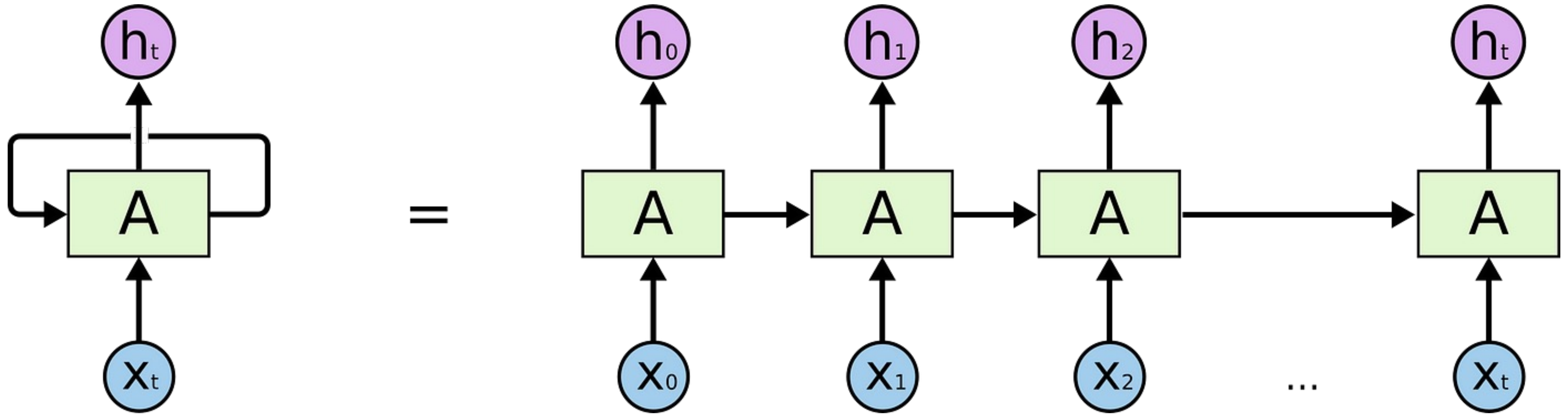
Deep Learning for Sequences

ICS/DATA 435 and ICS 635

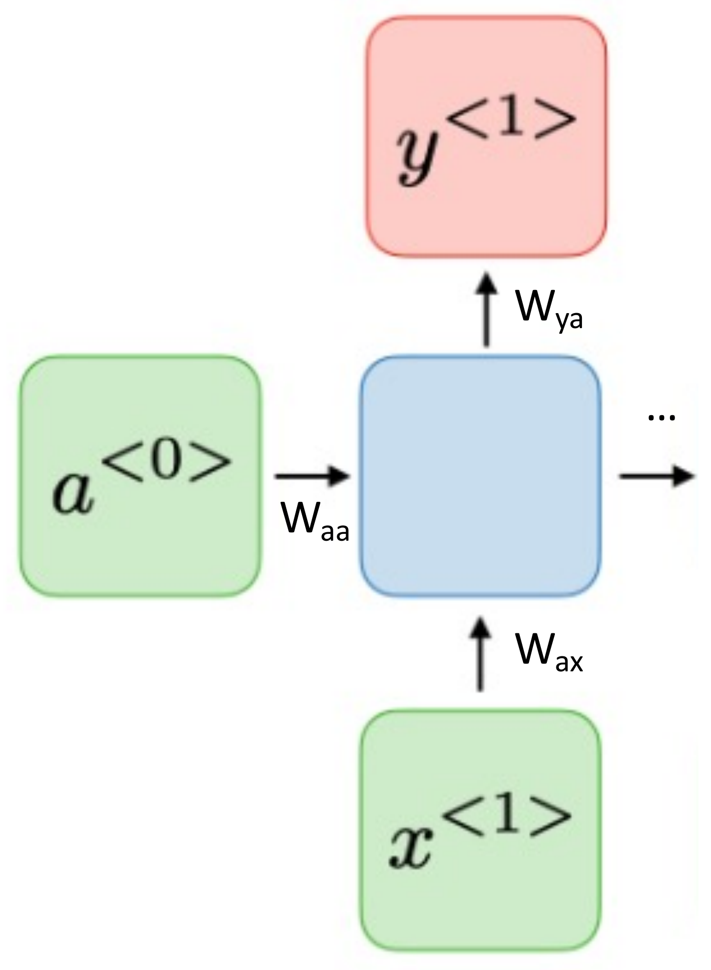
Spring 2023

Recurrent Neural Networks (RNNs)

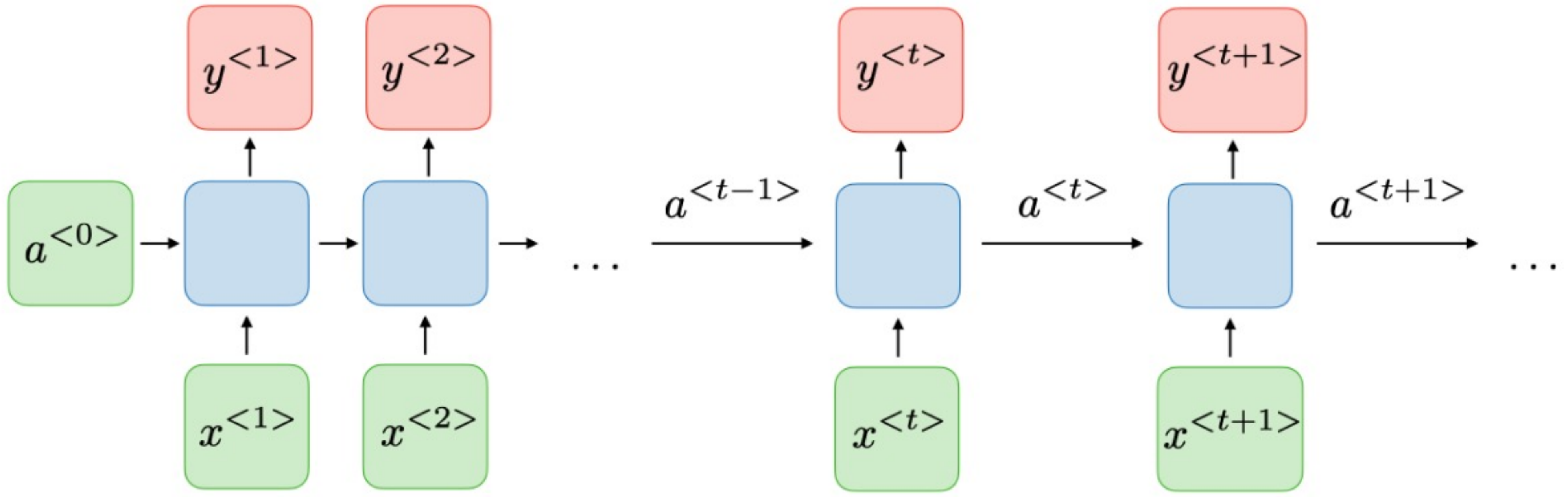
Basic RNN



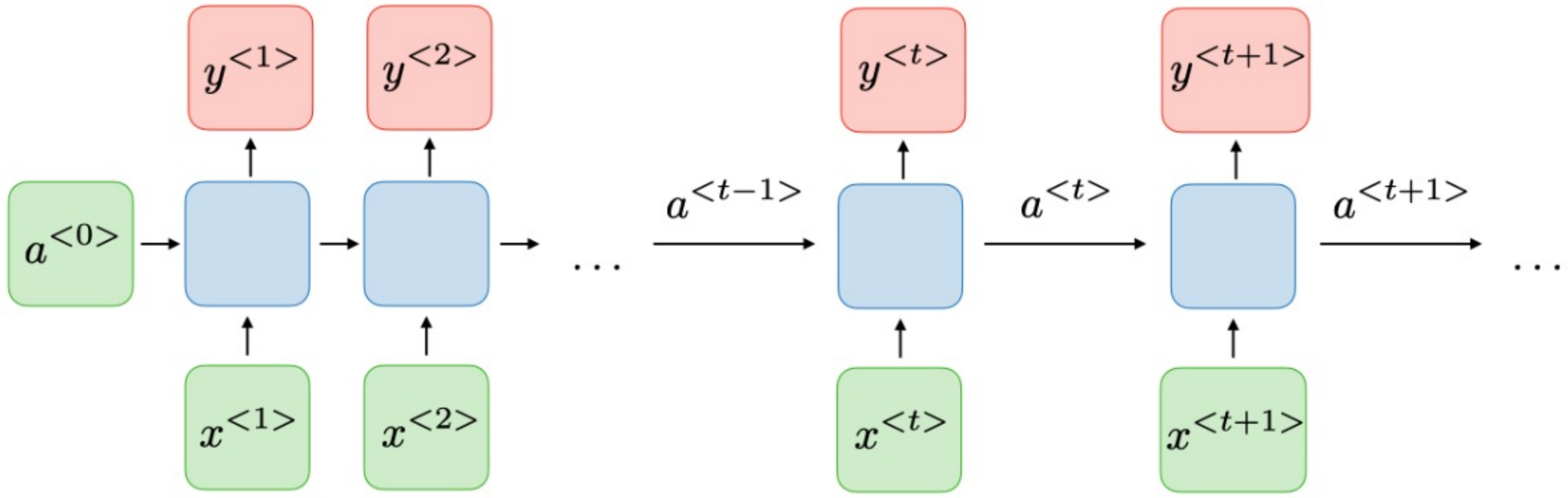
Three Weight Matrices in an RNN:



RNN



RNN



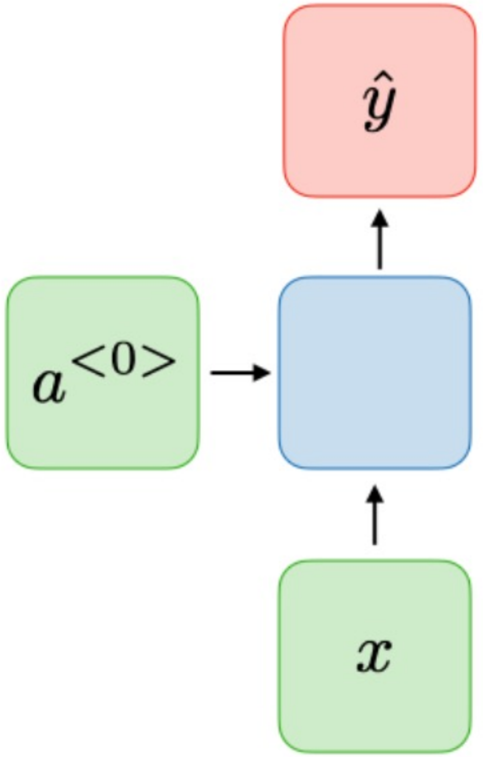
$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

Advantages of RNNs

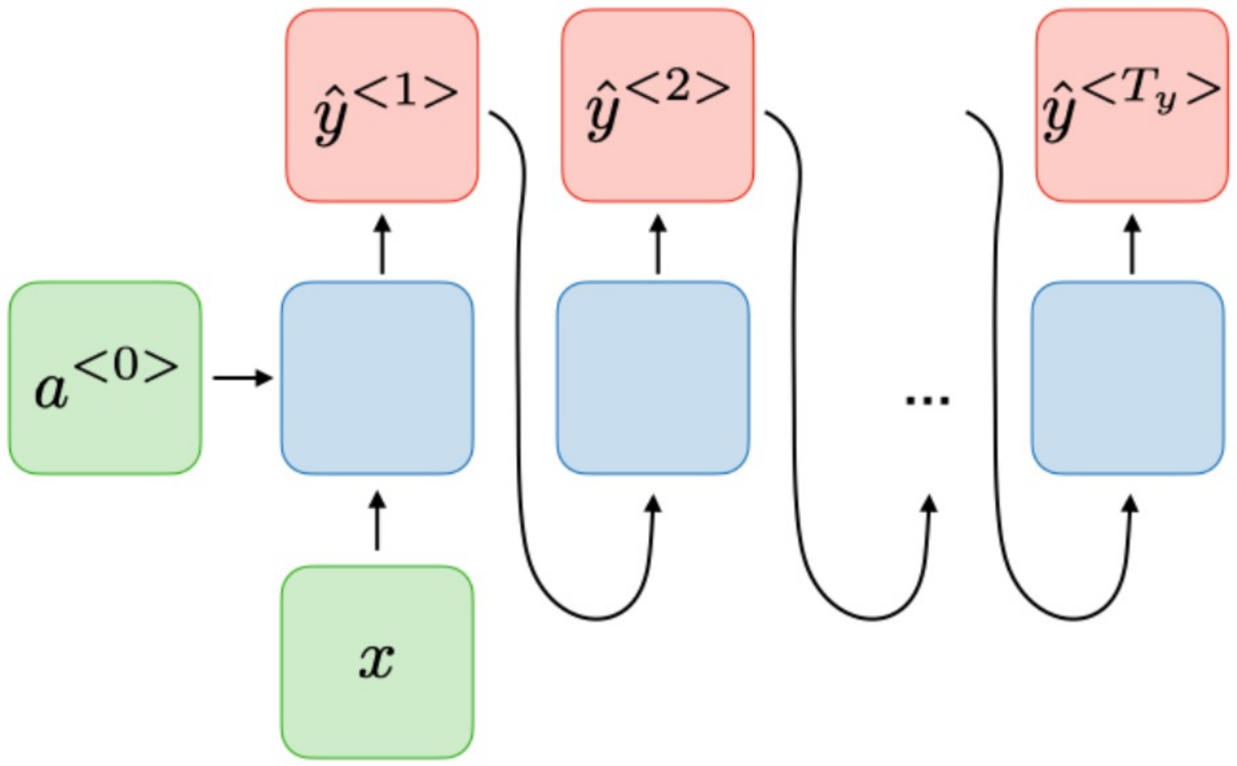
- Can work with any input sequence length
    - Since same weight matrices are used at each time step
- Model size is same regardless of input sequence length
- Computation takes into account historical information

One-to-One Prediction



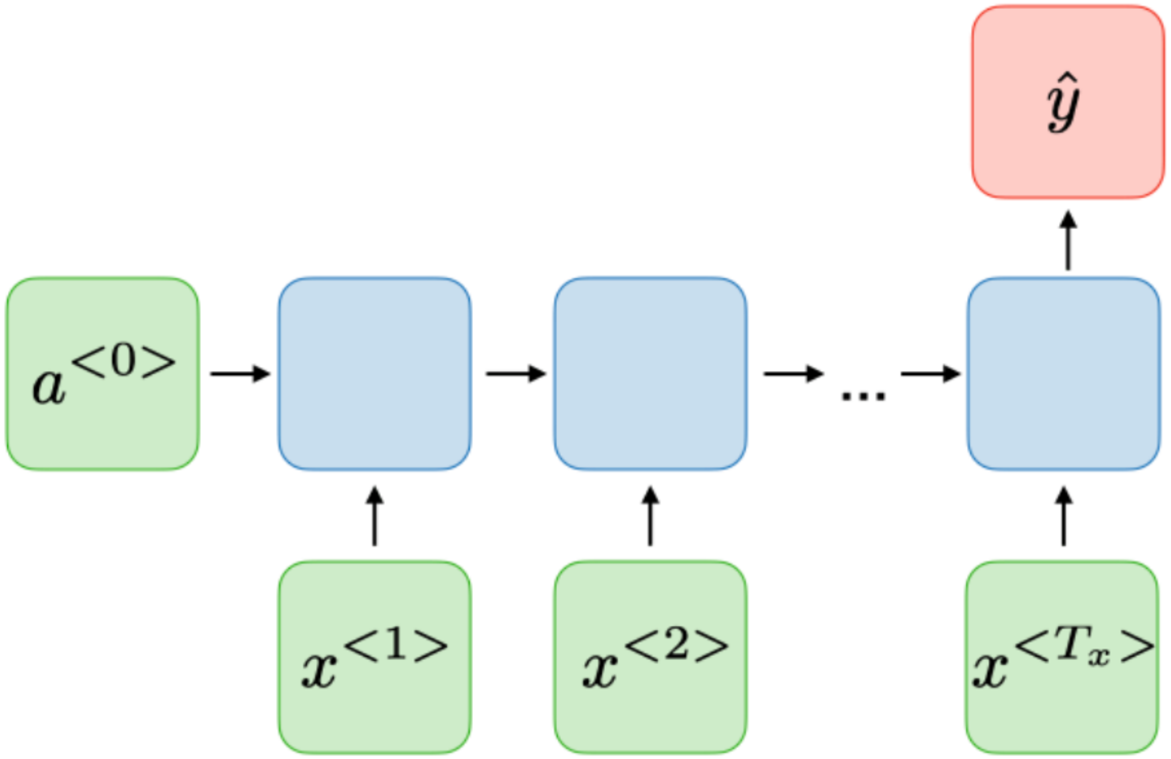
One-to-Many Prediction

(Example: music generation)



