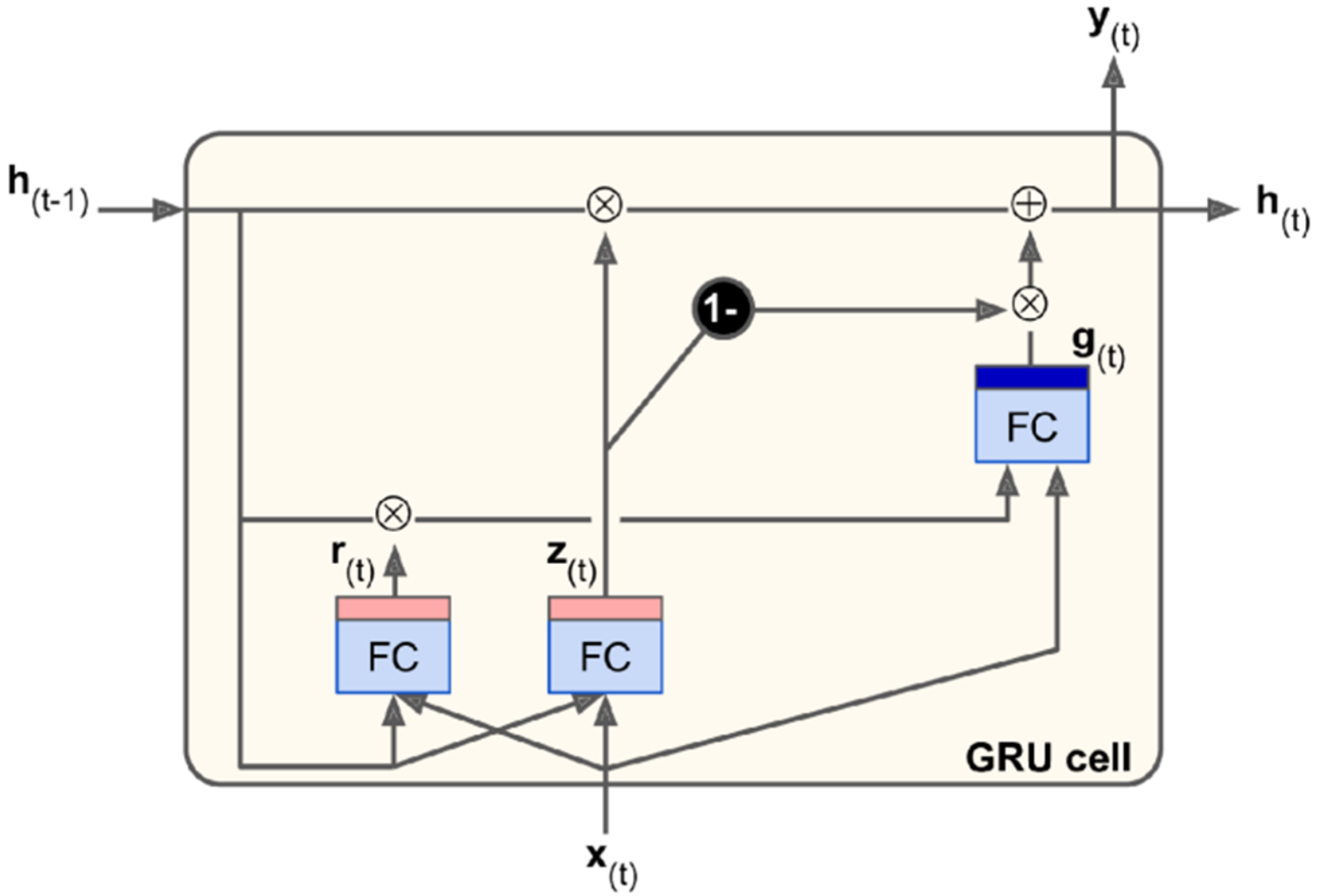

Deep RNN (left) unrolled through time (right)

# Long Short-Term Memory (LSTM)

- When the data traversing an **RNN**, **some information is lost at each time step**. After a while, the RNN's state contains virtually no trace of the first inputs.



# Gated Recurrent Unit (GRU)



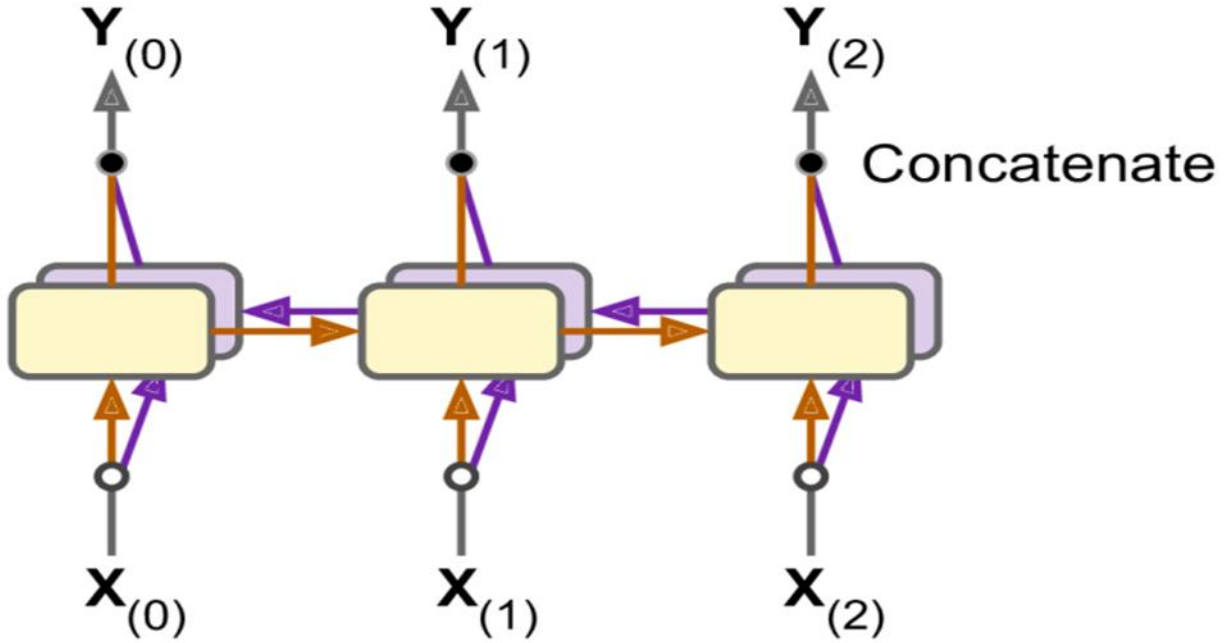
# Bidirectional RNNs

**For example**: in **Neural Machine Translation**, it is often preferable to look ahead at the next words before encoding a given word.

- Consider the phrases "the queen of the United Kingdom", "the queen of hearts", and "the queen bee": to properly encode the word “queen”, you need to look ahead.

# Bidirectional RNNs

- To implement this, **run two recurrent layers** on the same inputs, one reading the words from **left to right** and the other reading them from **right to left**. Then simply **concatenating** them.



# Self-Attention

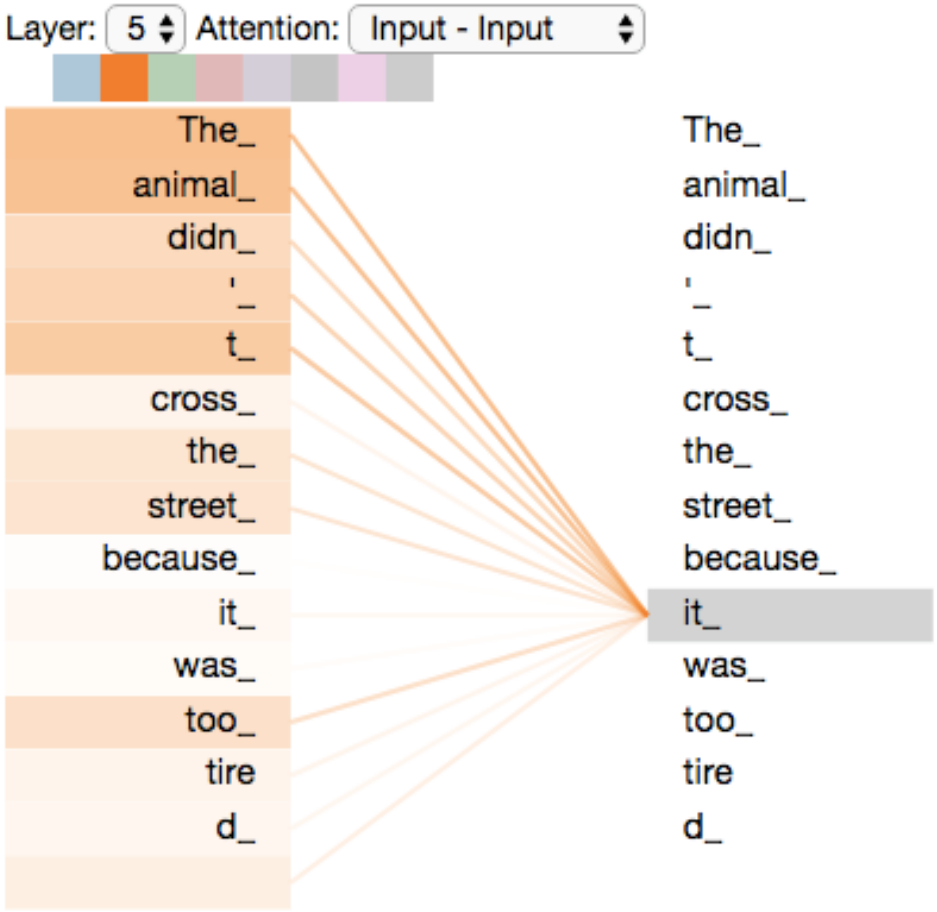
# Self-Attention

- The following sentence is an input sentence we want to translate: **"The animal didn't cross the street because it was too tired“**

- What does **"it"** in this sentence refer to?
- Is it referring to the **street** or to **the animal**? It’s a simple question to a human but not as simple to an algorithm.

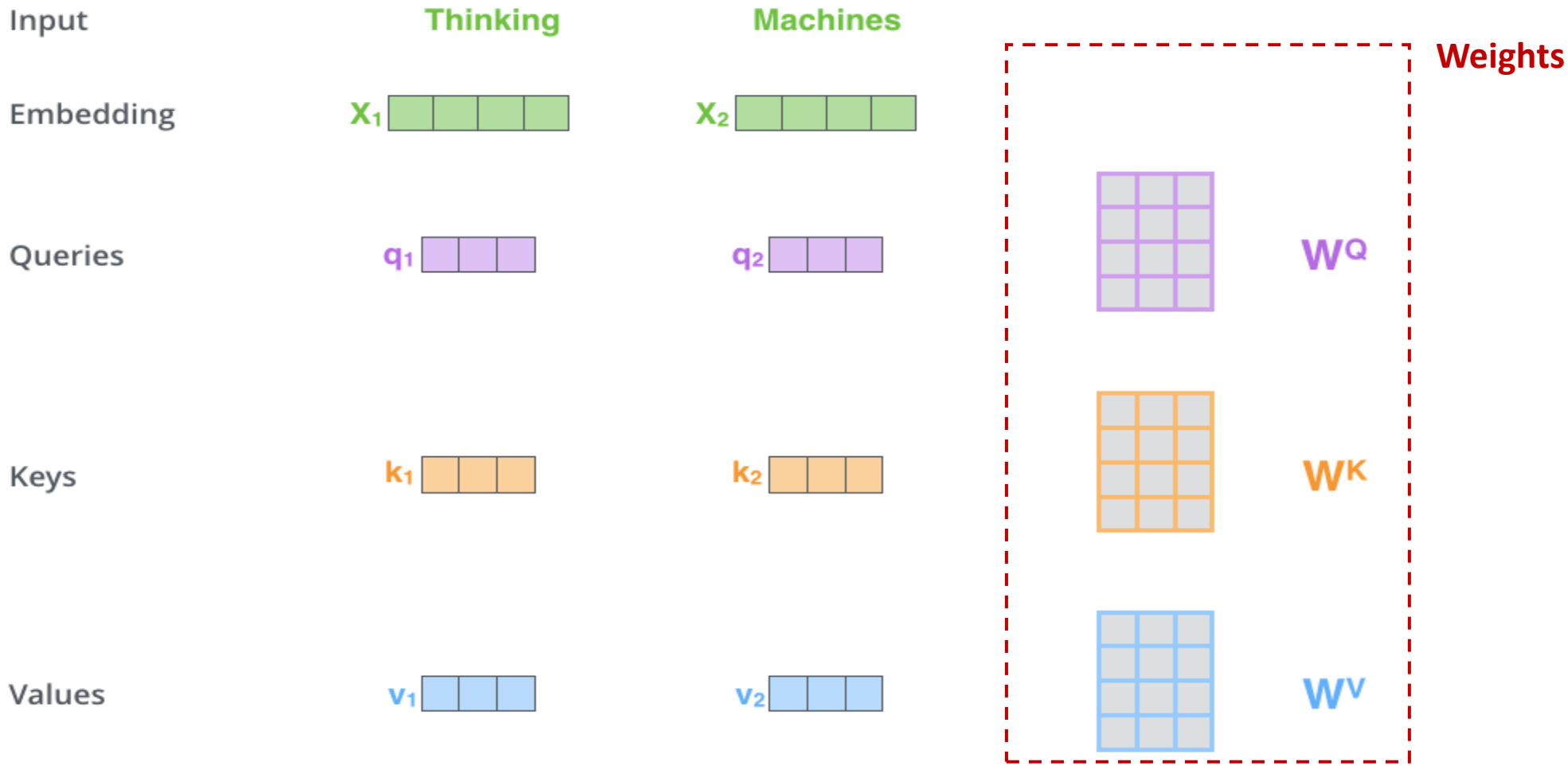
- When the model is processing the word "it", self-attention allows it to associate **"it"** with **"animal".**