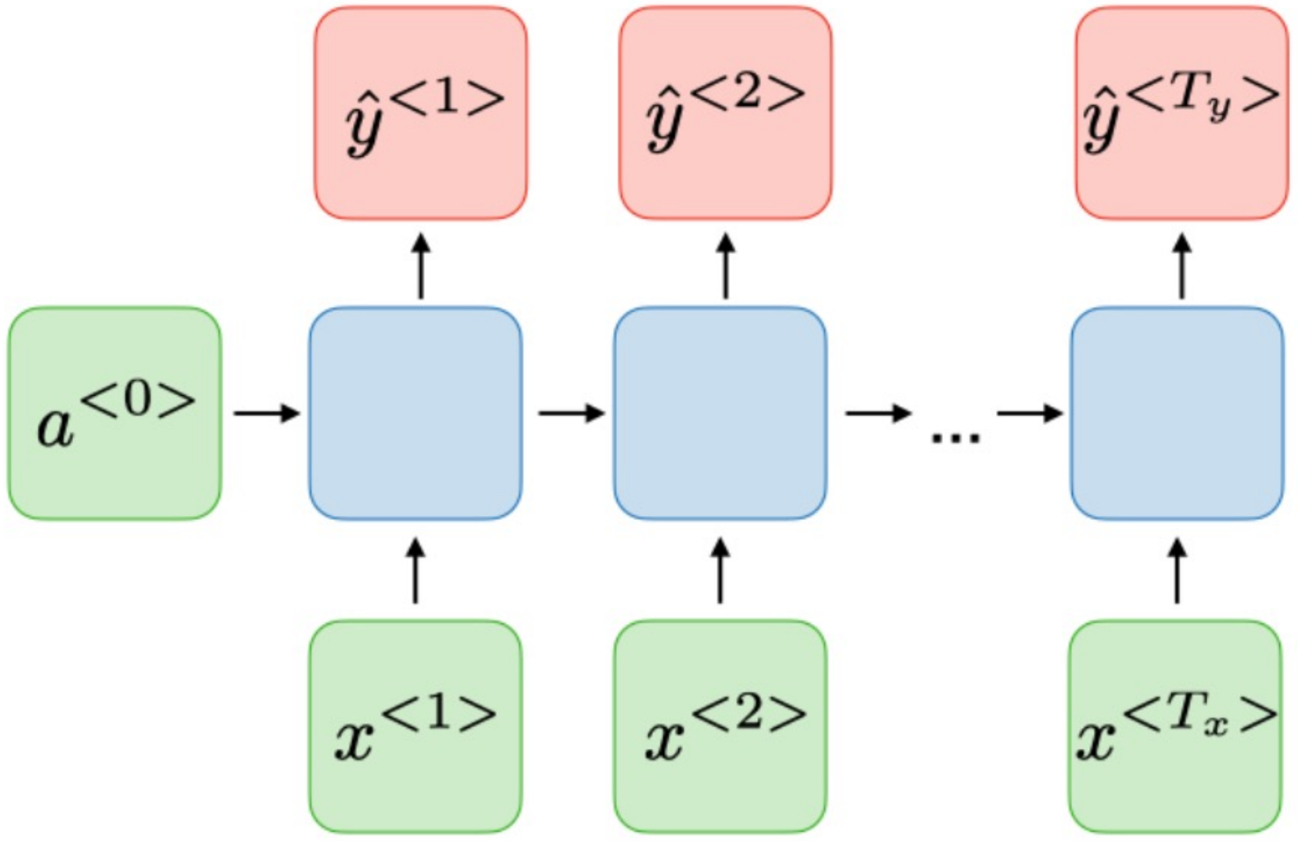
Many-to-One Prediction

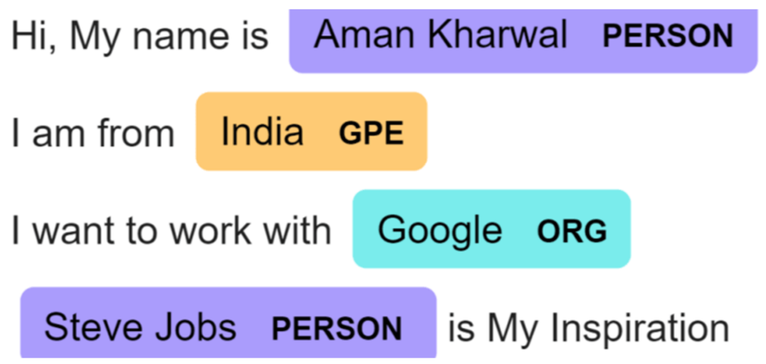
(Example: text sentiment classification)



Many-to-Many Prediction

(Example: named entity recognition)



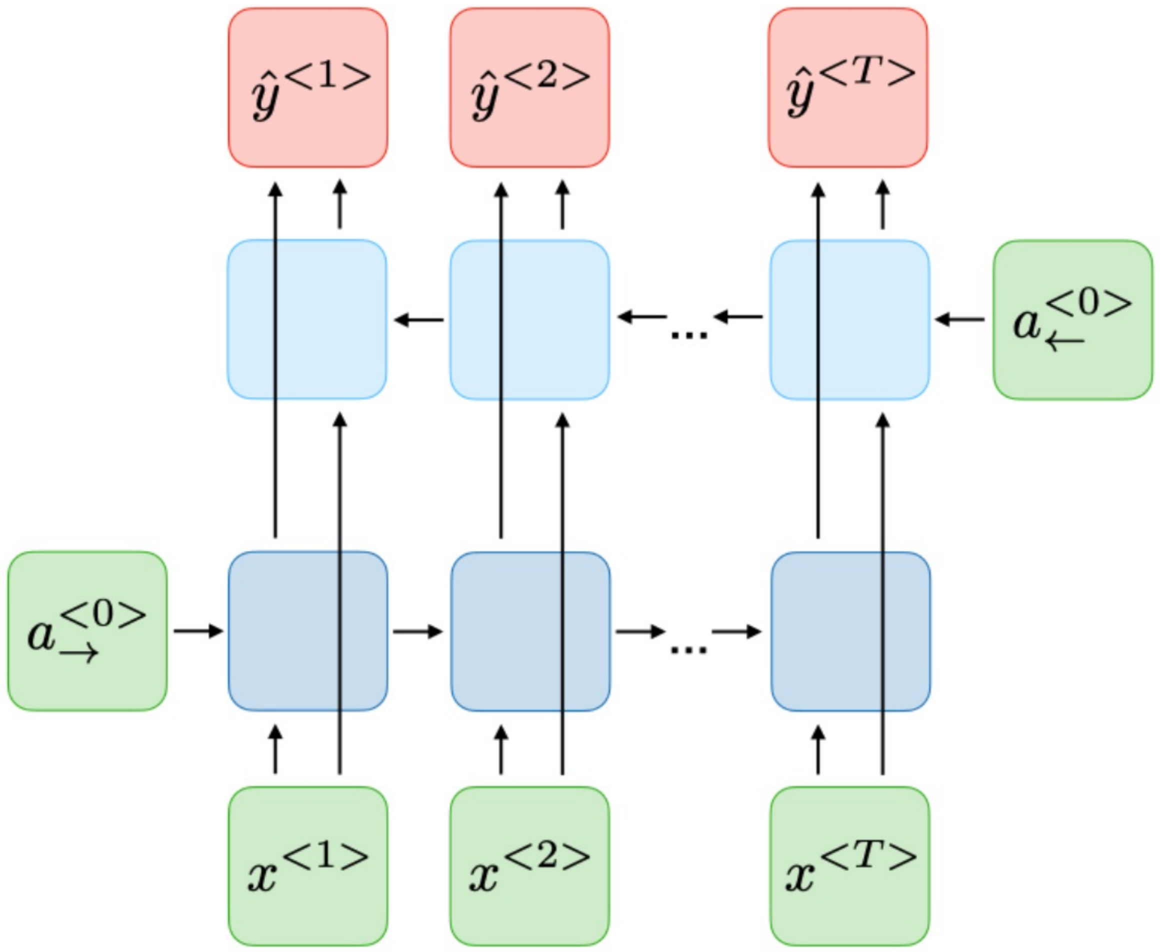

Hi, My name is Aman Kharwal **PERSON**

I am from India **GPE**

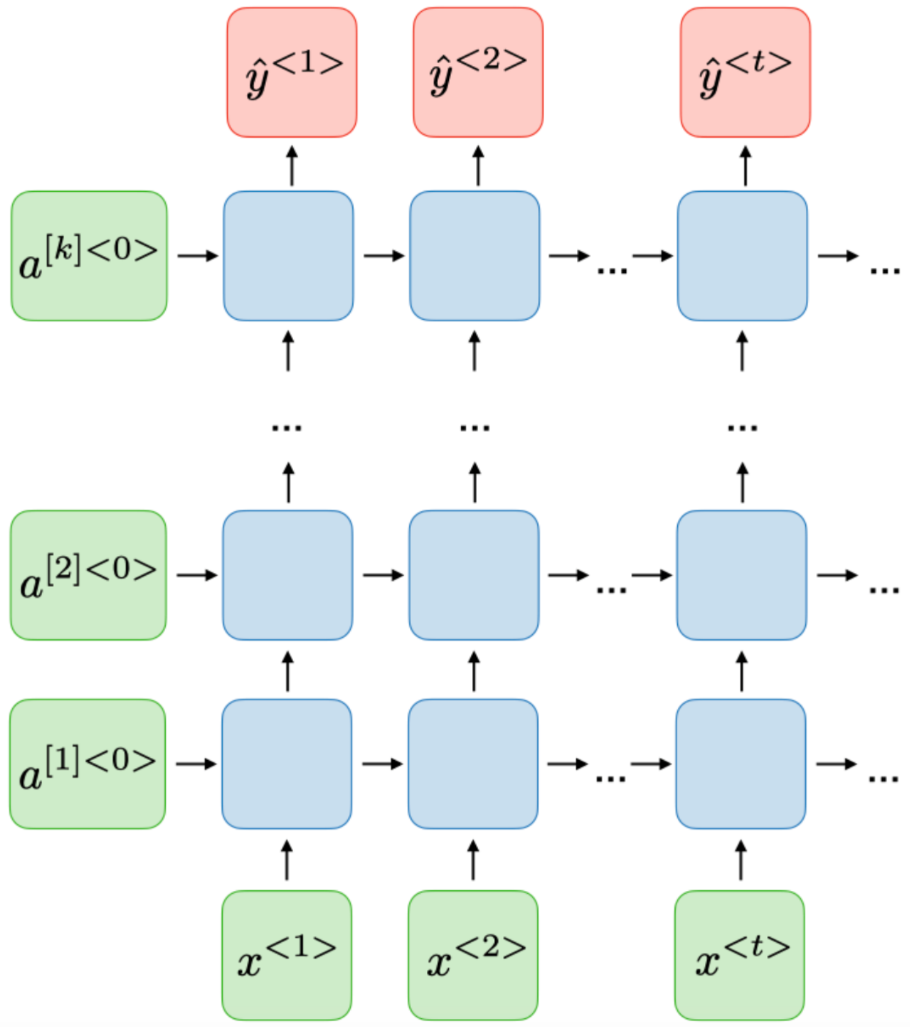
I want to work with Google **ORG**

Steve Jobs **PERSON** is My Inspiration

Bidirectional RNNs



Deep RNN



RNN Loss Function

The loss function of all time steps is defined based on the loss at every time step as follows:

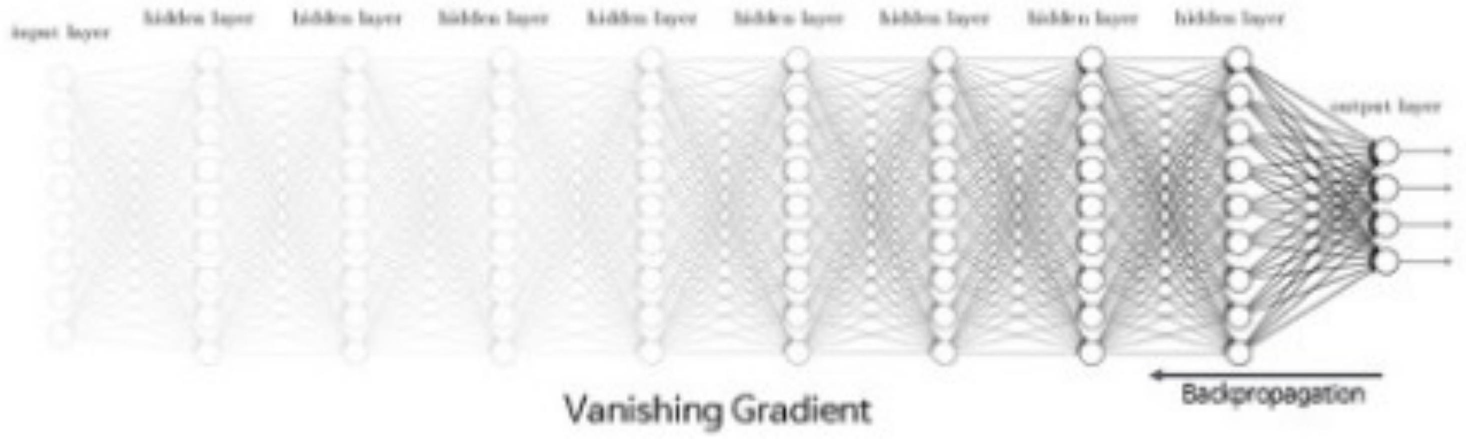
$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{<t>}, y^{<t>})$$

Backprop through Time

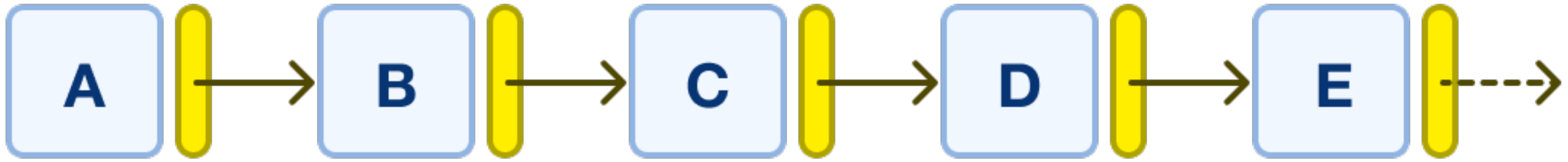
Backpropagation is done at each point in time. At timestep *T*, the derivative of the loss L with respect to weight matrix *W* is expressed as follows:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^{(T)}}{\partial W}$$

The Problem: Vanishing/Exploding Gradients



Remember: gradients of NNs involve many multiplications, with more multiplications per layer



$$\frac{da}{dx} a(b(c(d(e(f(g(x)))))))$$

$$\frac{da}{\partial x} = \frac{da}{db} \times \frac{db}{dc} \times \frac{dc}{dd} \times \frac{dd}{de} \times \frac{de}{df} \times \frac{df}{dg} \times \frac{dg}{dx}$$

Vanishing/Exploding Gradients

Vanishing Gradients:

- Sequence of length 100 has O(100) multiplications
- Let's say each connecting weight is 0.5
- $0.5^{100} = 7.89 \times 10^{-31}$

Exploding Gradients:

- Sequence of length 100 has O(100) multiplications
- Let's say each connecting weight is 2
- $2^{100} = 1.27 \times 10^{30}$

Vanishing/Exploding Gradients

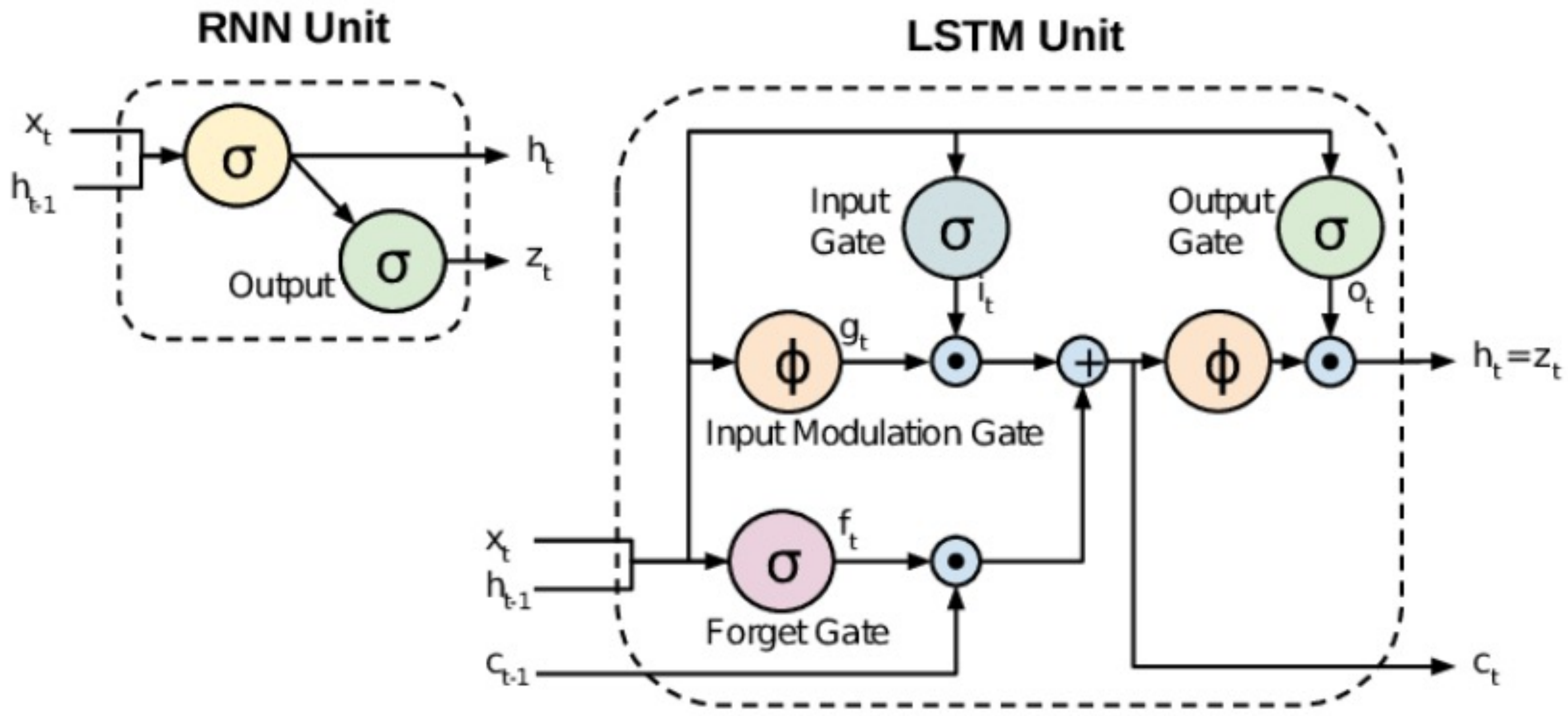
Vanishing Gradients:

- Sequence of length 300 has O(300) multiplications
- Let’s say each connecting weight is 0.5
- $0.5^{300} = 4.91 \times 10^{-91}$

Exploding Gradients:

- Sequence of length 300 has O(300) multiplications
- Let’s say each connecting weight is 2
- $2^{300} = 2.04 \times 10^{90}$

Remedy for the Vanishing Gradient Problem: Long Short-Term Memory (LSTM) Networks

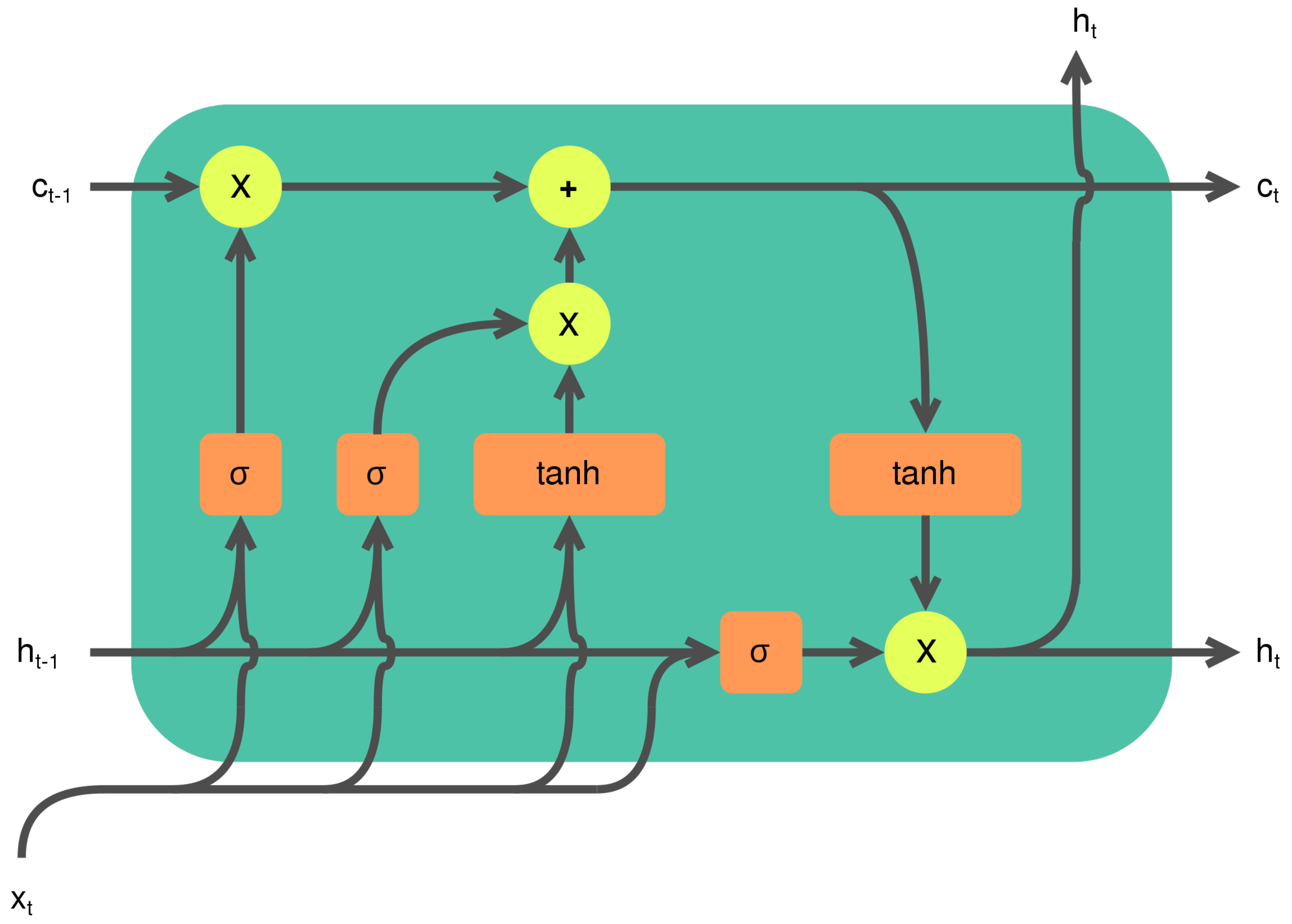


LSTMs

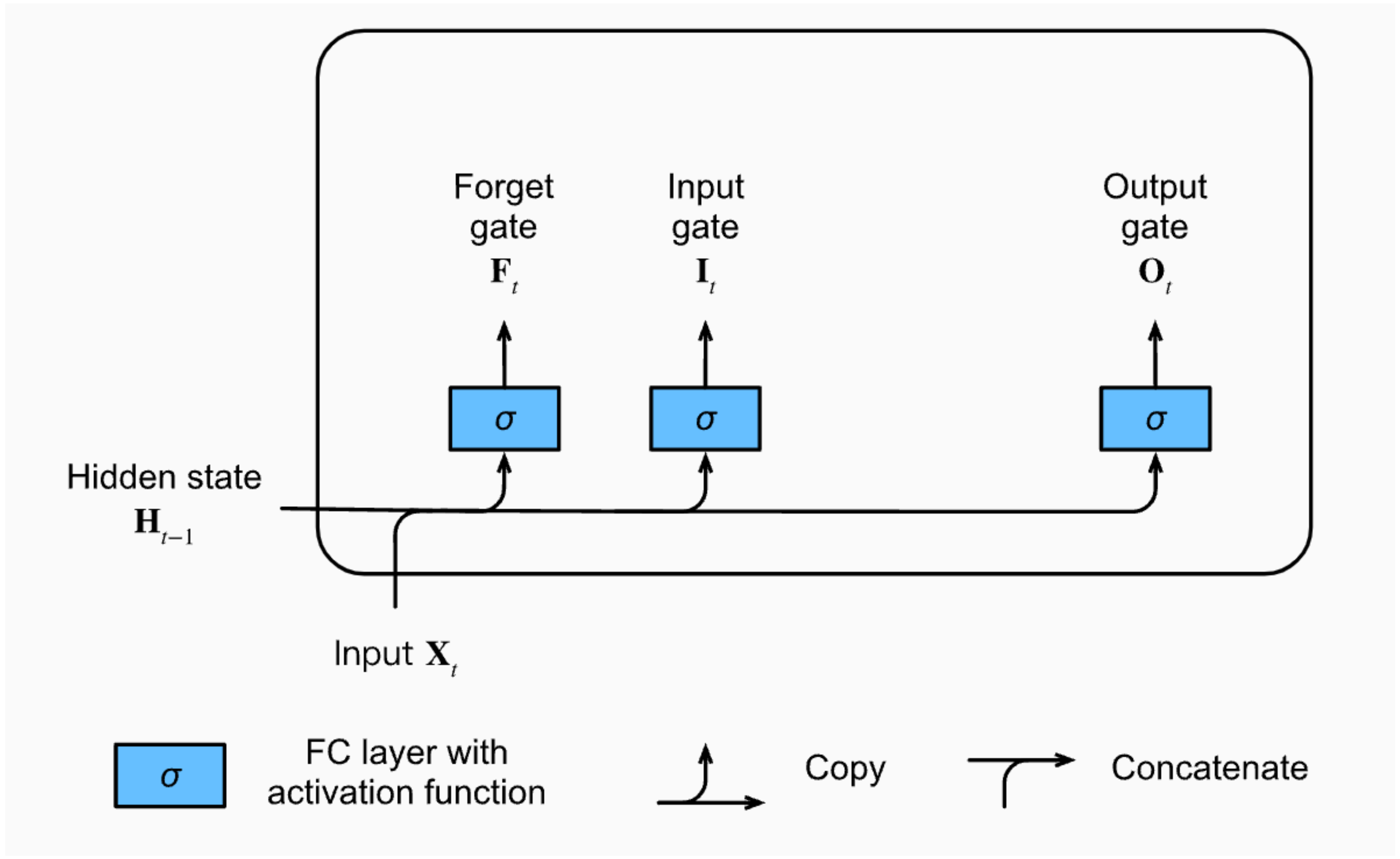
LSTM cells have “gating mechanisms” which allow it to retain the important information from previous time steps in the long-term (and short-term):

- **Input gate**: determines how much of the input node’s value should be added to the current memory cell internal state
- **Forget gate**: determines whether to keep the current value of the memory or flush it
- **Output gate**: determines whether the memory cell should influence the output at the current time step

LSTM Cell

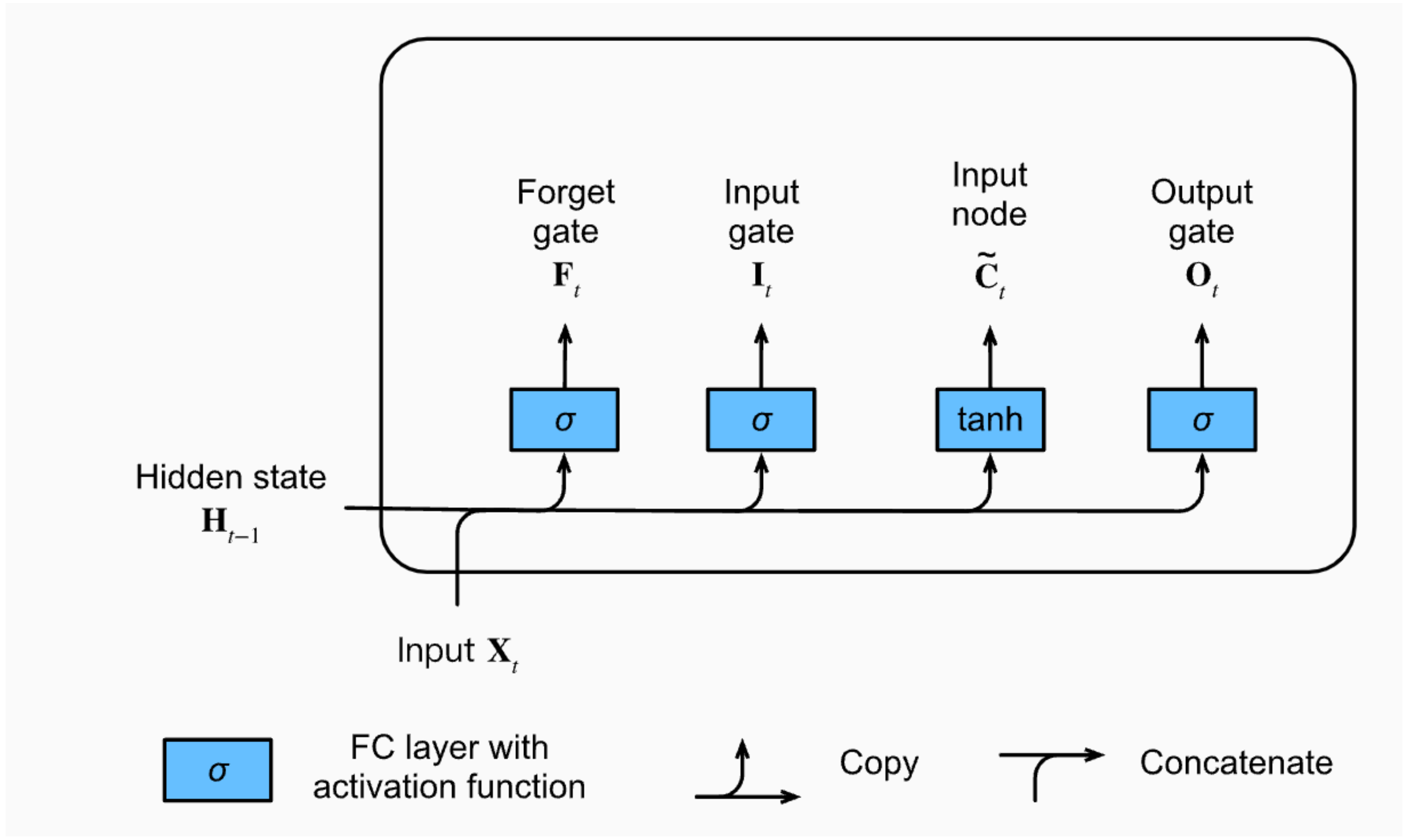


Breaking It Down: Gates


Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Output gate $\mathbf{O}_t$
σ
σ
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation function
Copy
Concatenate