# Self-Attention



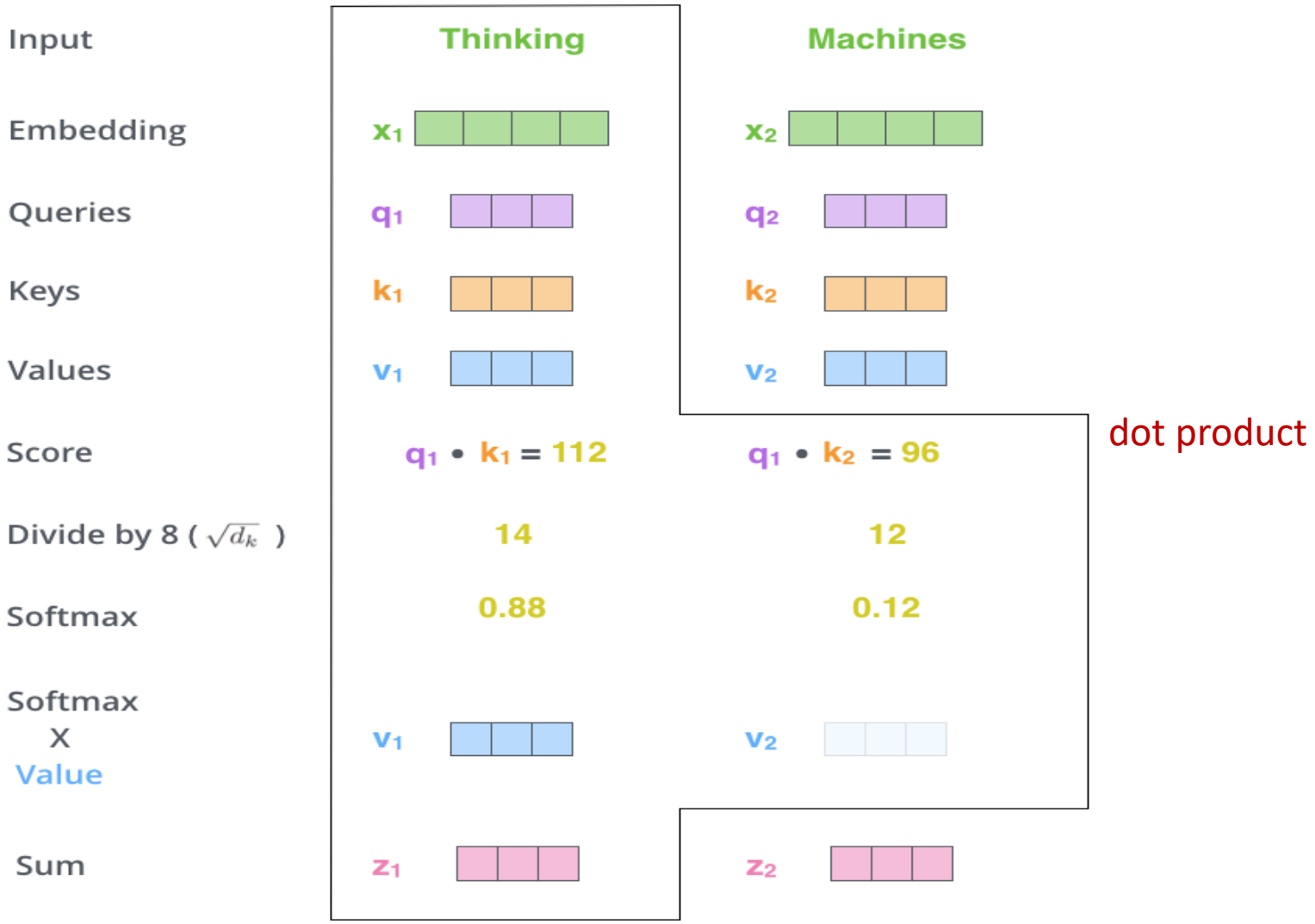

As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

# Self-Attention in Detail

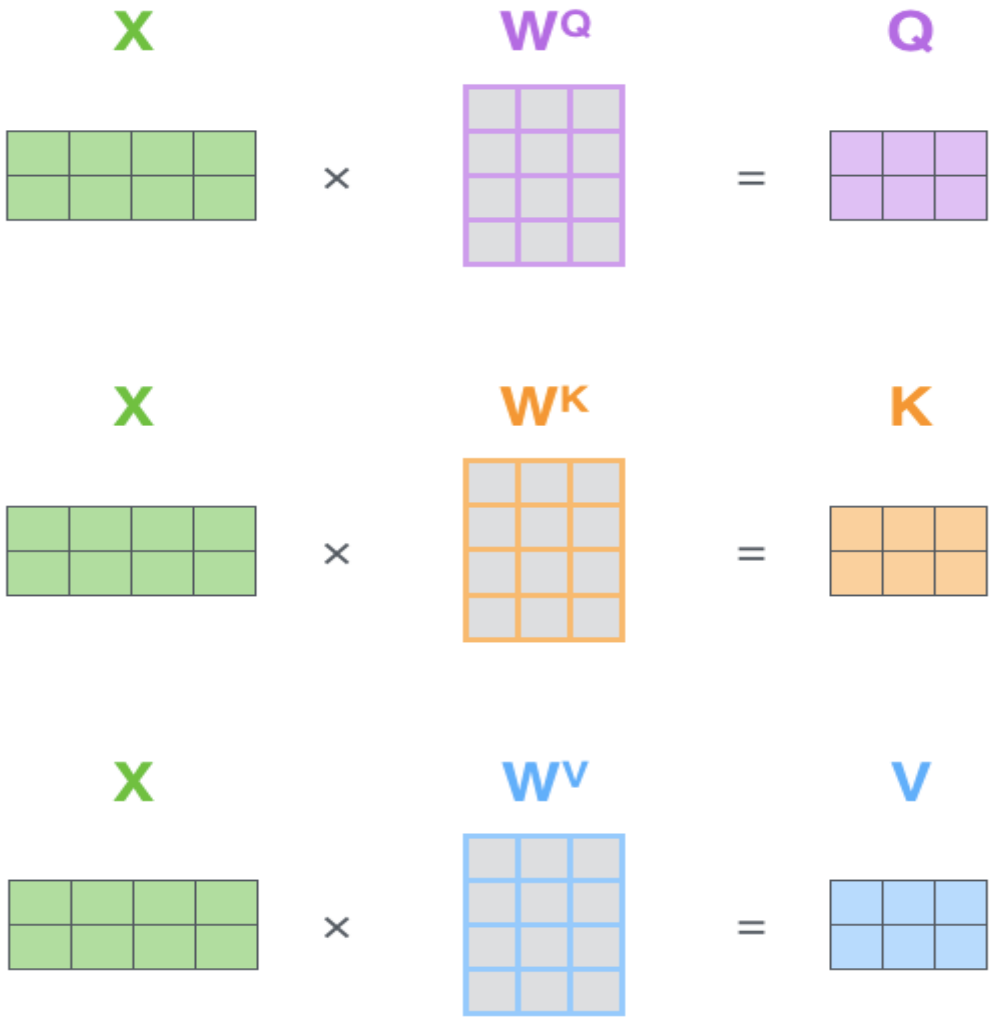



Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

# Self-Attention in Detail



# Matrix Calculation of Self-Attention




Every row in the X matrix corresponds to a word in the input sentence.

# The Attention Mechanism from Scratch

```python
from numpy import array
from numpy import random
from numpy import dot
from scipy.special import softmax

# Encoder representations of four different words
word_1 = array([1, 0, 0])
word_2 = array([0, 1, 0])
word_3 = array([1, 1, 0])
word_4 = array([0, 0, 1])

# Generating the weight matrices
W_Q = random.randint(3, size=(3, 5))
W_K = random.randint(3, size=(3, 5))
W_V = random.randint(3, size=(3, 5))

print("W_Q : \n",W_Q,"\n W_K \n",W_K,"\n W_V \n",W_V)
```

```
W_Q :
 [[2 0 2 1 2]
 [0 0 1 2 2]
 [1 2 2 0 2]]
 W_K
 [[2 1 1 0 2]
 [2 2 0 0 1]
 [0 2 2 0 2]]
 W_V
 [[2 0 0 2 2]
 [2 1 1 1 0]
 [1 0 0 1 1]]
```

```python
# Generating the queries, keys and values
query_1 = word_1 @ W_Q
key_1   = word_1 @ W_K
value_1 = word_1 @ W_V
print("query_1 : \n",query_1)
print("key_1   : \n",key_1)
print("value_1 : \n",value_1)

query_2 = word_2 @ W_Q
key_2   = word_2 @ W_K
value_2 = word_2 @ W_V

query_3 = word_3 @ W_Q
key_3   = word_3 @ W_K
value_3 = word_3 @ W_V

query_4 = word_4 @ W_Q
key_4   = word_4 @ W_K
value_4 = word_4 @ W_V

```

```
query_1 :
 [2 0 2 1 2]
key_1   :
 [2 1 1 0 2]
value_1 :
 [2 0 0 2 2]
```

```python
# Scoring the first query vector against all key vectors
score_1 = array([dot(query_1, key_1), dot(query_1, key_2),
                 dot(query_1, key_3), dot(query_1, key_4)])
print("score_1    : ",score_1)
score_2 = array([dot(query_2, key_1), dot(query_2, key_2),
                 dot(query_2, key_3), dot(query_2, key_4)])
score_3 = array([dot(query_3, key_1), dot(query_3, key_2),
                 dot(query_3, key_3), dot(query_3, key_4)])
score_4 = array([dot(query_4, key_1), dot(query_4, key_2),
                 dot(query_4, key_3), dot(query_4, key_4)])

# Computing the weights by a softmax operation
softmax_1 = softmax(score_1 / key_1.shape[0] ** 0.5)
print("softmax_1   :",softmax_1)
softmax_2 = softmax(score_2 / key_1.shape[0] ** 0.5)
softmax_3 = softmax(score_3 / key_1.shape[0] ** 0.5)
softmax_4 = softmax(score_4 / key_1.shape[0] ** 0.5)

# Computing the attention by a weighted sum of the value vectors
attention_1 = (softmax_1[0] * value_1) + (softmax_1[1] * value_2) + (softmax_1[2] * value_3) + (softmax_1[3] * value_4)
print("attention_1 : ",attention_1)
attention_2 = (softmax_2[0] * value_1) + (softmax_2[1] * value_2) + (softmax_2[2] * value_3) + (softmax_2[3] * value_4)
attention_3 = (softmax_3[0] * value_1) + (softmax_3[1] * value_2) + (softmax_3[2] * value_3) + (softmax_3[3] * value_4)
attention_4 = (softmax_4[0] * value_1) + (softmax_4[1] * value_2) + (softmax_4[2] * value_3) + (softmax_4[3] * value_4)
```

```
score_1    :  [10  6 16  8]
softmax_1   : [0.06169402 0.01031225 0.90277064 0.02522309]
attention_1 :  [3.7803182  0.9130829  0.9130829  2.86723531 1.95415241]
```

# Matrix Calculation of Self-Attention

```python
from numpy import array
from numpy import random
from numpy import dot
from scipy.special import softmax
# Encoder representations of four different words
word_1 = array([1, 0, 0])
word_2 = array([0, 1, 0])
word_3 = array([1, 1, 0])
word_4 = array([0, 0, 1])
# Stacking the word embeddings into a single array
words = array([word_1, word_2, word_3, word_4])
# Generating the weight matrices
W_Q = random.randint(3, size=(3, 5))
W_K = random.randint(3, size=(3, 5))
W_V = random.randint(3, size=(3, 5))
# Generating the queries, keys and values
Q = words @ W_Q
K = words @ W_K
V = words @ W_V
# Scoring the query vectors against all key vectors
scores = Q @ K.transpose()
# Computing the weights by a softmax operation
softmax_weights = softmax(scores / K.shape[1] ** 0.5, axis=1)
# Computing the attention by a weighted sum of the value vectors
attention = softmax_weights @ V

print(attention)
```

```
[[1.86757486 1.81117752 0.97776709 0.99354807 1.97131516]
 [1.31344188 1.56645073 0.74699115 0.96376578 1.71075693]
 [1.94407694 1.91688789 0.99257915 0.99952531 1.99210446]
 [1.31344188 1.56645073 0.74699115 0.96376578 1.71075693]]
```

# Multi-Headed Attention

Multi-Headed Attention improves the performance of the attention layer in two ways:

- It expands the model's ability to focus on **different positions**. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the actual word itself.

- It gives the attention layer **multiple representation subspaces**.

# Multi-Headed Attention

1) This is our input sentence*

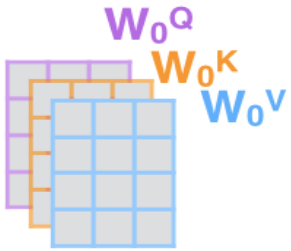
2) We embed each word*

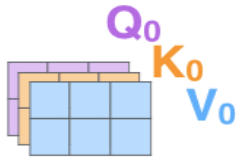
3) Split into 8 heads. We multiply X or R with weight matrices

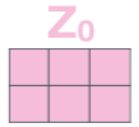
4) Calculate attention using the resulting Q/K/V matrices

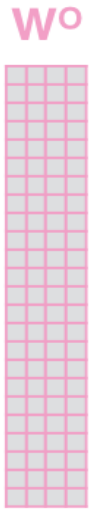
5) Concatenate the resulting Z matrices, then multiply with weight matrix $W^O$ to produce the output of the layer

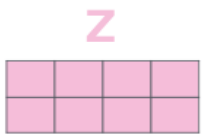
Thinking Machines

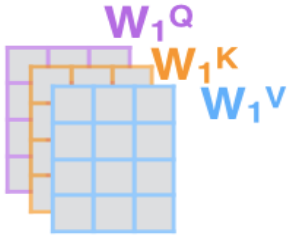



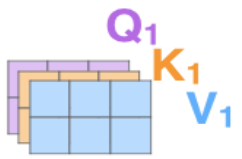



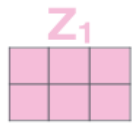



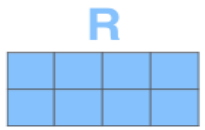



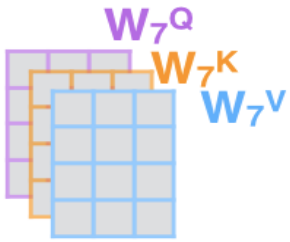

* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



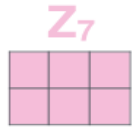



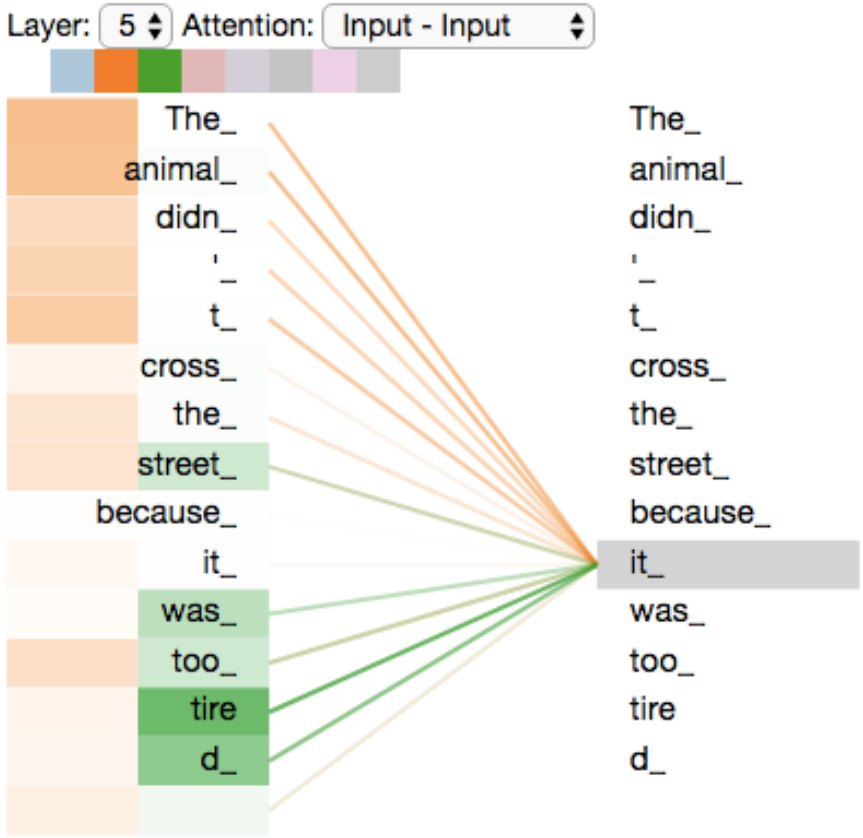



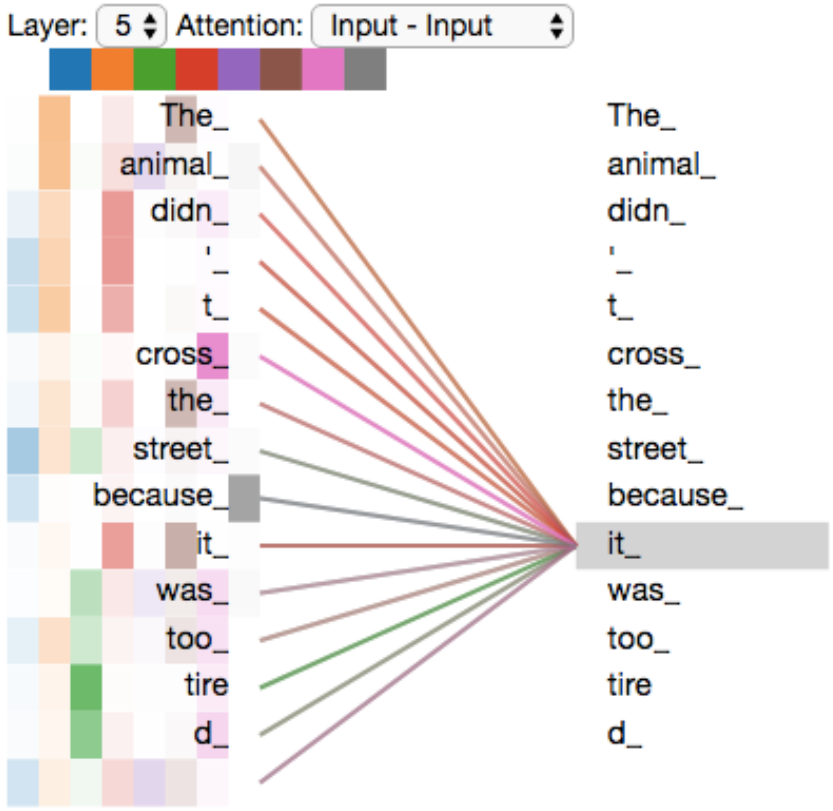




# Multi-Headed Attention




As we encode the word "**it**", one attention head is focusing most on "**the animal**", while another is focusing on "**tired**" , in a sense, the model's representation of the word "**it**" bakes in some of the representation of both "**animal**" and "**tired**".

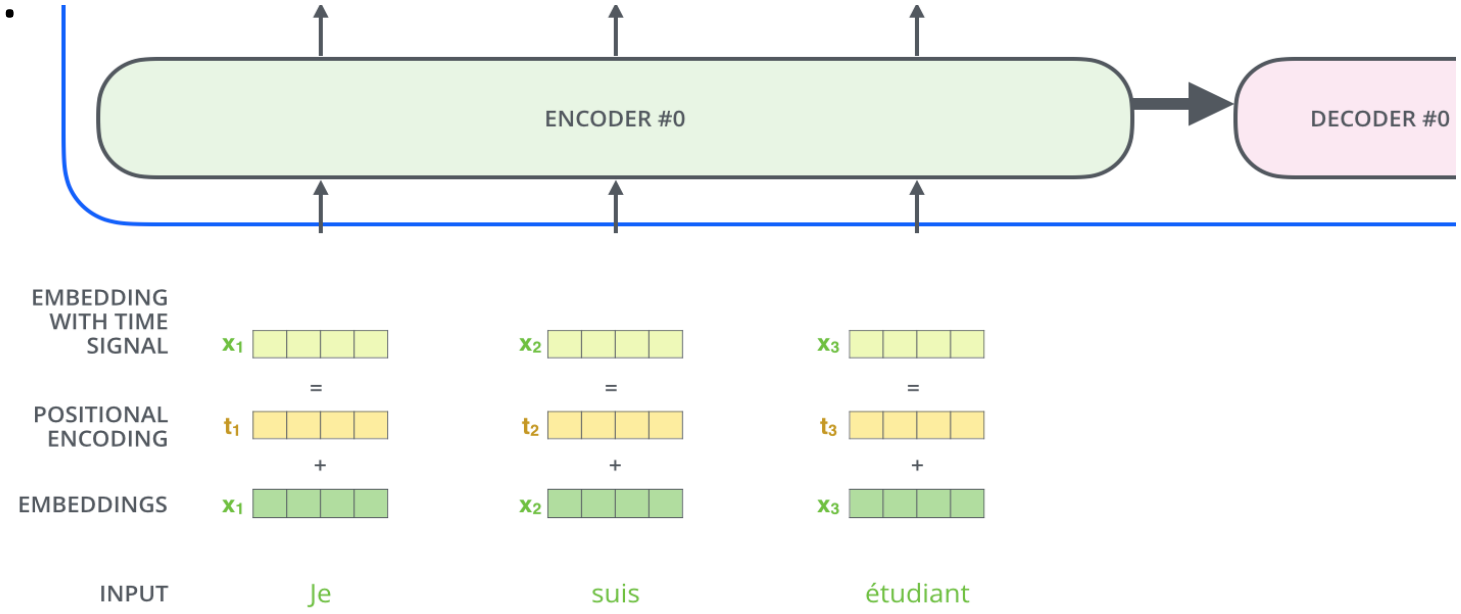



If we add all the attention heads to the picture, however, things can be harder to interpret.