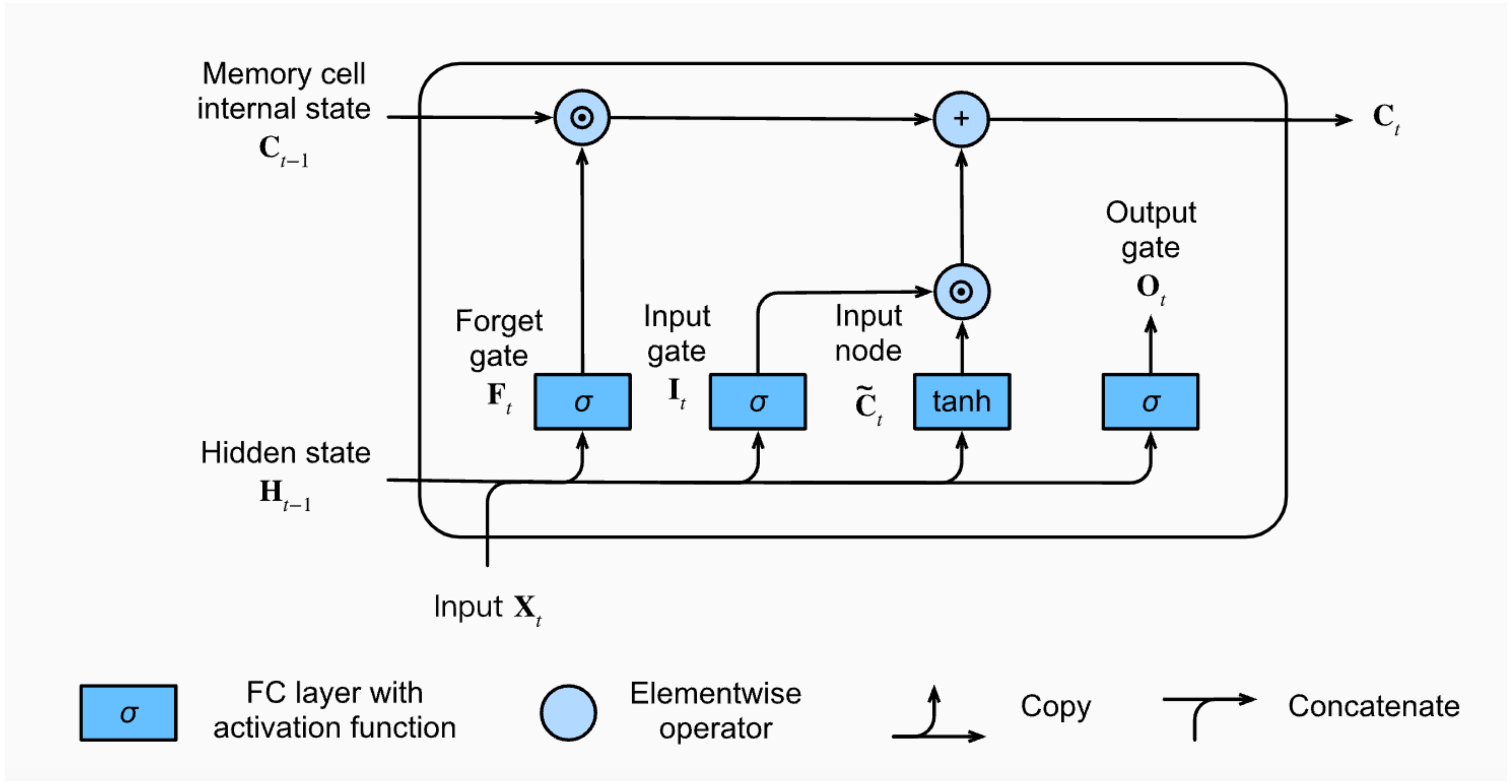
$$\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),
\end{aligned}$$

Breaking it Down: Input Node


Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
σ
tanh
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation function
Copy
Concatenate

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$$

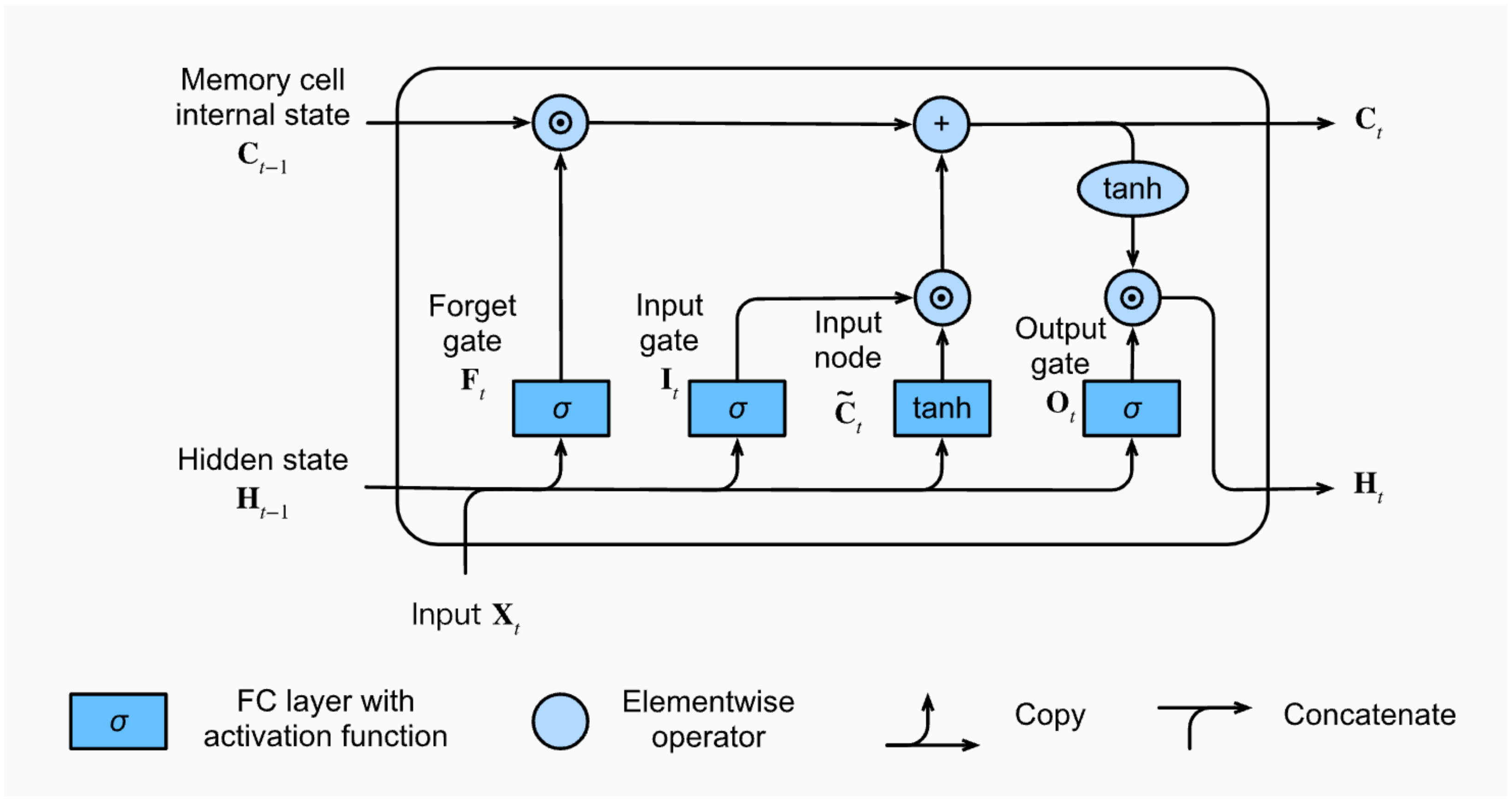
Breaking it Down: Memory Cell Internal State


Memory cell internal state $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
Output gate $\mathbf{O}_t$
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
σ
tanh
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

The input gate controls how much we take new data into account and the forget gate controls how much of the old cell internal state we retain.

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

Breaking it Down: Hidden State



When the output gate is close to 1, we allow the memory cell internal state to impact the subsequent layers uninhibited, whereas for output gate values close to 0, we prevent the current memory from impacting other layers of the network at the current time step.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

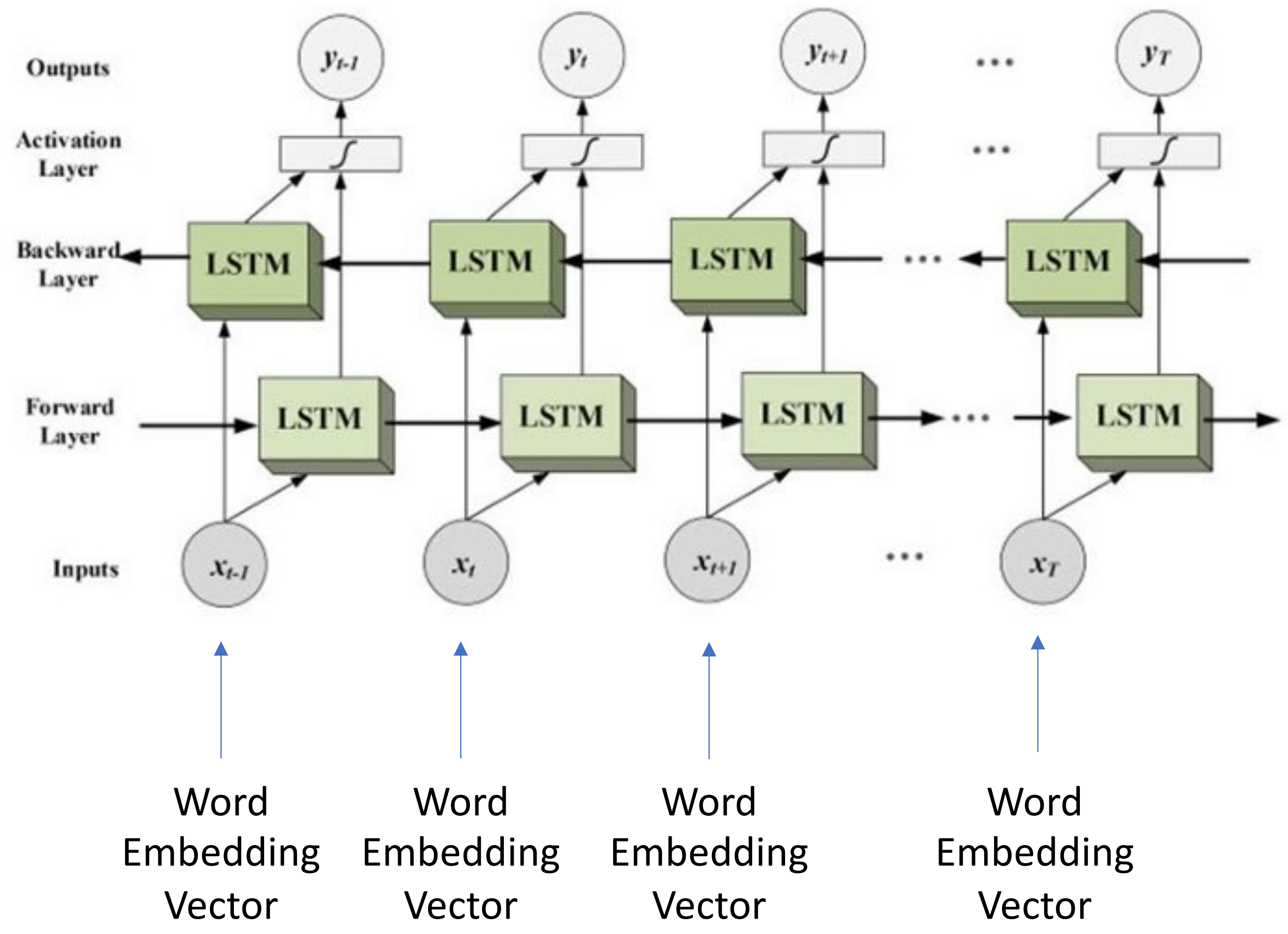
LSTMs

LSTM cells have “gating mechanisms” which allow it to retain the important information from previous time steps in the long-term (and short-term):

- **Input gate**: determines how much of the input node’s value should be added to the current memory cell internal state
- **Forget gate**: determines whether to keep the current value of the memory or flush it
- **Output gate**: determines whether the memory cell should influence the output at the current time step

State-of-the-art for NLP until 2017:
Word Embeddings + Recurrent Neural Networks

State-of-the-art Pre-2017



LSTM milestones:

- Google starts using LSTMs for speech recognition on Google Voice (2015)
- Amazon generates the voices behind Alexa using bidirectional LSTMs for text-to-speech (2016)
- Facebook performed some 4.5 billion automatic translations every day using LSTMs (2017)

State-of-the-art for NLP since 2017: Transformer Models

Transformer Models

Looks complex, but not complex when you break it down.

We have already learned about many of these components.

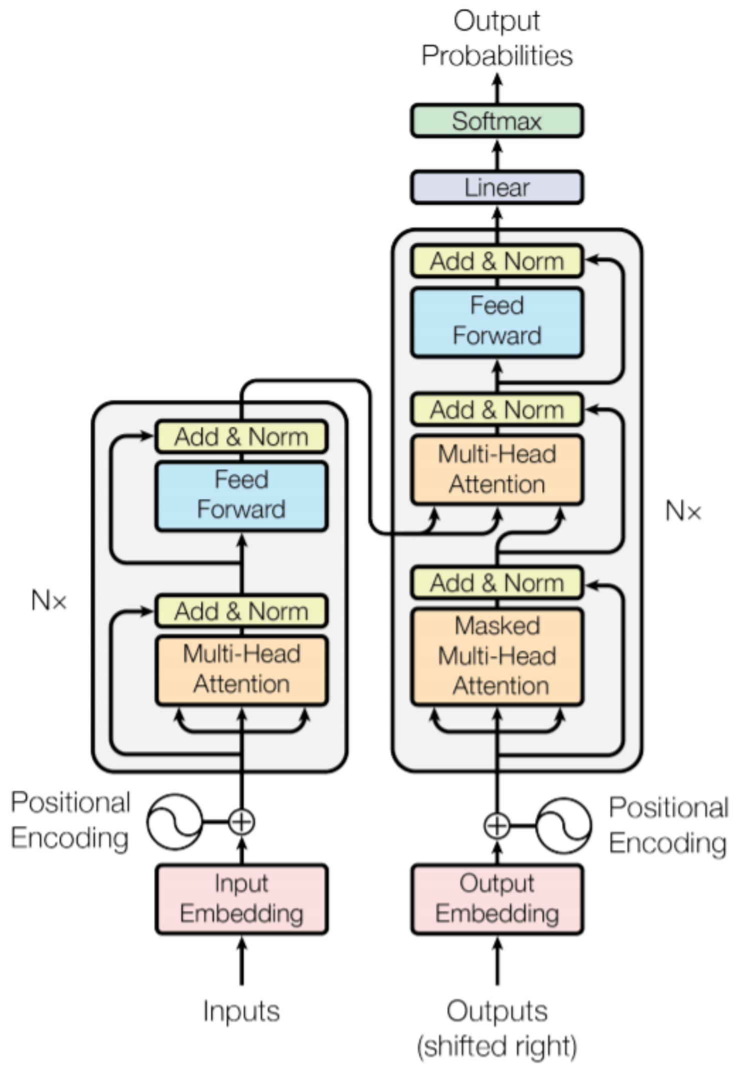


Figure 1: The Transformer - model architecture.