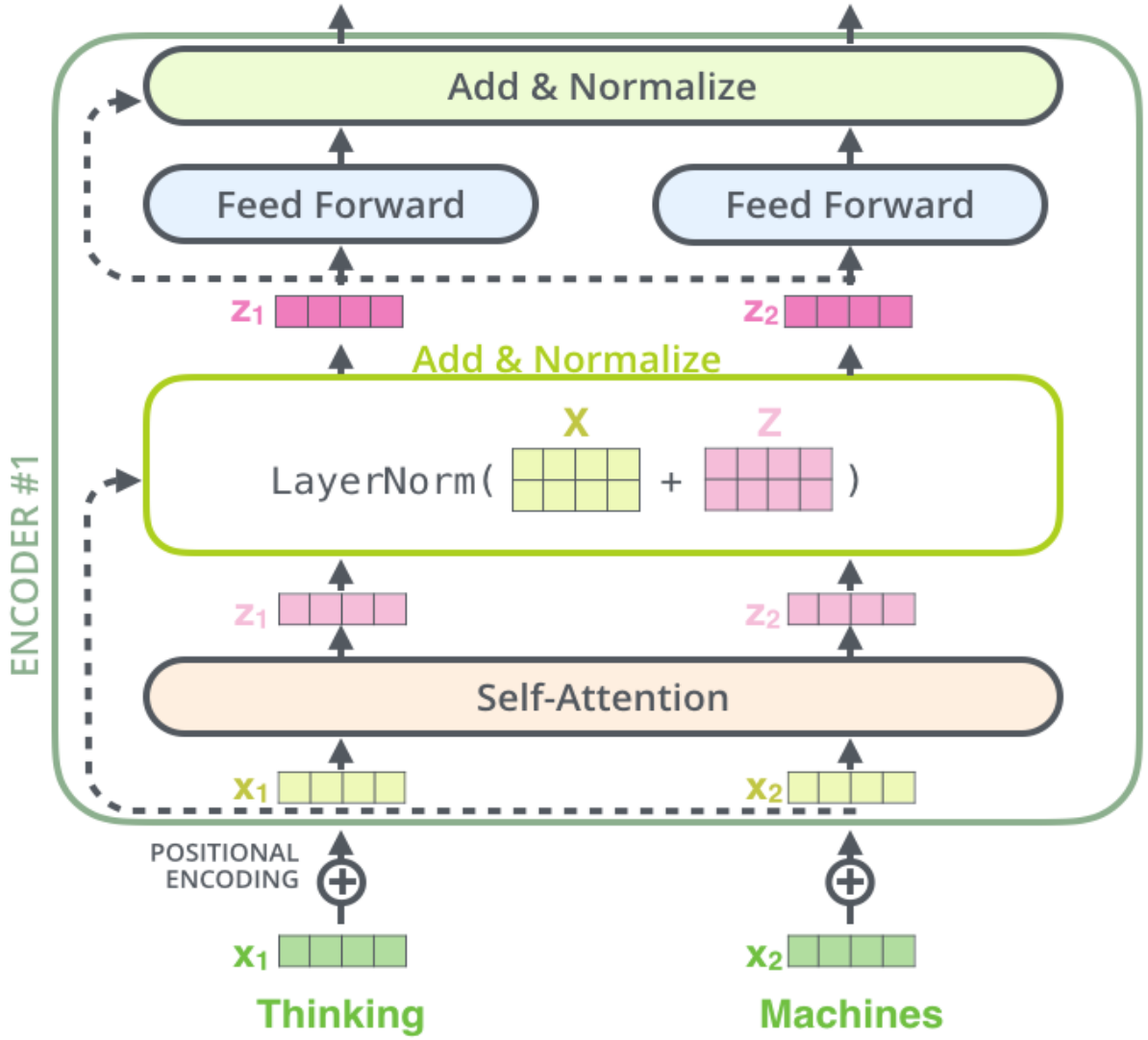
# Transformer

**Positional Encoding:**

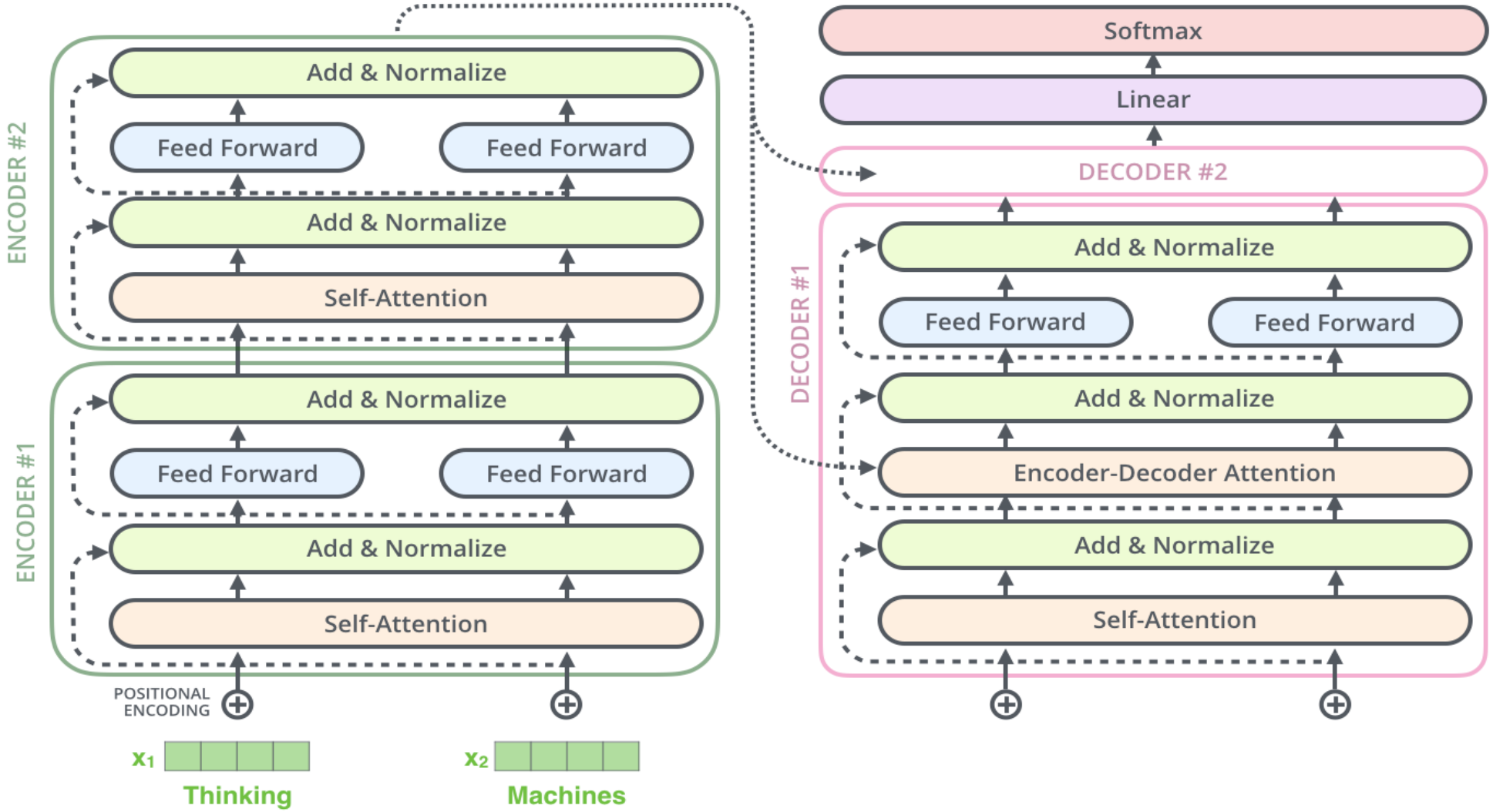
The transformer adds a **vector** to **each input embedding**. These vectors follow a specific pattern that the model learns, which helps it **determine the position** of each word or the **distance** between different words in the sequence.



# Transformer



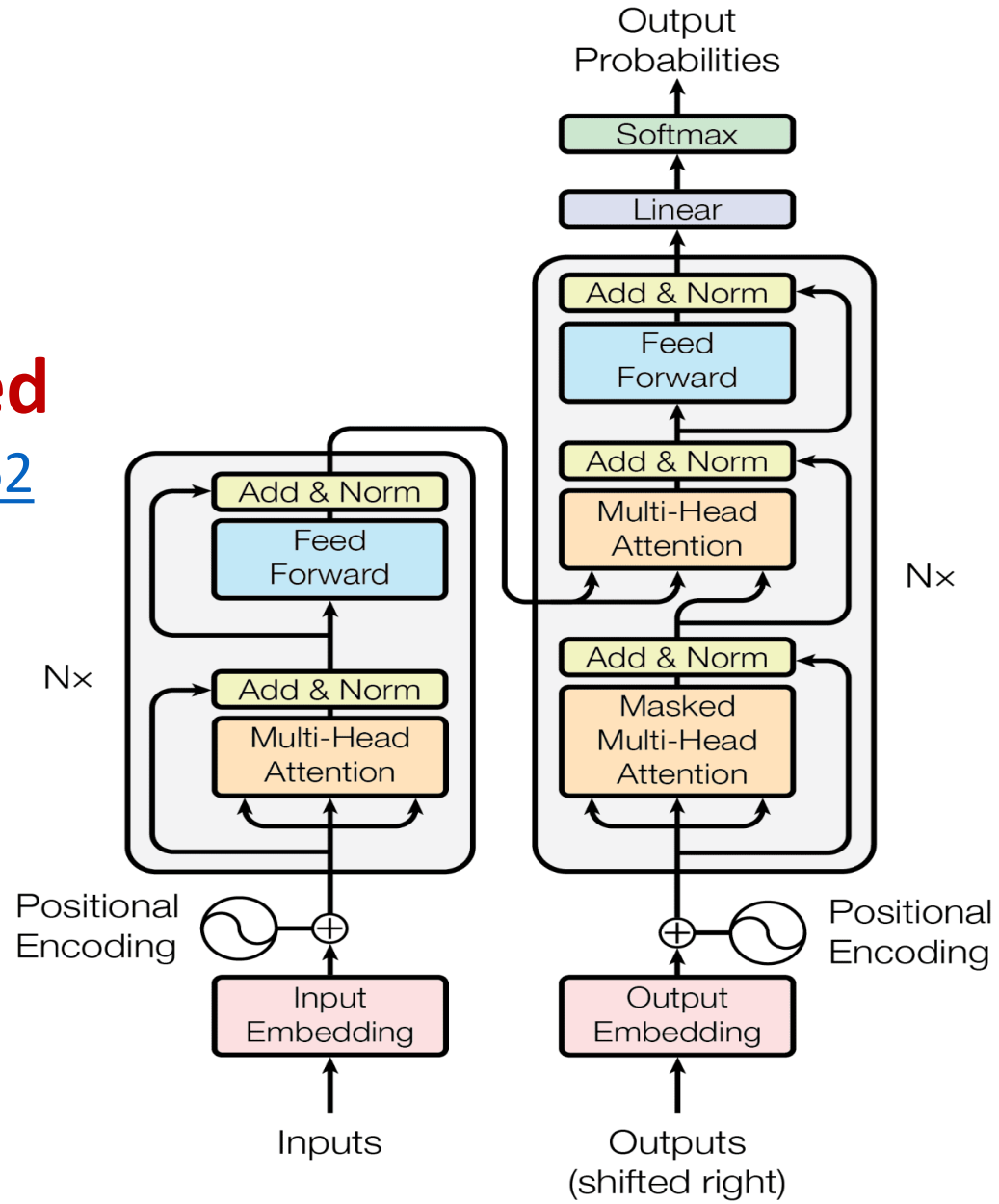
# Transformer



# Transformer

## Attention Is All You Need

https://arxiv.org/abs/1706.03762



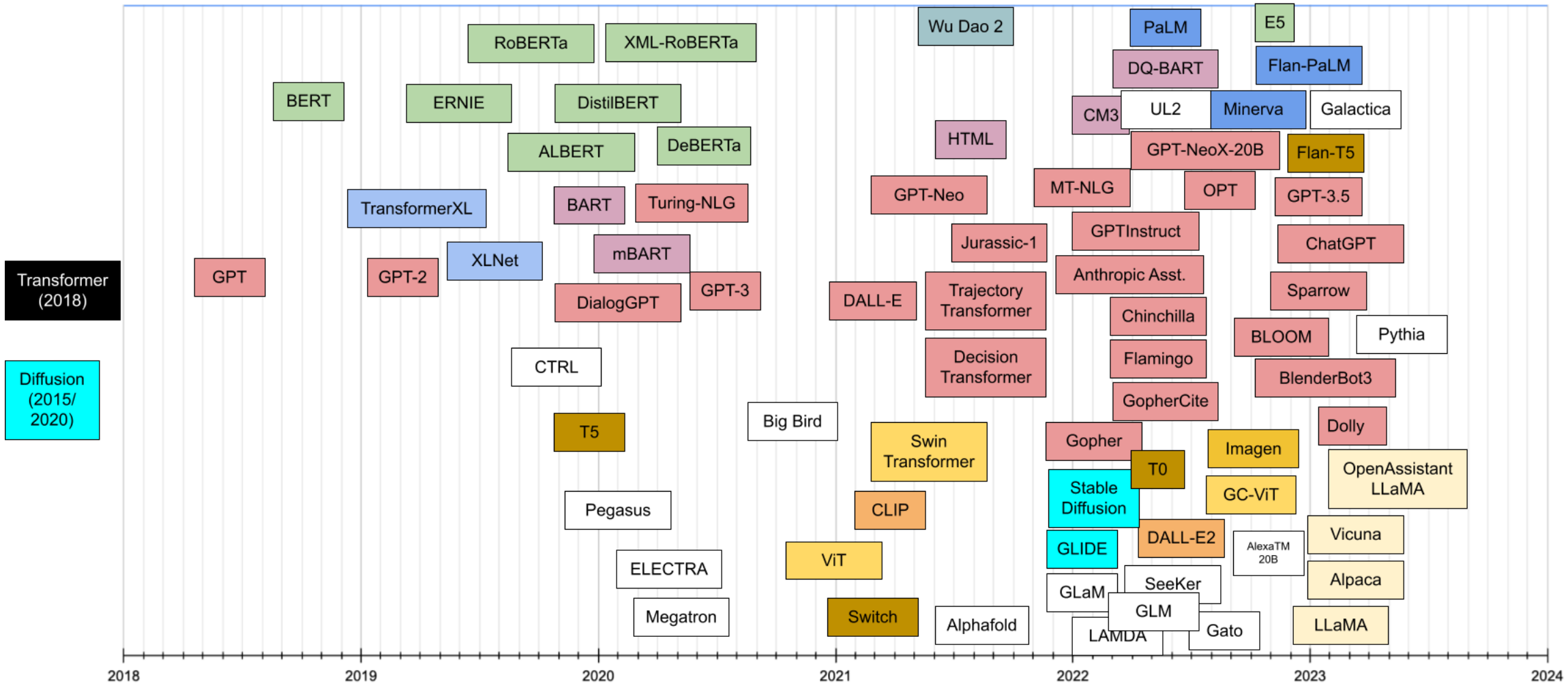
# Transformer

## The Annotated Transformer

**a line-by-line implementation**

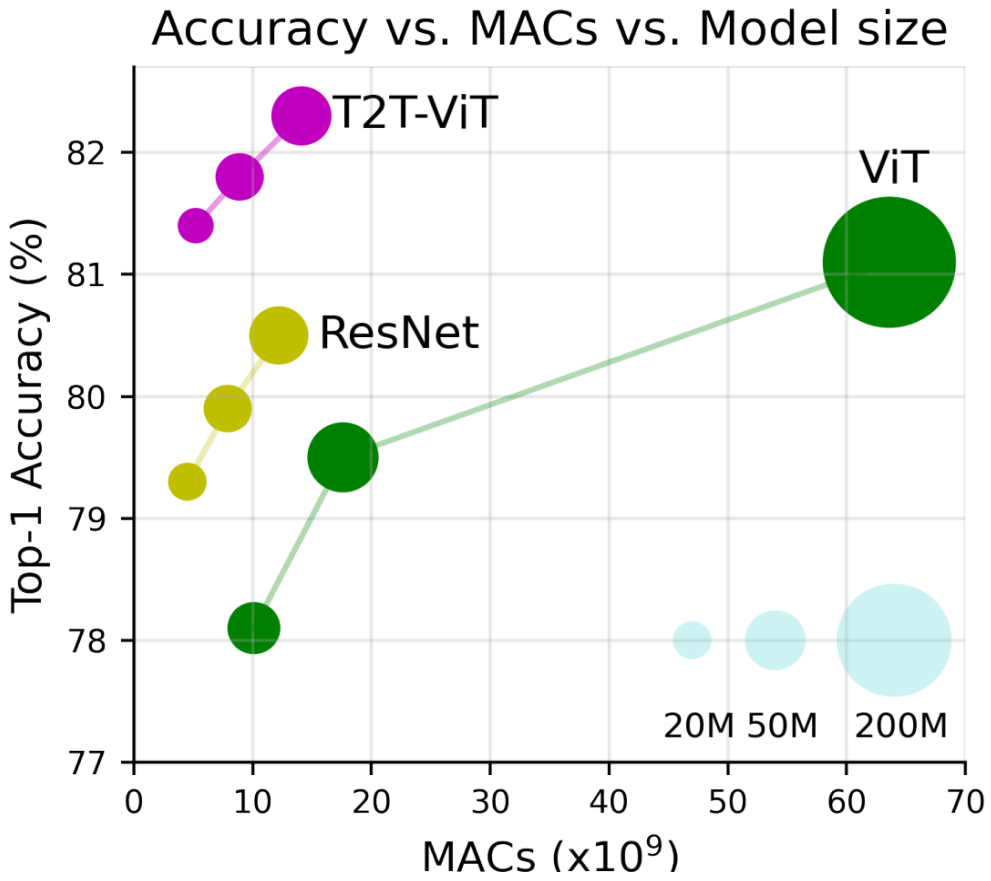
http://nlp.seas.harvard.edu/annotated-transformer/

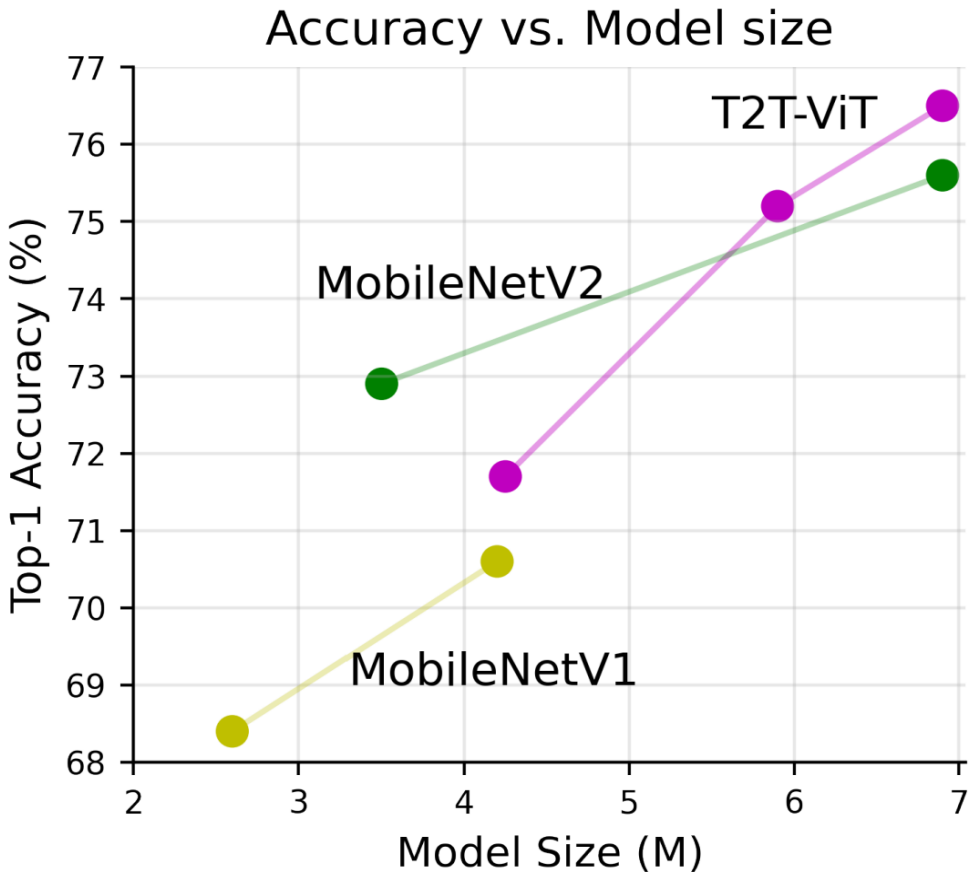
# Transformer



# Vision Transformer (ViT)

# Vision Transformers (ViTs) vs CNNs




Performance benchmark comparison of Vision Transformers (ViT) with ResNet and MobileNet when trained from scratch on ImageNet.

# Vision Transformers (ViTs) vs CNNs

The authors in [1] demonstrated that CNNs trained on ImageNet are strongly biased towards recognizing textures rather than shapes. Below is an excellent example of such a case:

(a) Texture image

| | |
|---|---|
| 81.4% | **Indian elephant** |
| 10.3% | indri |
| 8.2% | black swan |

(b) Content image

| | |
|---|---|
| 71.1% | **tabby cat** |
| 17.3% | grey fox |
| 3.3% | Siamese cat |

(c) Texture-shape cue conflict

| | |
|---|---|
| 63.9% | **Indian elephant** |
| 26.4% | indri |
| 9.6% | black swan |

*[1]: ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness. https://arxiv.org/abs/1811.12231*

# Vision Transformers (ViTs) vs CNNs

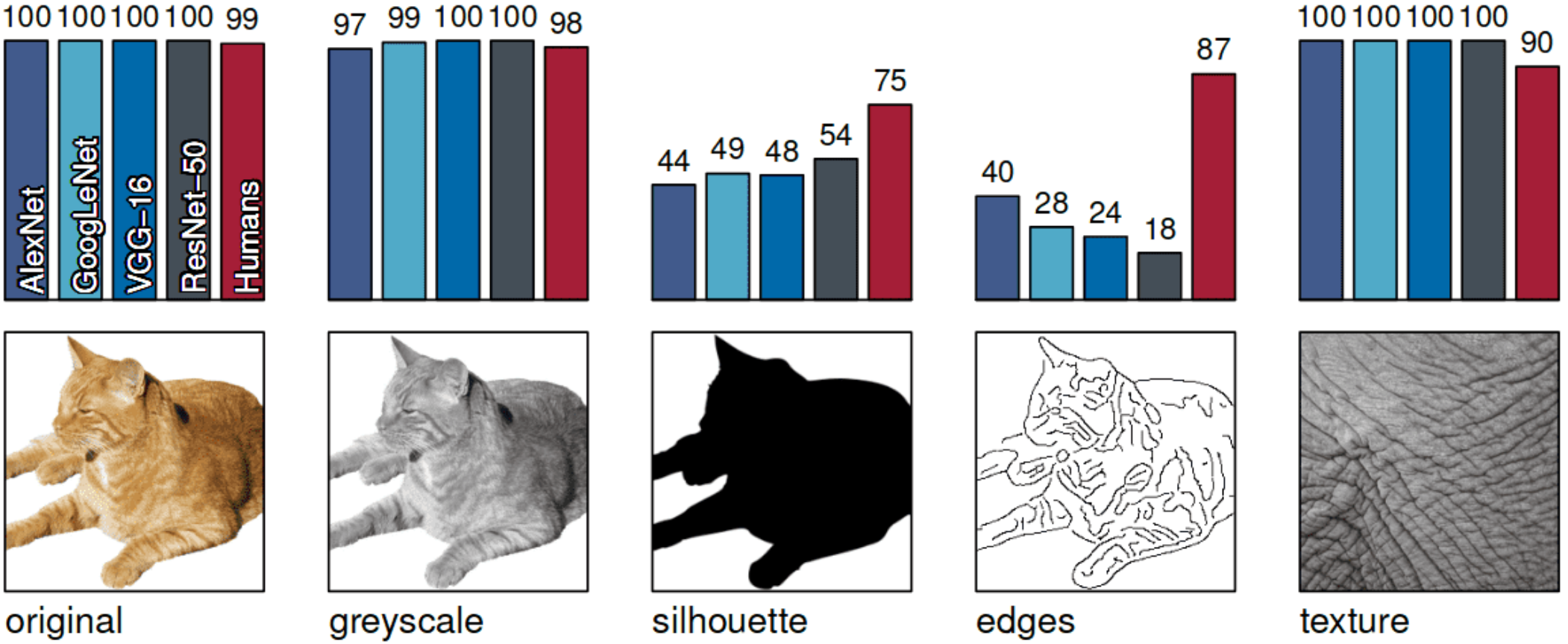
- Neuroscience studies (The importance of shape in early lexical learning [1]) showed that object shape is the single most important cue for human object recognition.

- By studying the visual pathway of humans regarding image recognition, researchers identified that the perception of object shape is invariant to most perturbations. So as far as we know, the shape is the most reliable cue.

- Intuitively, the object shape remains relatively stable, while other cues can be easily distorted by all sorts of noise [2].