Transformer Models

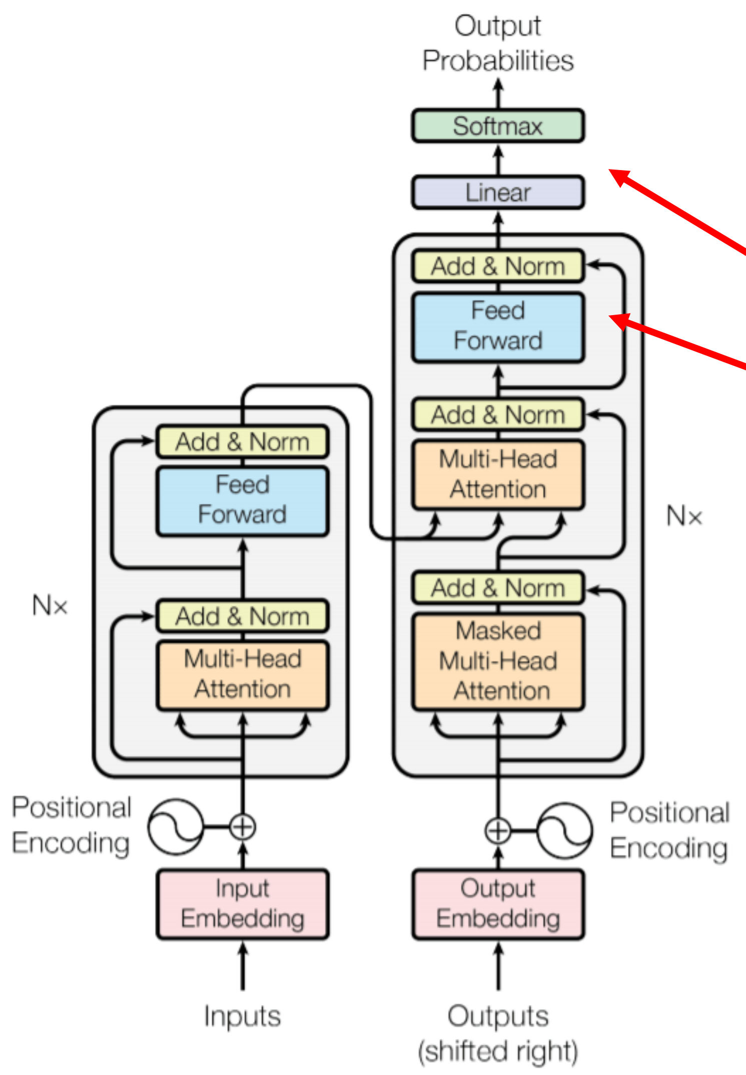
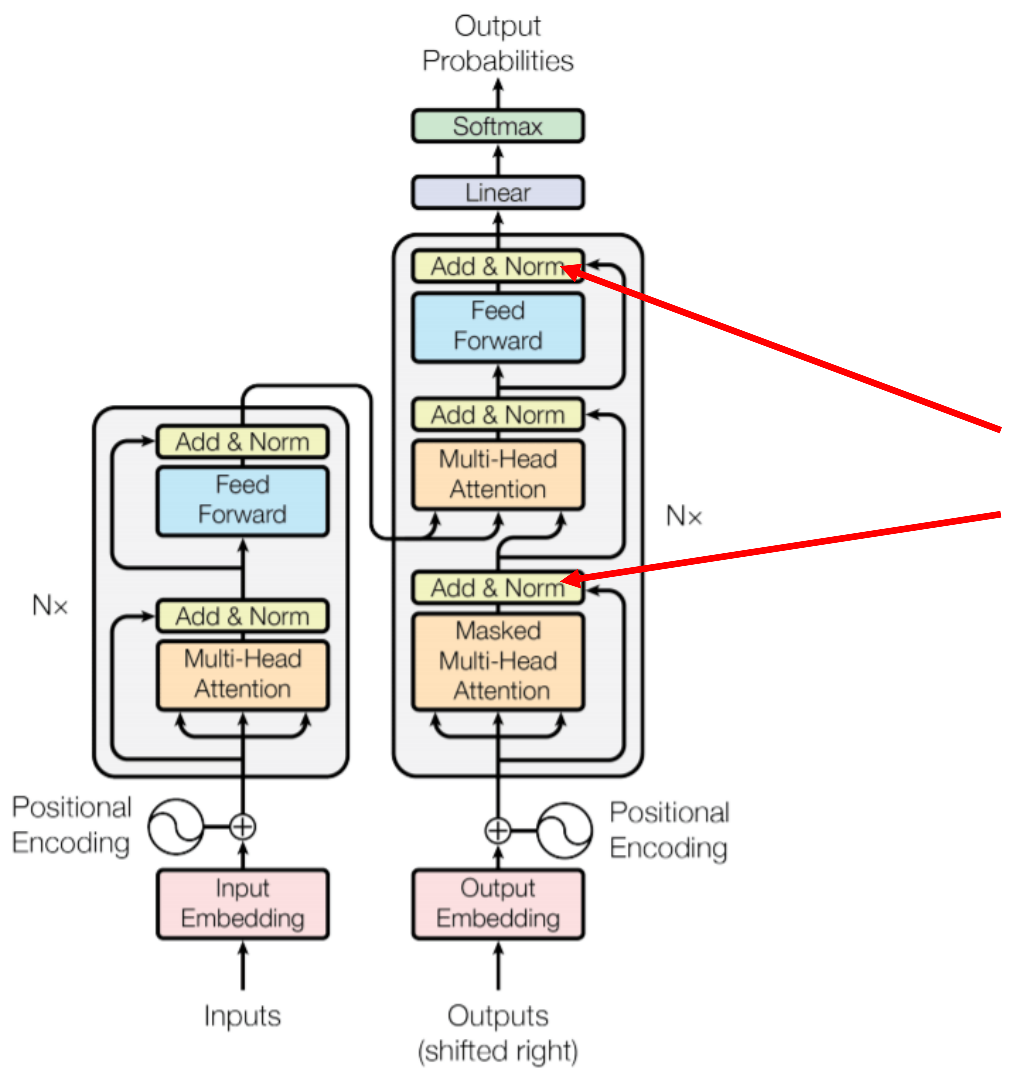


Figure 1: The Transformer - model architecture.

We already know about these types of layers.

Transformer Models



These operations are not central to Transformers but make neural networks easier to train in general.

Figure 1: The Transformer - model architecture.

Transformer Models

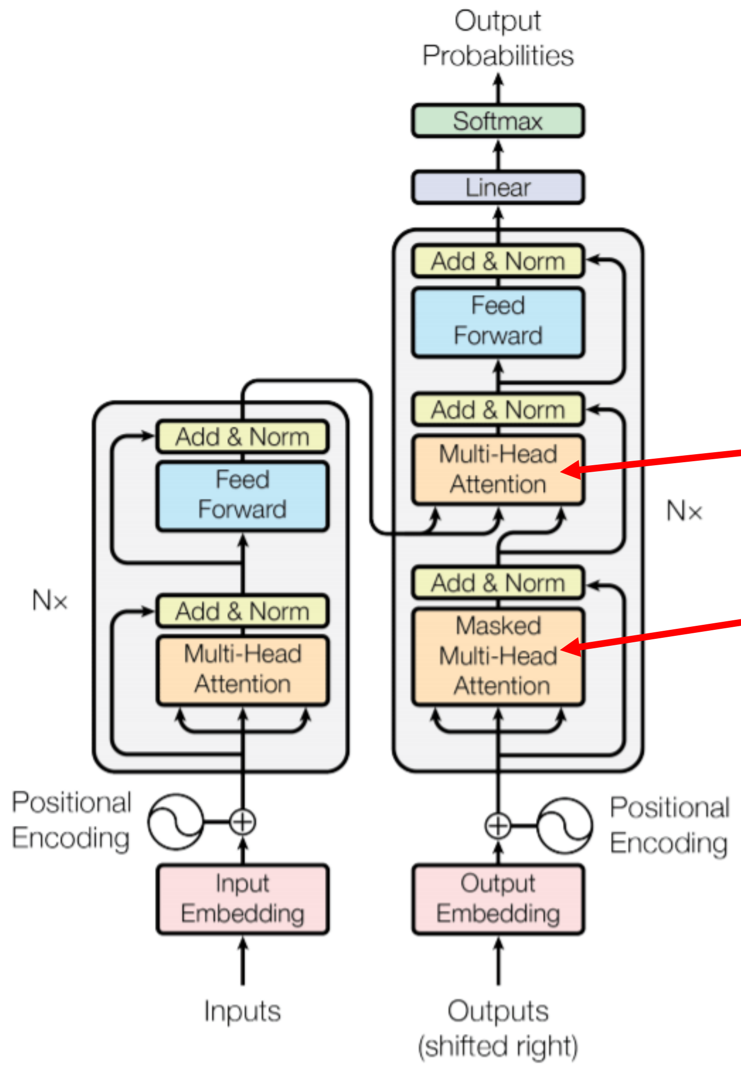
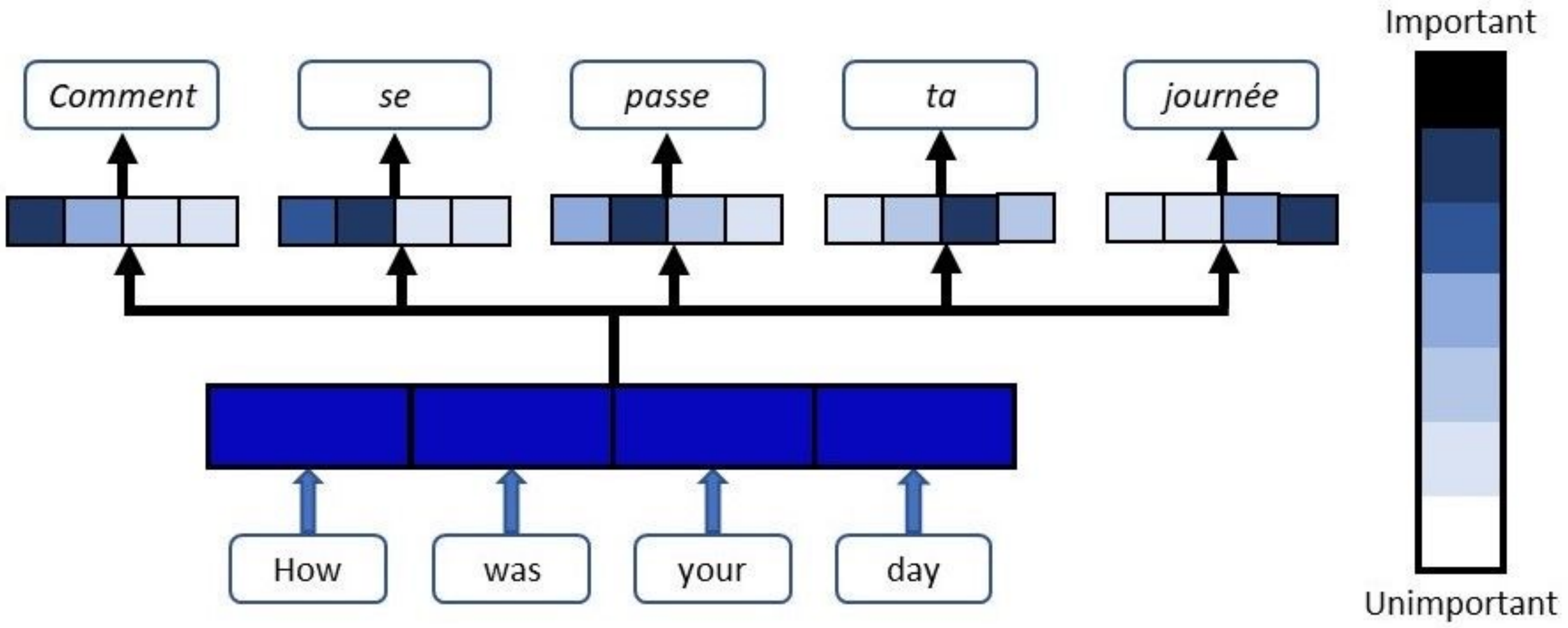


Figure 1: The Transformer - model architecture.

These new layers are central to what makes Transformers the state of the art today. Let's start by understanding Attention, then Multi-Head Attention, then Masked Multi-Head Attention.

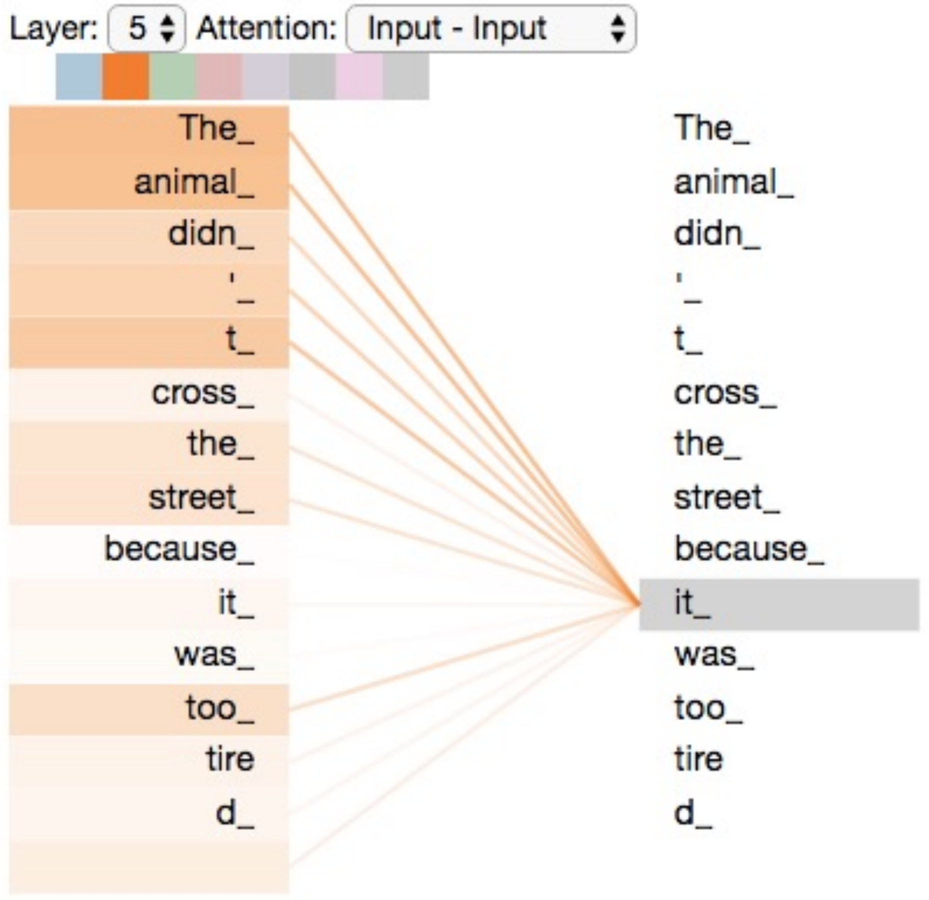
"Attention Is All You Need" (2017)

How much should each item in a sequence weight / prioritize / attend to elements of another sequence?

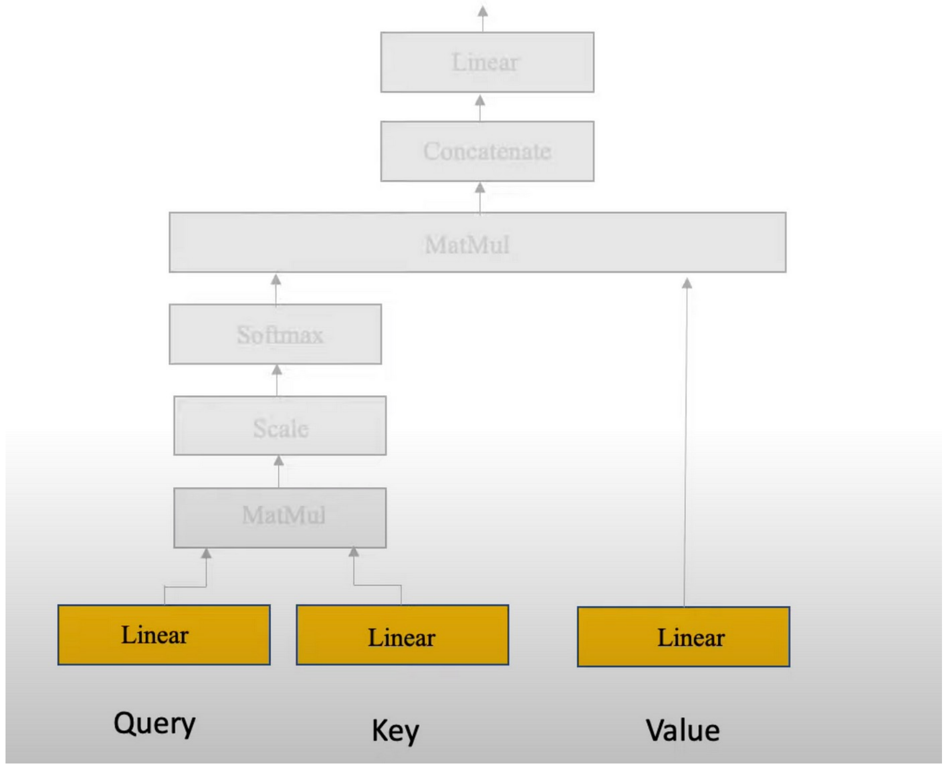


Self-Attention

How much should each item in a sequence weight / prioritize / attend to elements of itself?