*1: https://psycnet.apa.org/doi/10.1016/0885-2014(88)90014-7*
*2: https://arxiv.org/abs/1811.12231*

# Vision Transformers (ViTs) vs CNNs



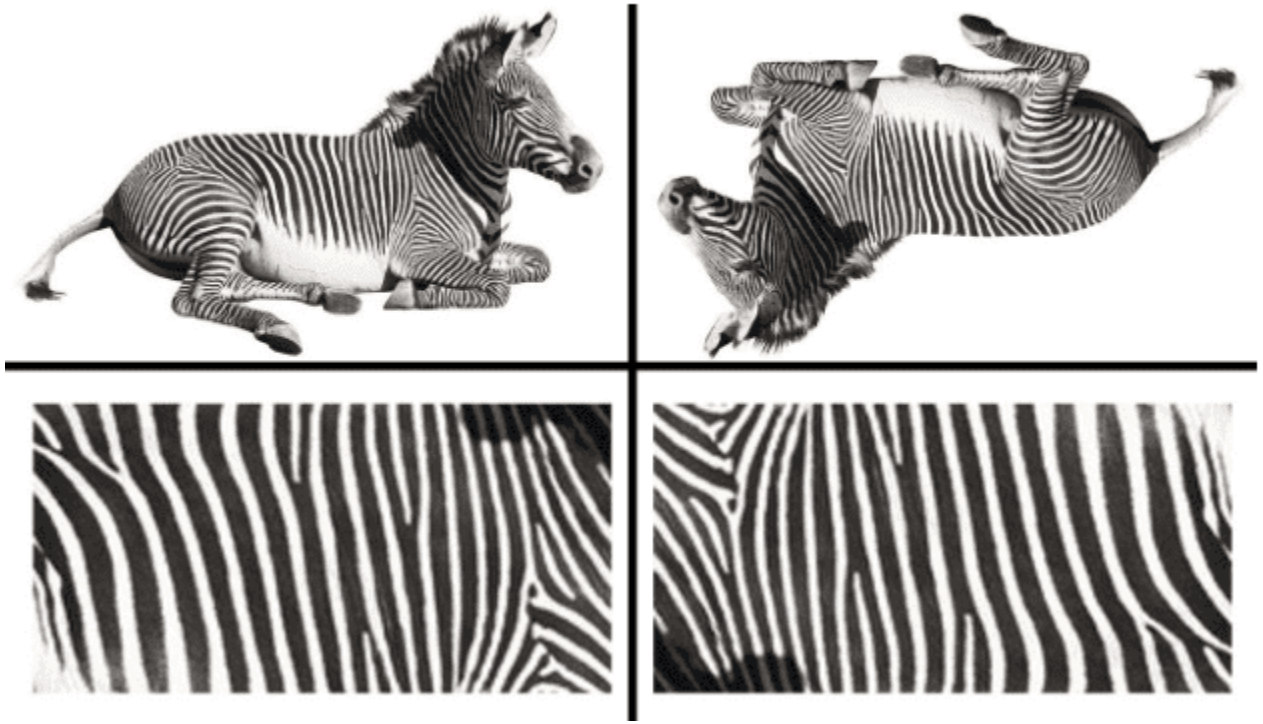

Accuracies and example stimuli for five different experiments without cue conflict.

Source: https://arxiv.org/abs/1811.12231

# Vision Transformers (ViTs) vs CNNs

- The texture is not sufficient for determining whether the zebra is rotated. Thus, predicting rotation requires modeling shape, to some extent.

- The object's shape can be invariant to rotations.

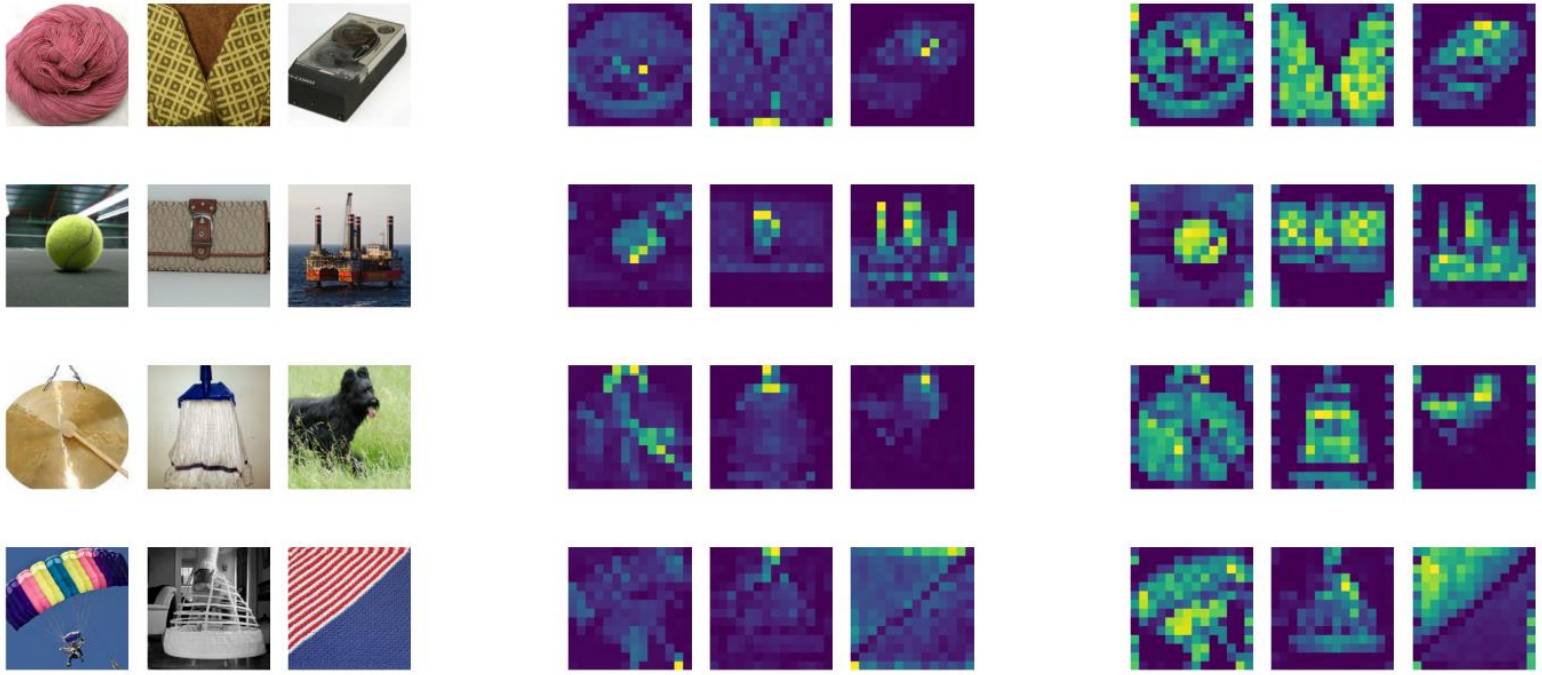


# Vision Transformers (ViTs) vs CNNs

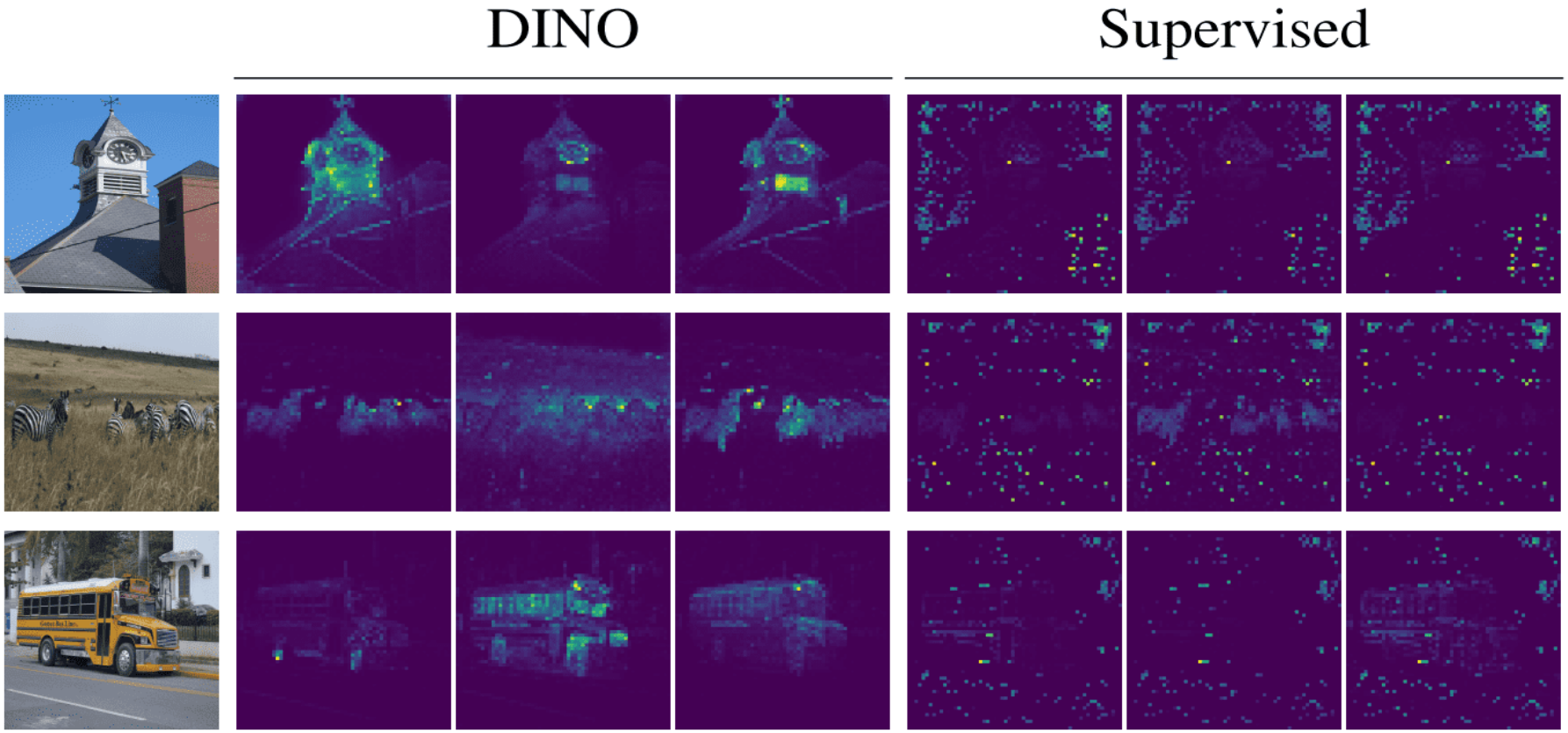
The self-attention captures long-range dependencies and contextual information in the input data.
The self-attention mechanism allows a ViT model to attend to different regions of the input data based on their relevance to the task at hand.

Raw images (Left) and attention maps of ViT-S/16 with (Right) and without (Middle).

https://arxiv.org/abs/2106.01548



# Vision Transformers (ViTs) vs CNNs



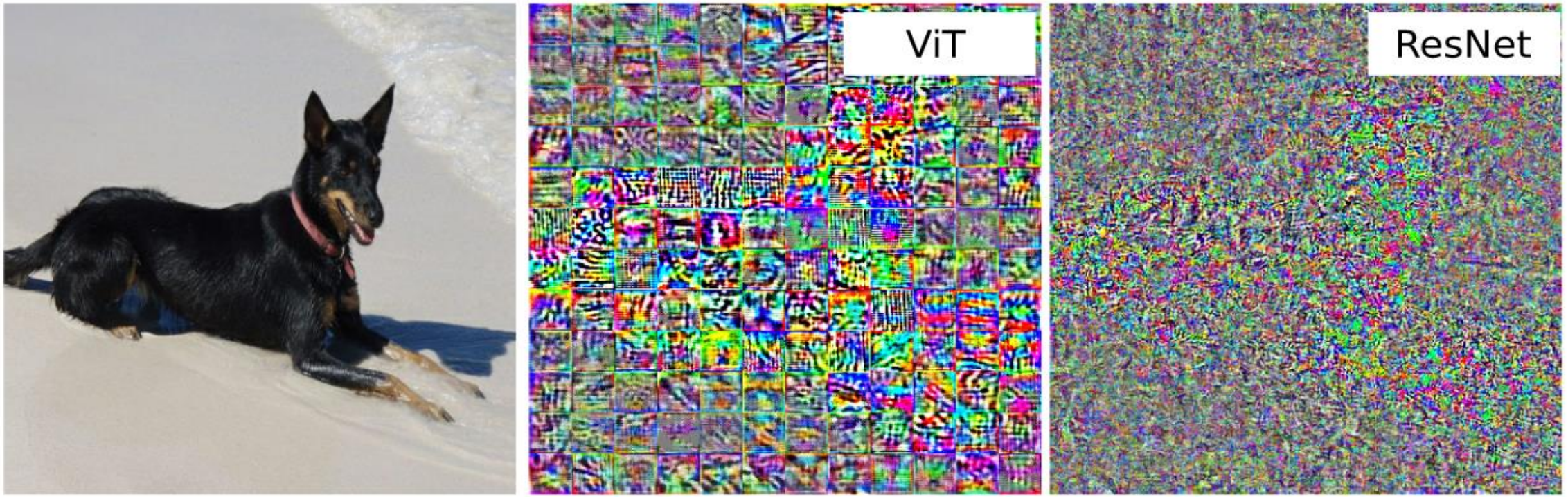

The authors in [1] looked at the self-attention of the CLS token on the heads of the last layer. Crucially, no labels are used during the self-supervised training. These maps demonstrate that the learned class-specific features lead to remarkable unsupervised segmentation masks and visibly correlate with the shape of semantic objects in the images.