Implementing Self-Attention



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

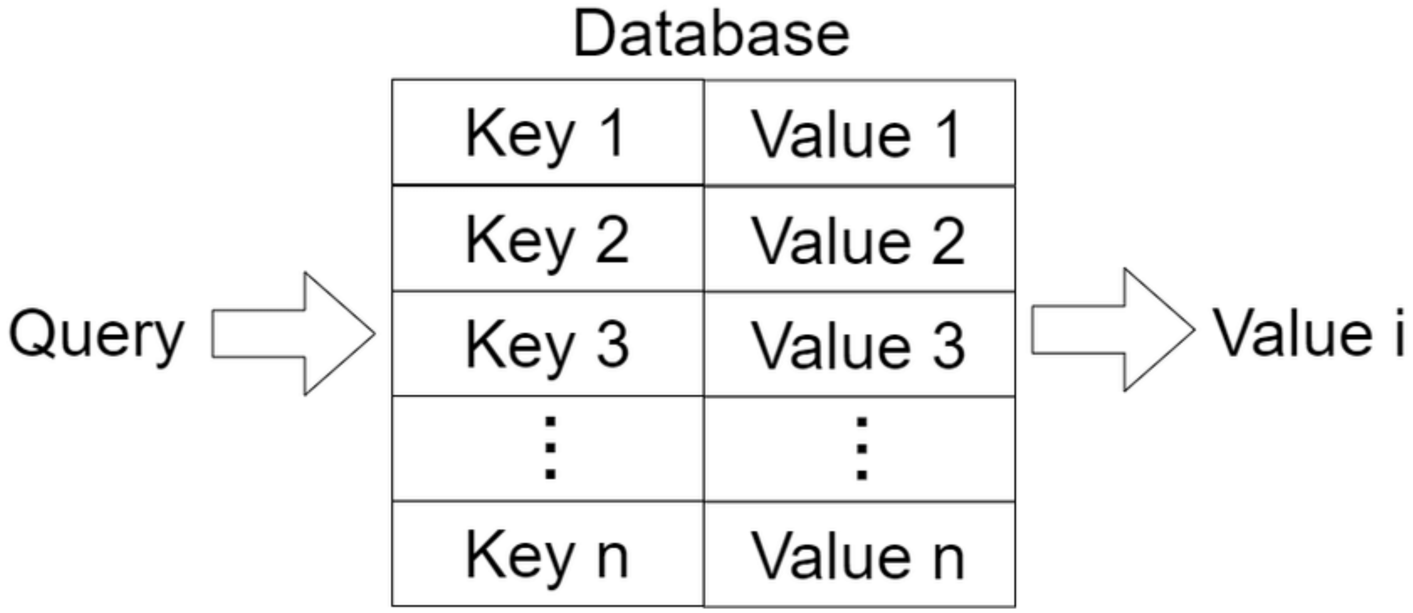
Q, K, and V

Queries are representations of the word in question

Keys are representations of every other word that the Query word is compared against

Values are different representations of the Keys / words that the Query word is compared against

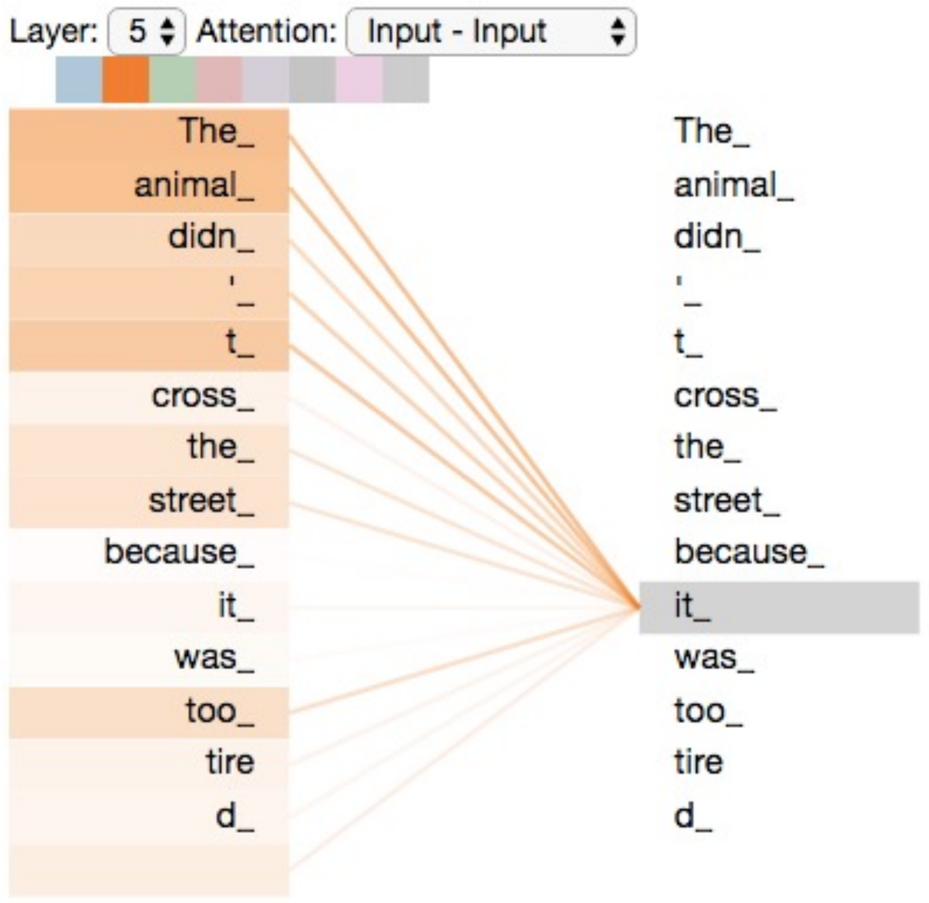
Database Analogy:



Self-Attention

How much should each item in a sequence weight / prioritize / attend to elements of itself?

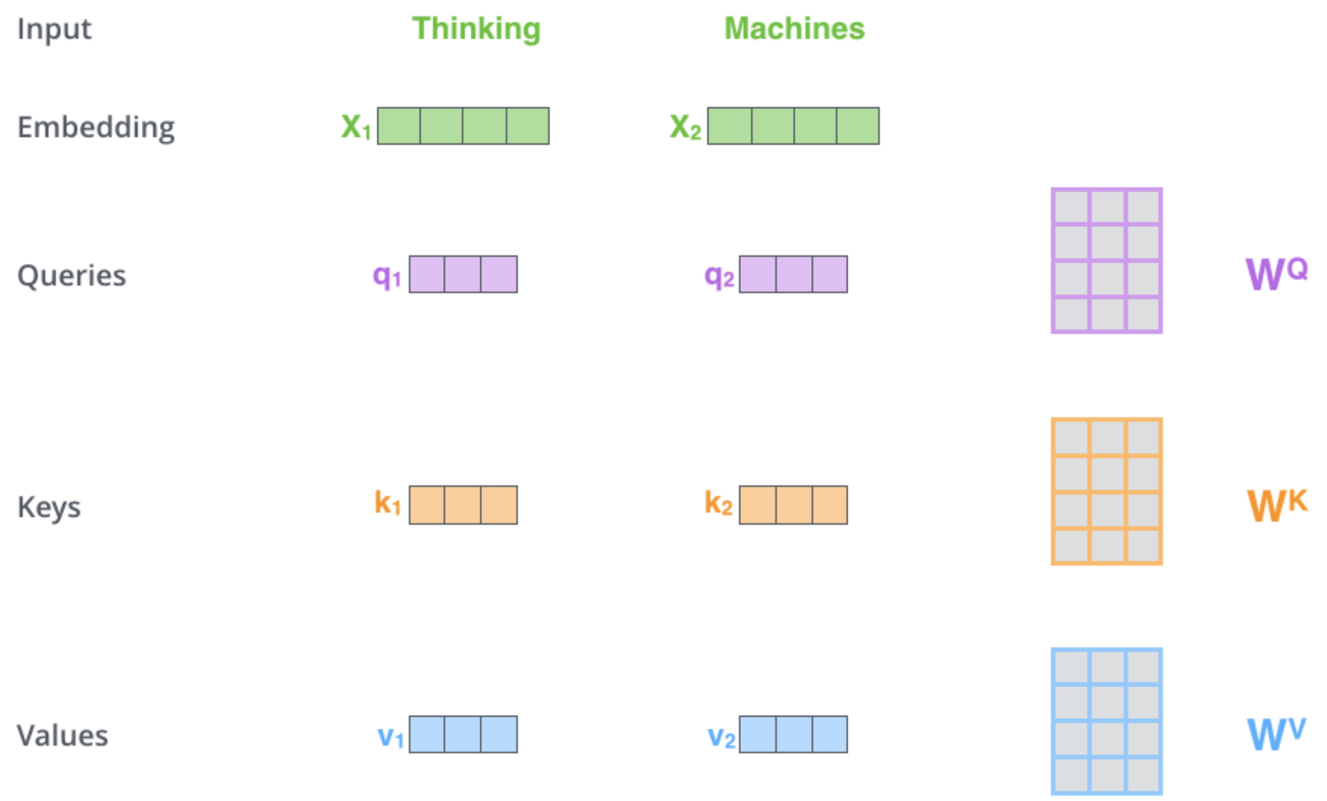
Keys



Query

Self-Attention Deconstructed

The Query (Q), Key (K), and Value (V) are representations of the word which are learned by the Transformer network. Multiply corresponding weight matrix by input to get Q / K / V vectors.



Self-Attention Deconstructed

Recall that we can often use dot product to measure similarity.

We want the similarity between the key of each word and the value of every other word.

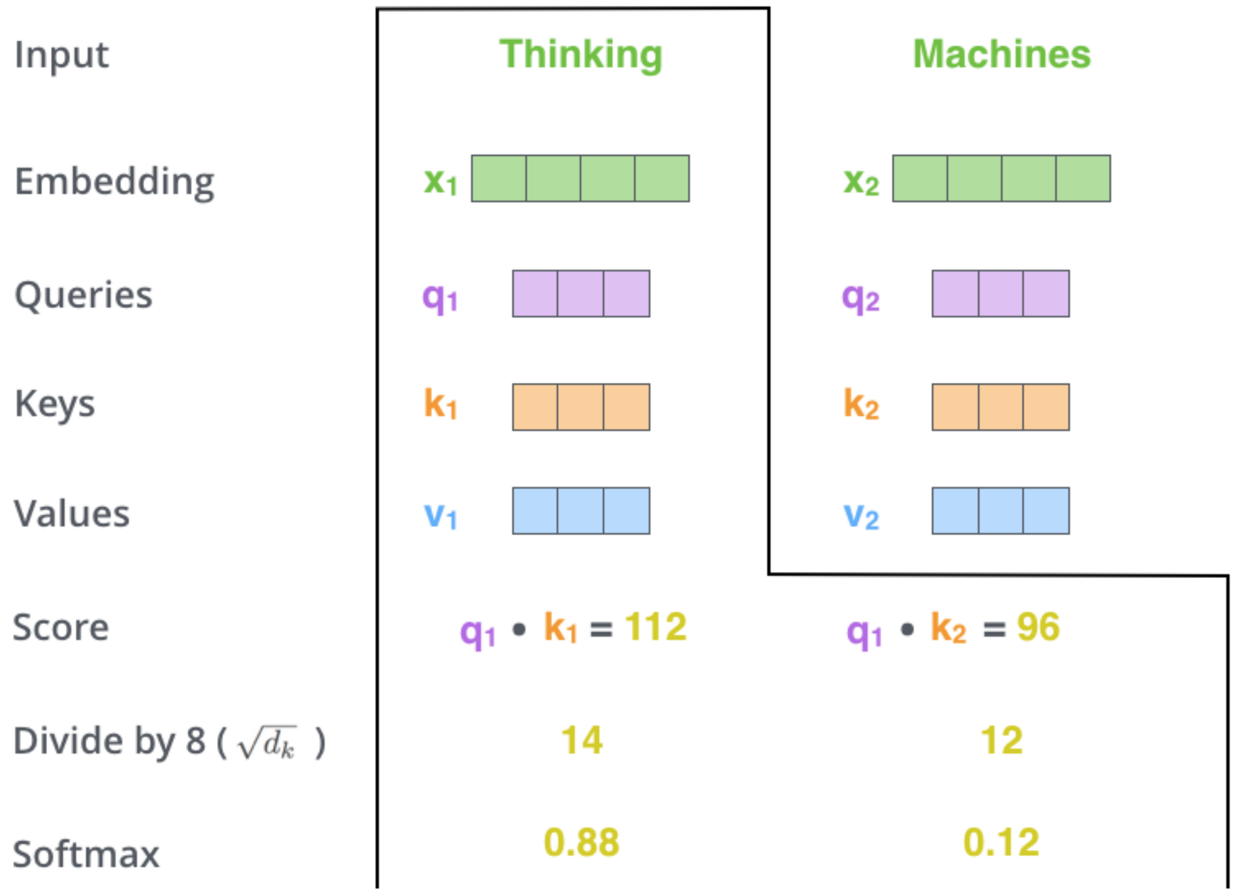
$$Similarity(A, B) = \frac{A \, . \, B}{scaling}$$

$$Similarity(A, B) = \frac{A \, . \, B^T}{scaling}$$

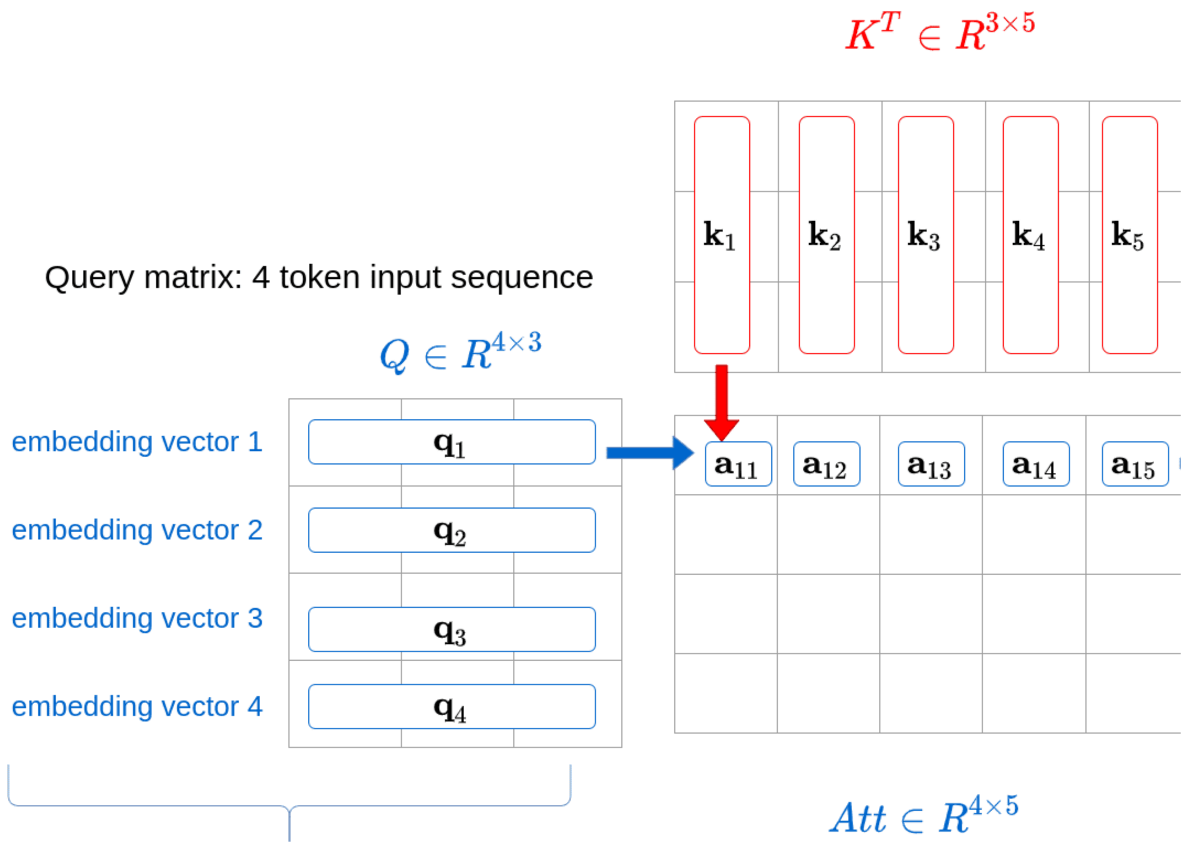
$$Similarity(Q, K) = \frac{Q \, . \, K^T}{scaling}$$

Self-Attention Deconstructed

As usual, we can apply the Softmax function to get probabilities:



Softmax(Q.K^T) gives us a similarity matrix between each word and every other word



Embeddings are processed in parallel
with a simple matrix multiplication

$Att \in R^{4\times5}$

Attention matrix
each dot product show the similarity
between a query and key vector
(softmax is applied row-wise)

Self-Attention Deconstructed

So far, we already have most of Self-Attention deconstructed:

$$\text{Attention}(Q, K, V) = \text{softmax}(\underbrace{\frac{QK^T}{\sqrt{d_k}}}_{\text{This part is covered so far.}})V$$

Self-Attention Deconstructed

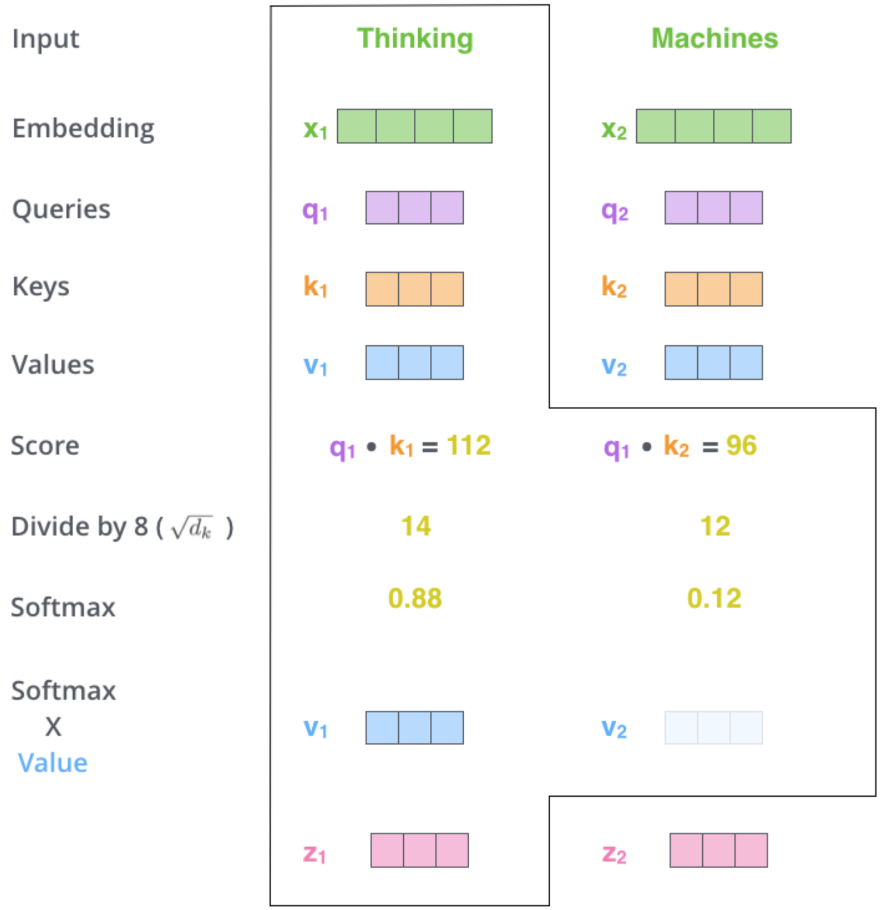
So far, we already have most of Self-Attention deconstructed:

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

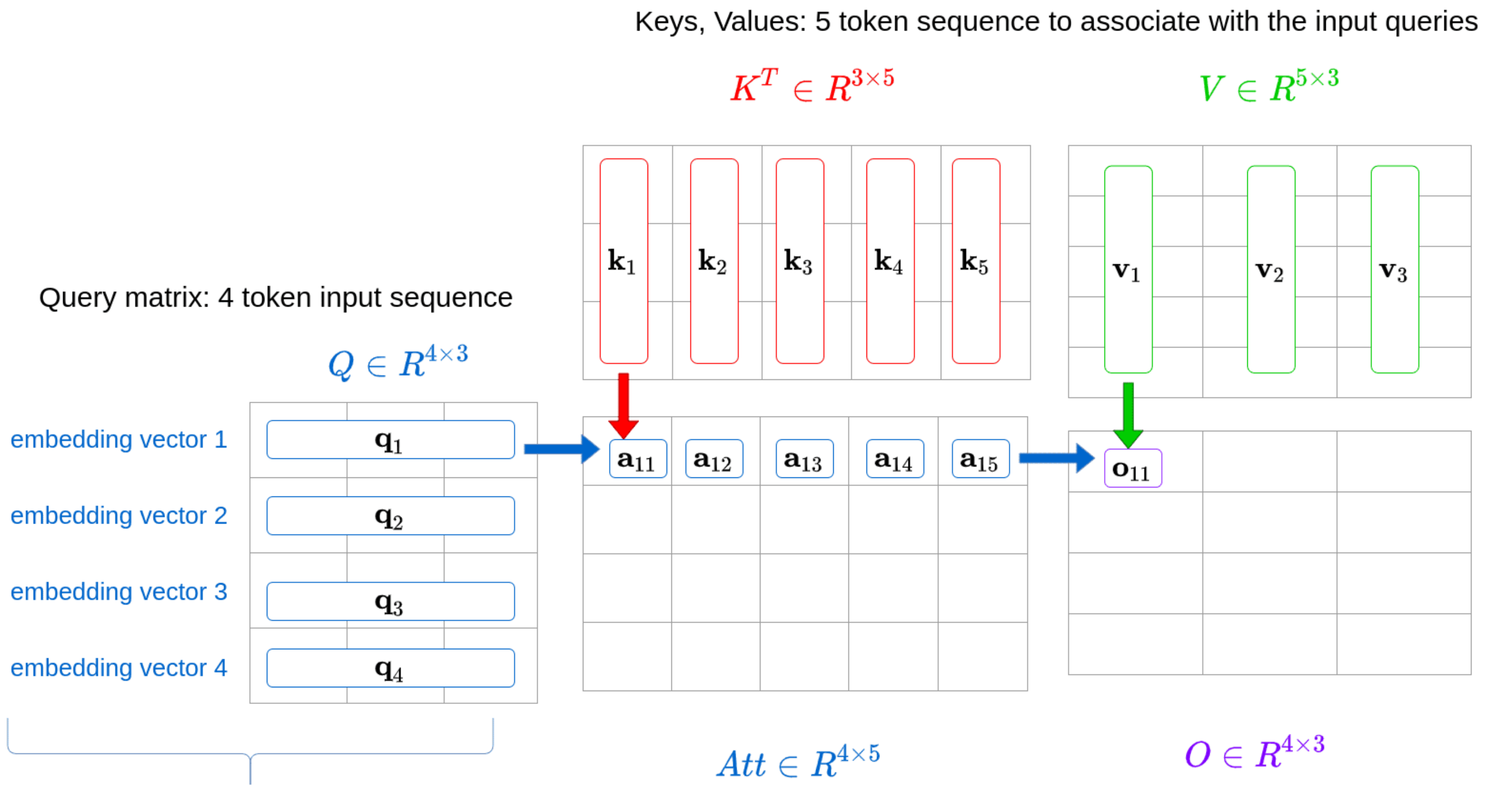
What about V?

Self-Attention Deconstructed

Since we ended up with a softmax probability score for each position of the sequence, we can multiply it by a representation of the input word/element to get a vector which will be passed to the next layer:



Multiply attention weights Softmax(Q.K^T) by V to get the output weighted values

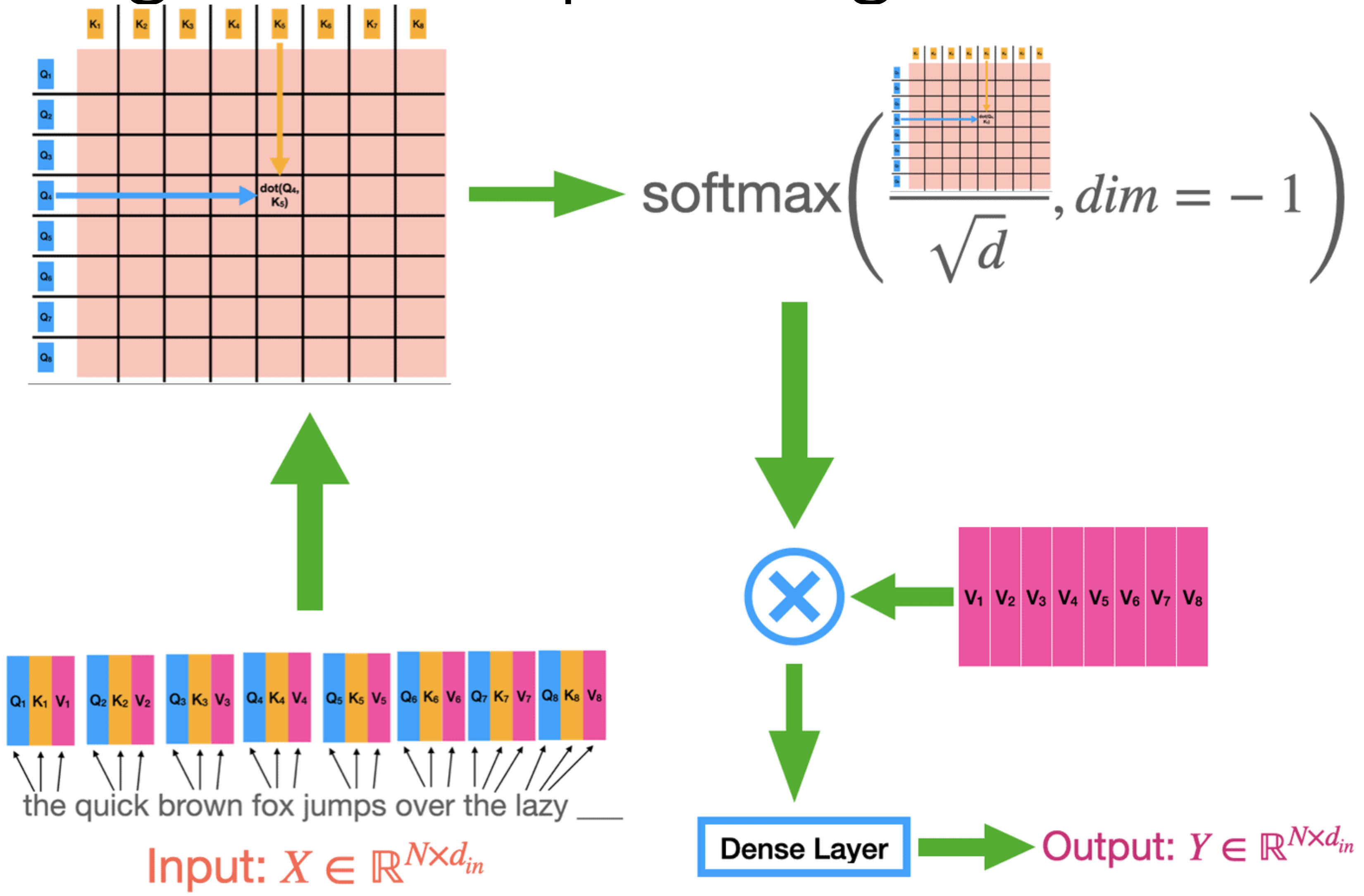


Embeddings are processed in parallel with a simple matrix multiplication

Attention matrix
each dot product show the similarity between a query and key vector
(softmax is applied row-wise)

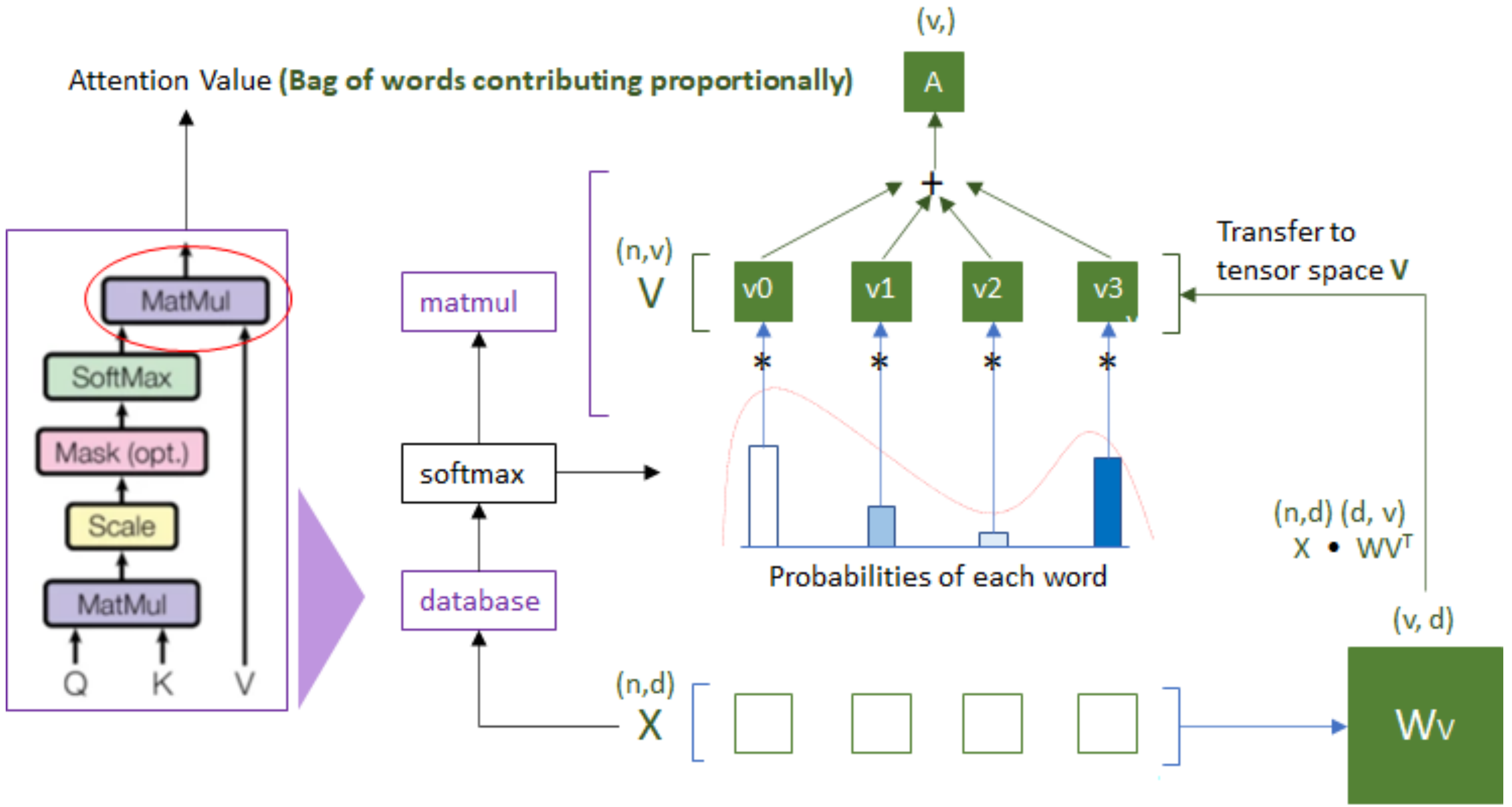
The output weighted values:
information is aggregated/routed

Multiply attention weights Softmax(Q.K^T) by V to get the output weighted values



K₁ K₂ K₃ K₄ K₅ K₆ K₇ K₈
Q₁ Q₂ Q₃ Q₄ Q₅ Q₆ Q₇ Q₈
dot(Q₄, K₅)
softmax
$\mathrm{softmax}\left(\frac{\cdot}{\sqrt{d}}, dim = -1\right)$
V₁ V₂ V₃ V₄ V₅ V₆ V₇ V₈
Q₁ K₁ V₁ Q₂ K₂ V₂ Q₃ K₃ V₃ Q₄ K₄ V₄ Q₅ K₅ V₅ Q₆ K₆ V₆ Q₇ K₇ V₇ Q₈ K₈ V₈
the quick brown fox jumps over the lazy ___
Input: $X \in \mathbb{R}^{N \times d_{in}}$
Dense Layer
Output: $Y \in \mathbb{R}^{N \times d_{in}}$