1: **Self-Supervised Vision Transformers with DINO** https://arxiv.org/abs/2104.14294

# Vision Transformers (ViTs) vs CNNs

- The adversarial perturbations computed for a ViT and a ResNet model.

- The adversarial perturbations are qualitatively very different even though both models may perform similarly in image recognition.

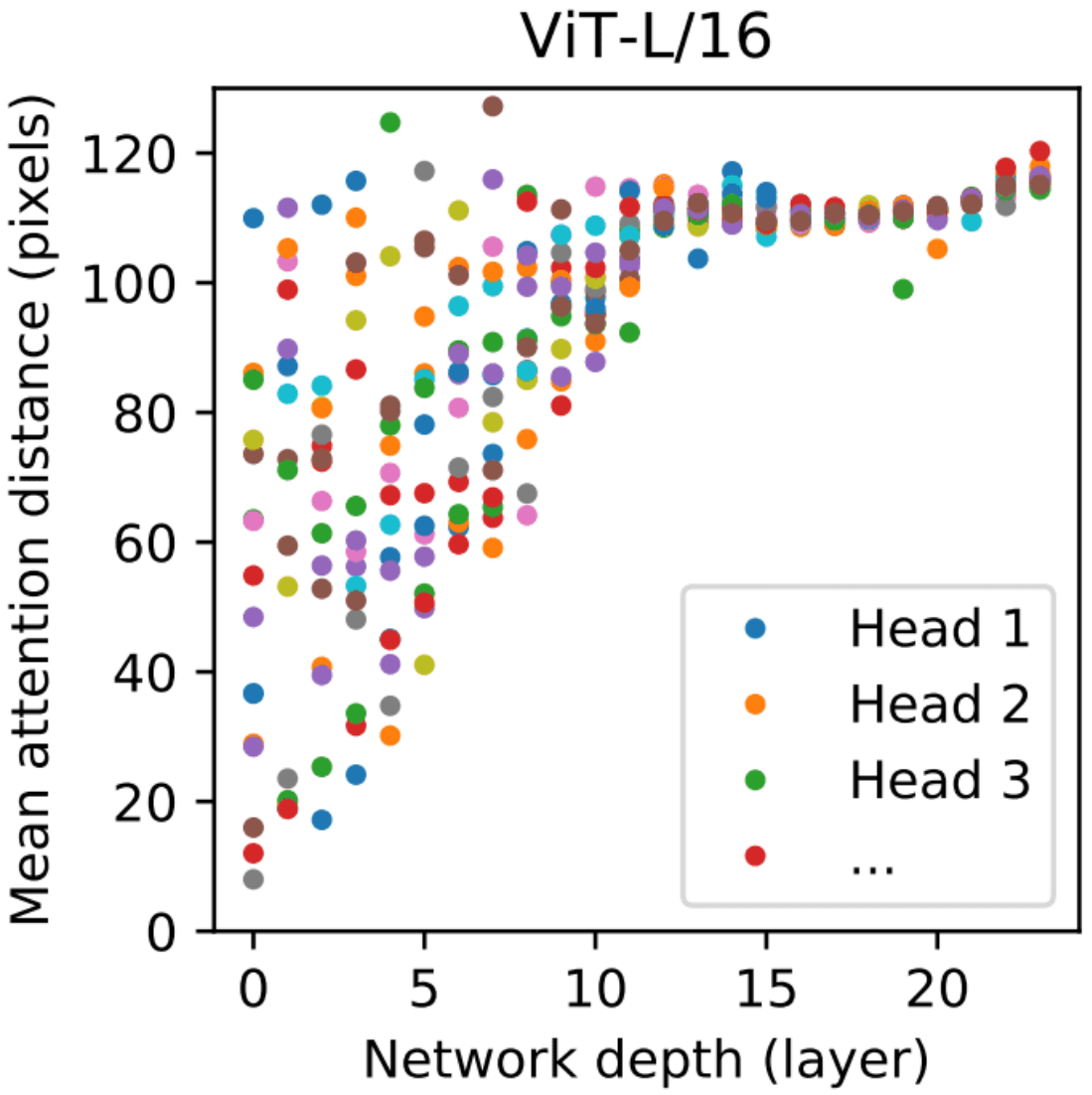



ViTs and ResNets process their inputs very differently. https://arxiv.org/abs/2103.14586

# Vision Transformers (ViTs) vs CNNs

- The transformer can attend to all the tokens (image patches) at each block by design. The originally proposed ViT model in [1] already demonstrated that heads from early layers tend to attend to far-away pixels, while heads from later layers do not.

*How heads of different layers attend to their surrounding pixels [1].*



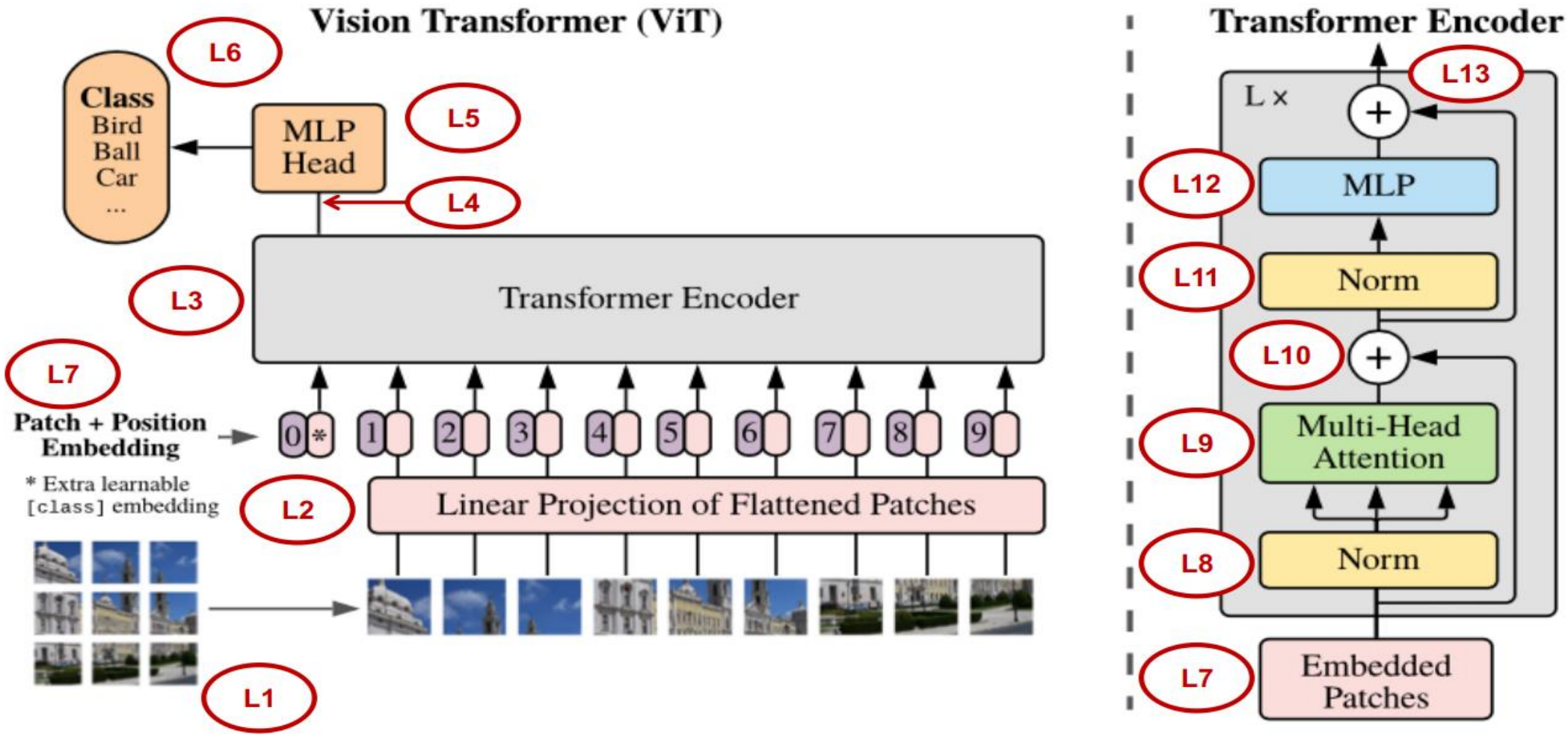

[1]: https://arxiv.org/abs/2010.11929

# Vision Transformers (ViTs)

## How the Vision Transformer Works:

1. **Split** an image into patches
2. **Flatten** the patches
3. Produce lower-dimensional linear **embeddings** from the flattened patches
4. Add **positional embeddings**
5. Feed the sequence as an **input** to a standard **transformer** encoder
6. **Pretrain** the model with image labels (fully **supervised** on a huge dataset)
7. **Finetune** on the downstream dataset for image classification

# Vision Transformers (ViTs)




*https://github.com/hichemfelouat/my-codes-of-machine-learning/blob/master/Vision_Transformer_(ViT)_for_Image_Classification_(cifar10_dataset).ipynb*

# Vision Transformers (ViTs)

**Table 1** – Image classification benchmarks on **ImageNet-1K** dataset (Deng et al., 2009).

| Method | Param (M) | FLOPs (G) | Image Size | Top-1 (%) |
|---|---|---|---|---|
| ResMLP-S12 (Touvron et al., 2021a) | 15 | 3.0 | $224^2$ | 76.6 |
| PVT-v2-B1 (Wang et al., 2022) | 13 | 2.1 | $224^2$ | 78.7 |
| **GC ViT-XXT** | 12 | 2.1 | $224^2$ | **79.8** |
| DeiT-Small/16 (Touvron et al., 2021b) | 22 | 4.6 | $224^2$ | 79.9 |
| T2T-ViT-14 (Yuan et al., 2021) | 22 | 5.2 | $224^2$ | 81.5 |
| **GC ViT-XT** | 20 | 2.6 | $224^2$ | **82.0** |
| ResNet50 (He et al., 2016) | 25 | 4.1 | $224^2$ | 76.1 |
| Swin-T (Liu et al., 2021) | 29 | 4.5 | $224^2$ | 81.3 |
| CoAtNet-0 (Dai et al., 2021) | 25 | 4.2 | $224^2$ | 81.6 |
| PVT-v2-B2 (Wang et al., 2022) | 25 | 4.0 | $224^2$ | 82.0 |
| ConvNeXt-T (Liu et al., 2022) | 29 | 4.5 | $224^2$ | 82.1 |
| Focal-T (Yang et al., 2021b) | 29 | 4.9 | $224^2$ | 82.2 |
| CSwin-T (Dong et al., 2022) | 23 | 4.3 | $224^2$ | 82.7 |
| **GC ViT-T** | 28 | 4.7 | $224^2$ | **83.4** |
| **GC ViT-T2** | 34 | 5.5 | $224^2$ | **83.7** |
| ResNet-101 (He et al., 2016) | 44 | 7.9 | $224^2$ | 77.4 |
| Swin-S (Liu et al., 2021) | 50 | 8.7 | $224^2$ | 83.0 |
| ConvNeXt-S (Liu et al., 2022) | 50 | 8.7 | $224^2$ | 83.1 |
| PVT-v2-B3 (Wang et al., 2022) | 45 | 6.9 | $224^2$ | 83.2 |
| CoAtNet-1 (Dai et al., 2021) | 42 | 8.4 | $224^2$ | 83.3 |
| Focal-S (Yang et al., 2021b) | 51 | 9.1 | $224^2$ | 83.5 |
| CSwin-S (Dong et al., 2022) | 35 | 6.9 | $224^2$ | 83.6 |
| **GC ViT-S** | 51 | 8.5 | $224^2$ | **84.3** |
| **GC ViT-S2** | 68 | 10.7 | $224^2$ | **84.7** |
| ResNet-152 (He et al., 2016) | 60 | 11.6 | $224^2$ | 78.3 |
| ViT-Base/16 (Dosovitskiy et al., 2020) | 86 | 17.6 | $224^2$ | 77.9 |
| DeiT-Base/16 (Touvron et al., 2021b) | 86 | 17.6 | $224^2$ | 81.8 |
| Swin-B (Liu et al., 2021) | 88 | 15.4 | $224^2$ | 83.3 |
| CoAtNet-2 (Dai et al., 2021) | 42 | 8.4 | $224^2$ | 83.3 |
| ConvNeXt-B (Liu et al., 2022) | 89 | 15.4 | $224^2$ | 83.8 |
| Focal-B (Yang et al., 2021b) | 90 | 16.0 | $224^2$ | 83.8 |
| PVT-v2-B5 (Wang et al., 2022) | 82 | 11.8 | $224^2$ | 83.8 |
| CSwin-B (Dong et al., 2022) | 78 | 15.0 | $224^2$ | 84.2 |
| BoTNet (Dong et al., 2022) | 79 | 19.3 | $256^2$ | 84.2 |
| **GC ViT-B** | 90 | 14.8 | $224^2$ | **84.9** |
| ConvNeXt-L (Liu et al., 2022) | 198 | 34.4 | $224^2$ | 84.3 |
| CoAtNet-3 (Dai et al., 2021) | 168 | 34.7 | $224^2$ | 84.5 |
| **GC ViT-L** | 201 | 32.6 | $224^2$ | **85.6** |

**Global Context Vision Transformer (GC ViT):**
*https://github.com/NVlabs/GCViT*

# Large Language Models

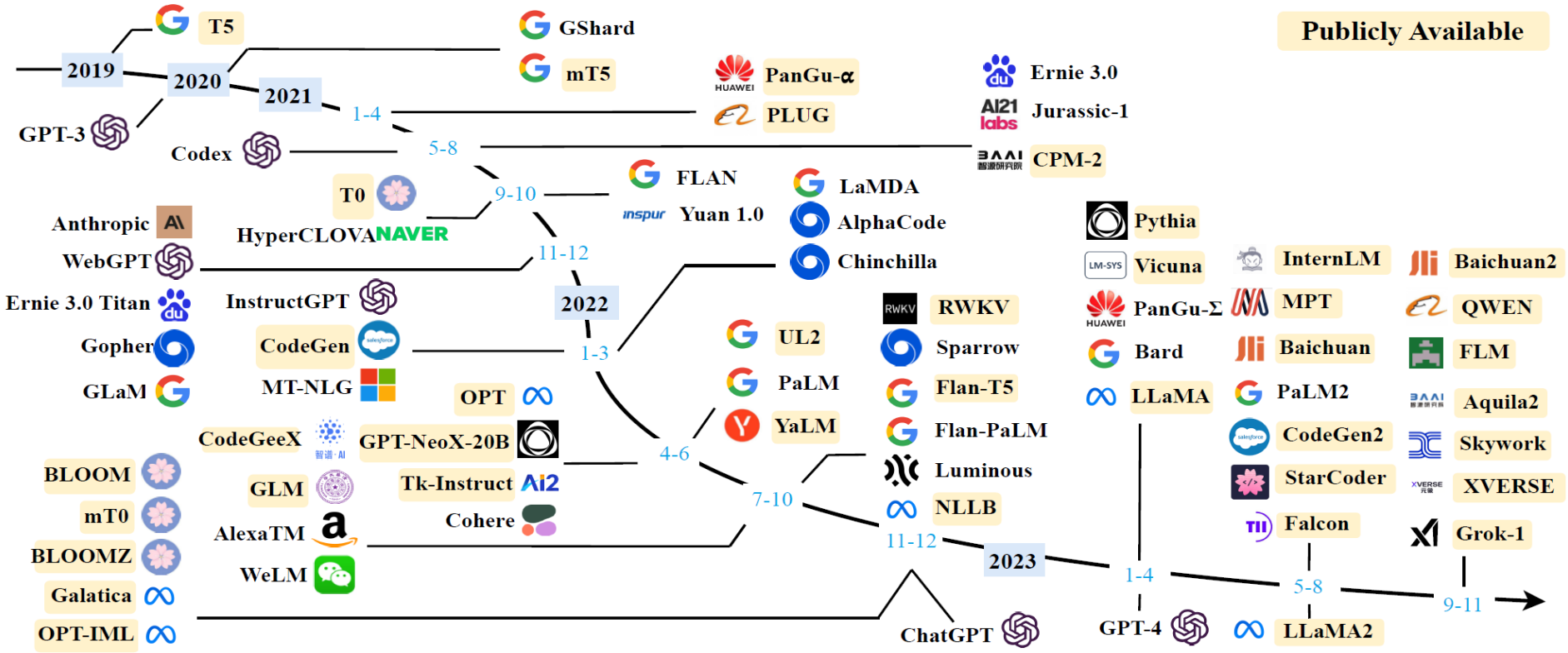
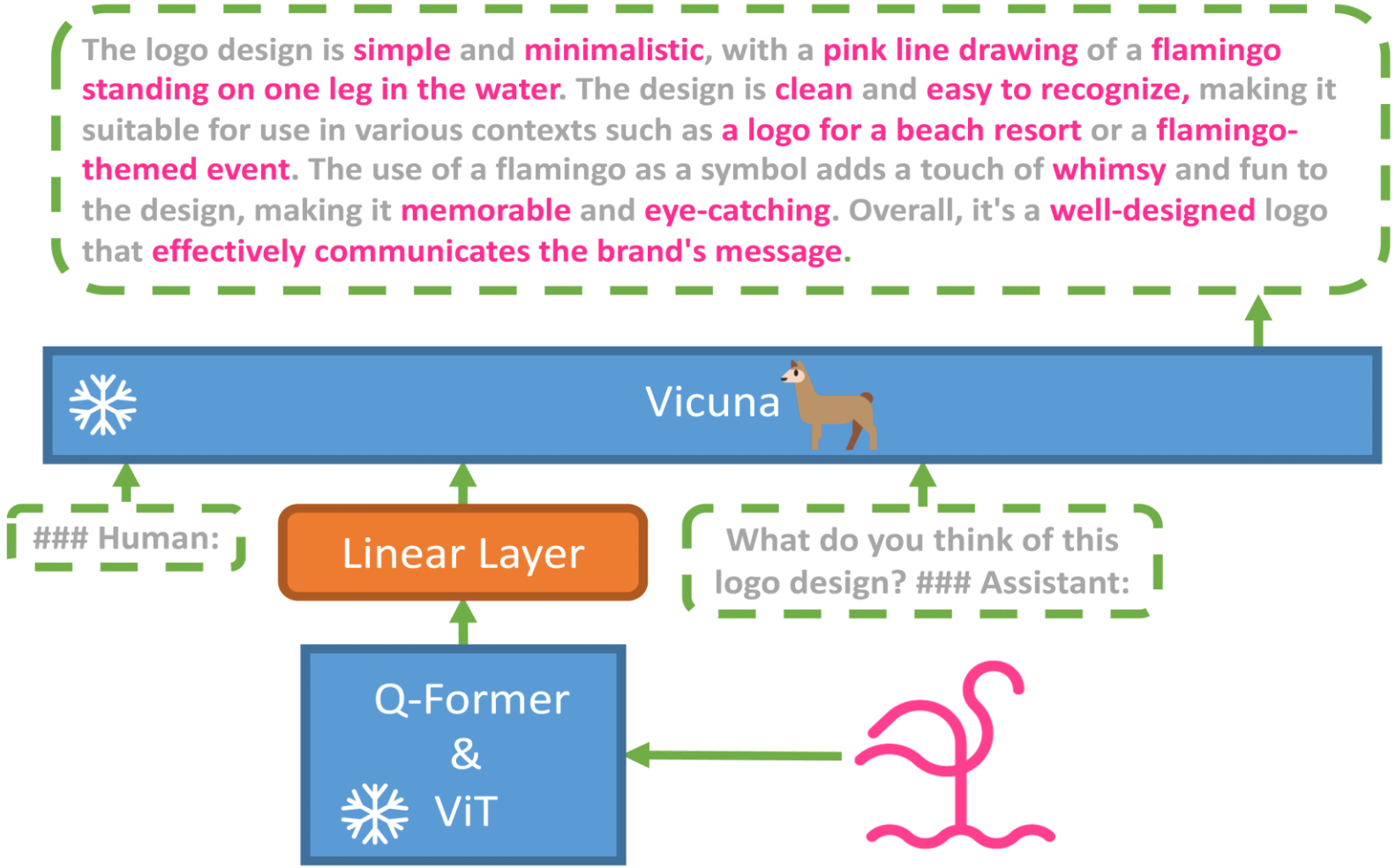



Fig. 3: A timeline of existing large language models (having a size larger than 10B) in recent years. The timeline was established mainly according to the release date (*e.g.*, the submission date to arXiv) of the technical paper for a model. If there was not a corresponding paper, we set the date of a model as the earliest time of its public release or announcement. We mark the LLMs with publicly available model checkpoints in yellow color. Due to the space limit of the figure, we only include the LLMs with publicly reported evaluation results.

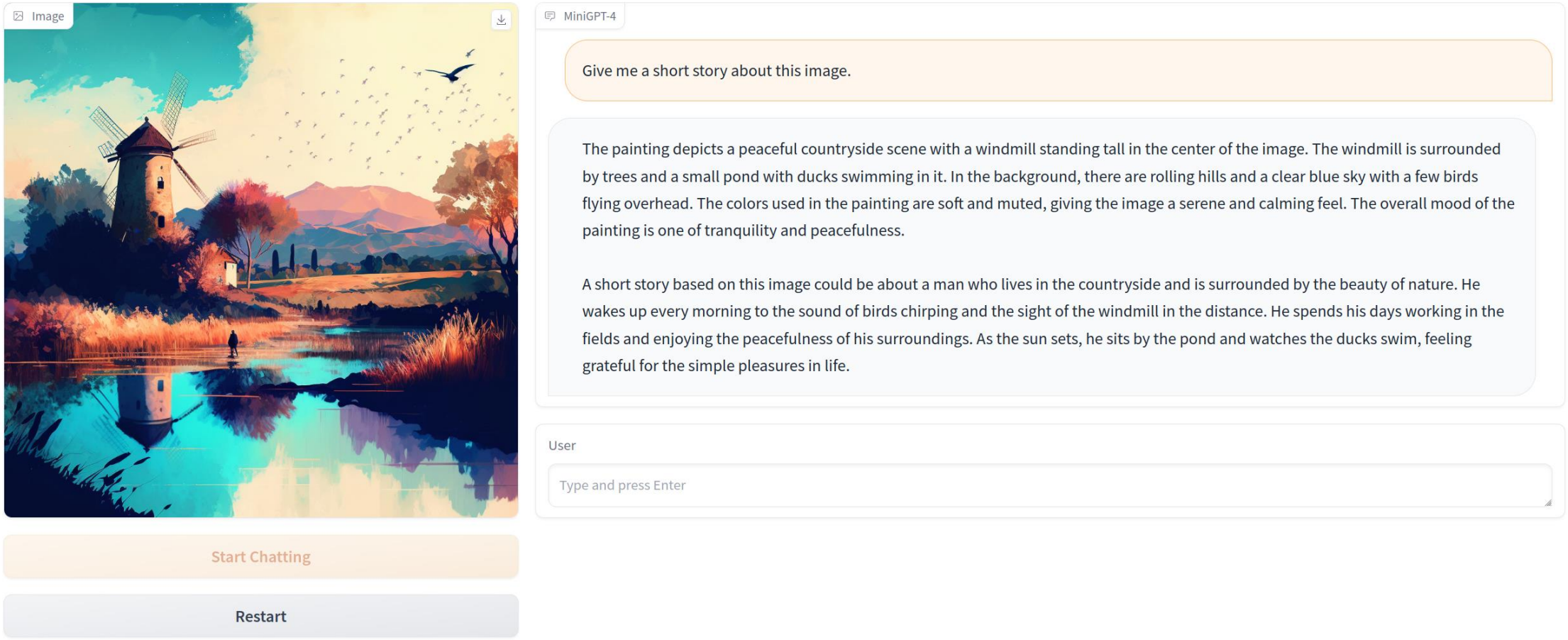
**A Survey of Large Language Models:**
*https://arxiv.org/abs/2303.18223*

# Vision Language Models




The architecture of MiniGPT-4
*https://minigpt-4.github.io*

# Vision Language Models




*https://github.com/Vision-CAIR/MiniGPT-4*

# Thank You For Attending
# Q&A

***Hichem Felouat …***