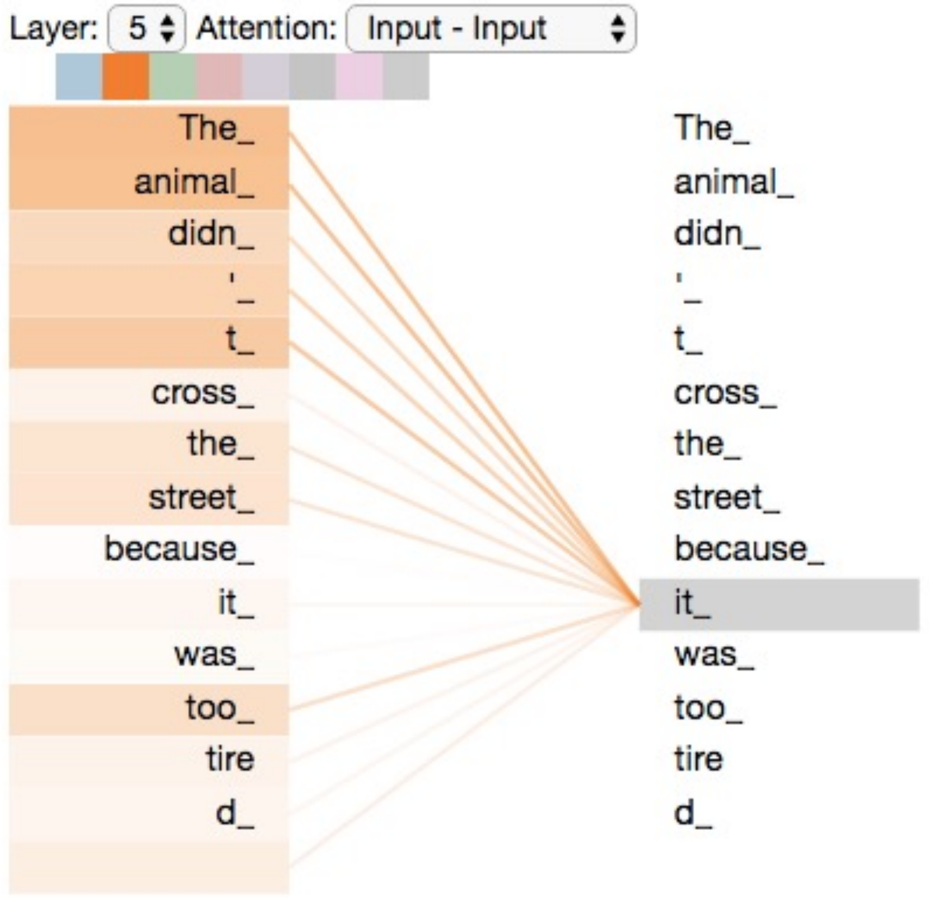
Multiply attention weights Softmax(Q.K^T) by V to get the output weighted values



Self-Attention: Main Takeaway

How much should each item in a sequence weight / prioritize / attend to elements of itself?

Keys

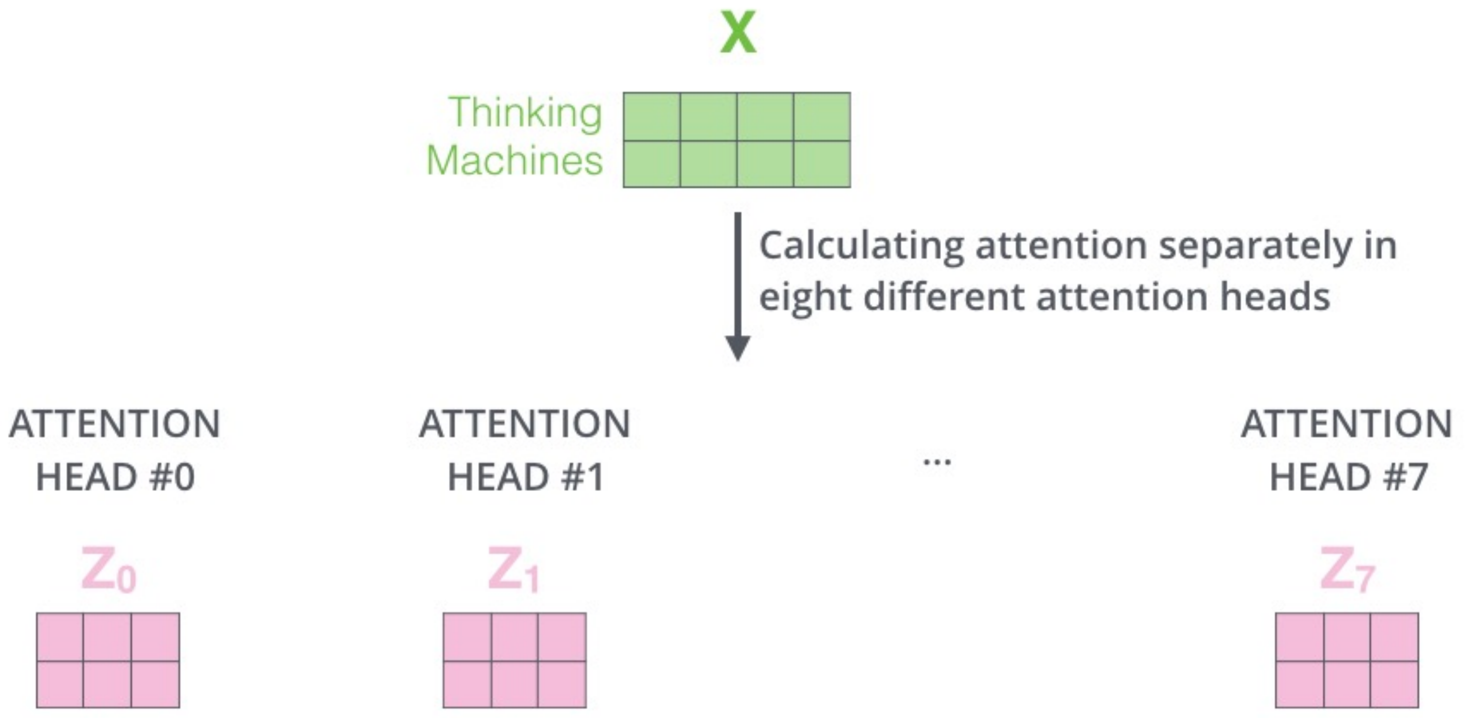


Query

Multi-Headed Attention

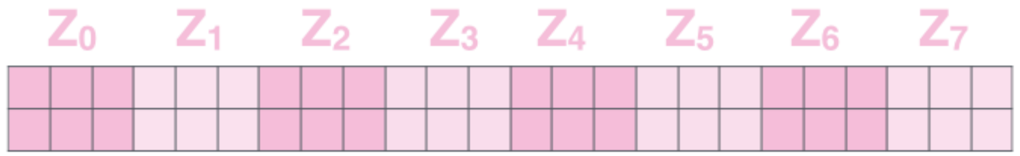
Calculate attention differently in multiple attention "heads"

This gives the attention layer multiple "representation subspaces" and expands the model's ability to focus on different positions



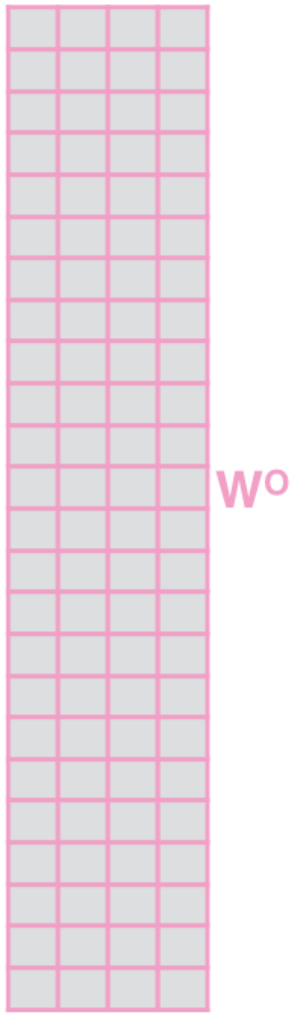
Multi-Headed Attention

1) Concatenate all the attention heads



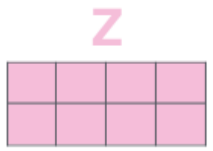
2) Multiply with a weight matrix W^O that was trained jointly with the model

X

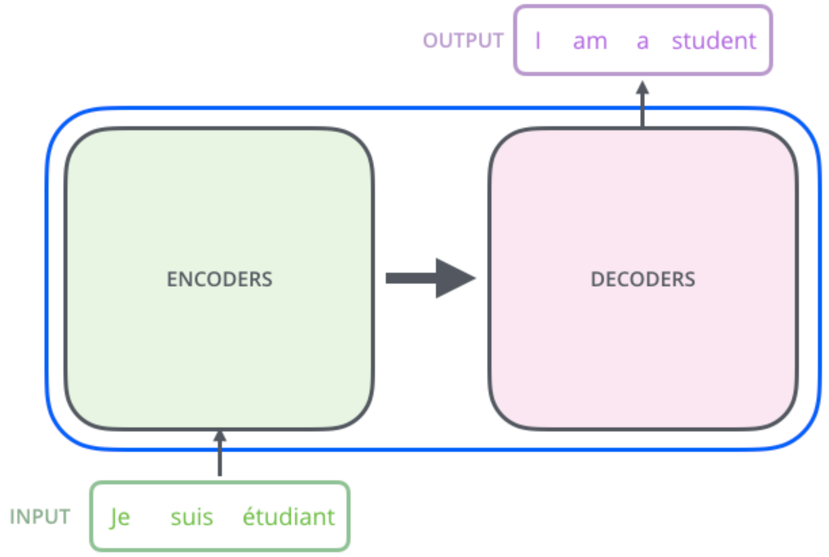


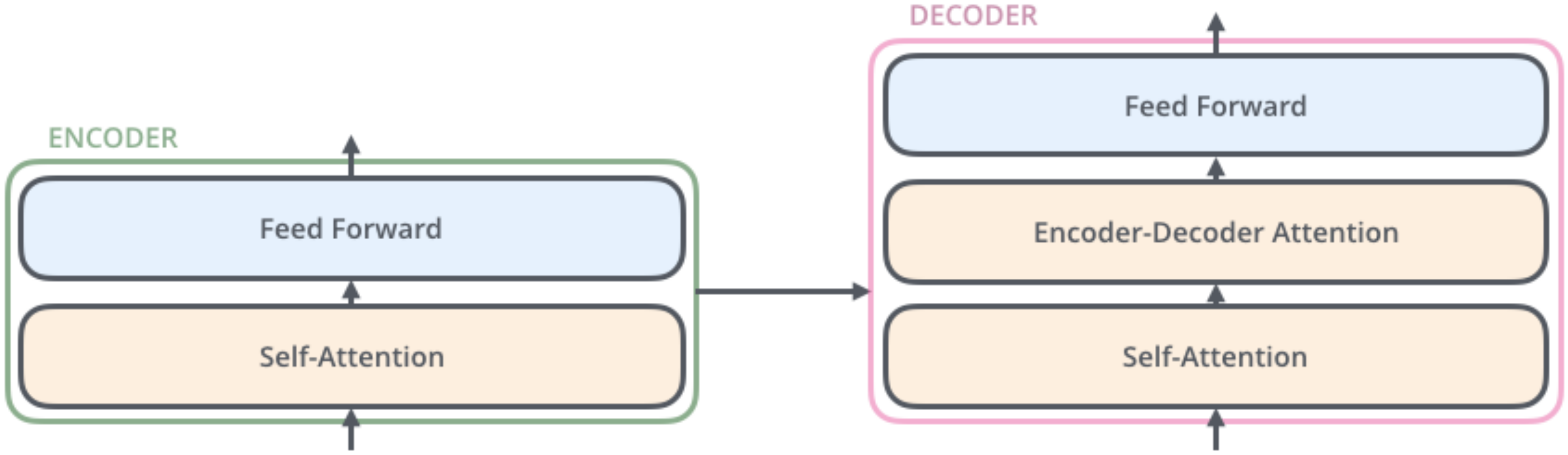
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

=

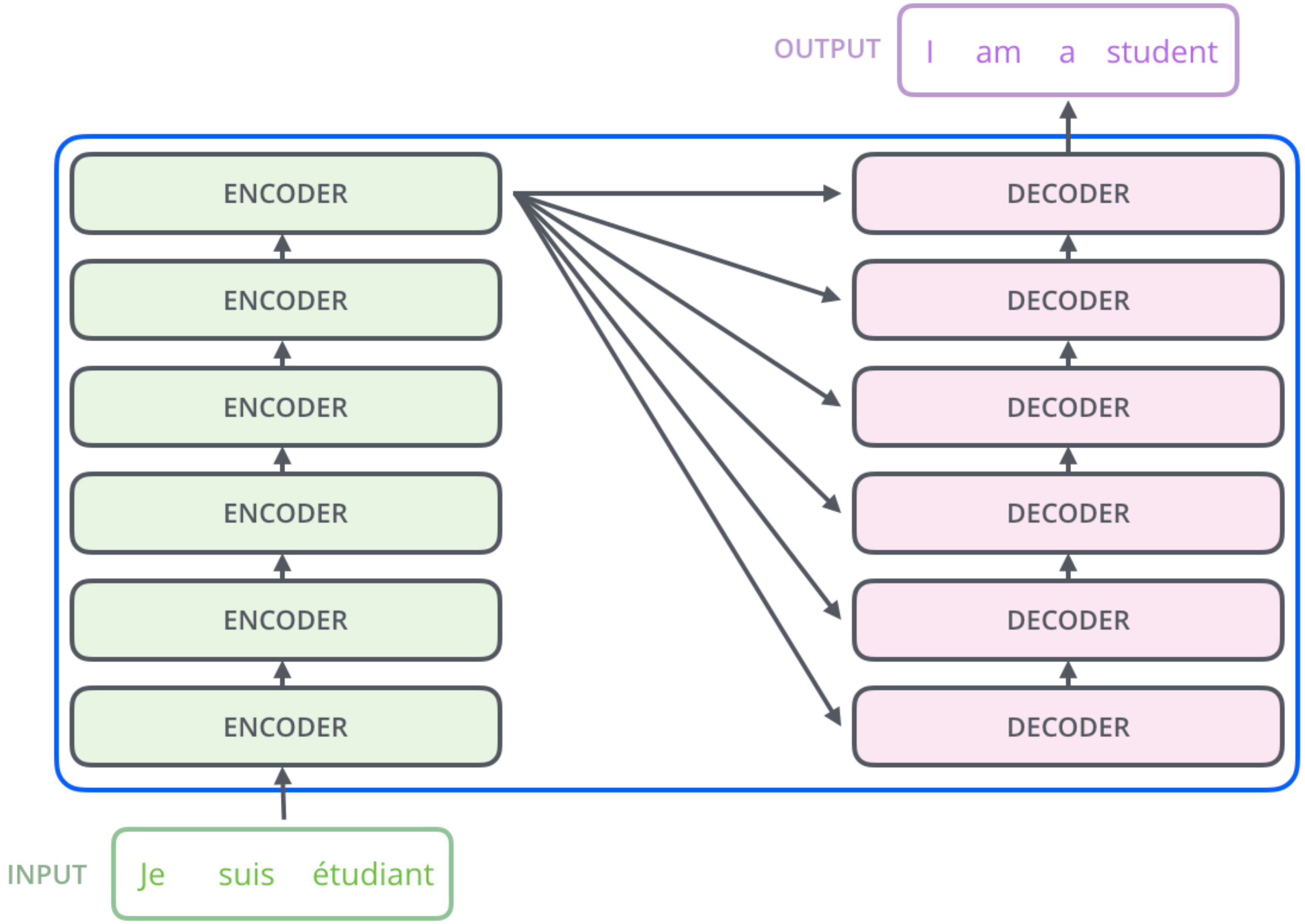


Attention throughout the Transformer

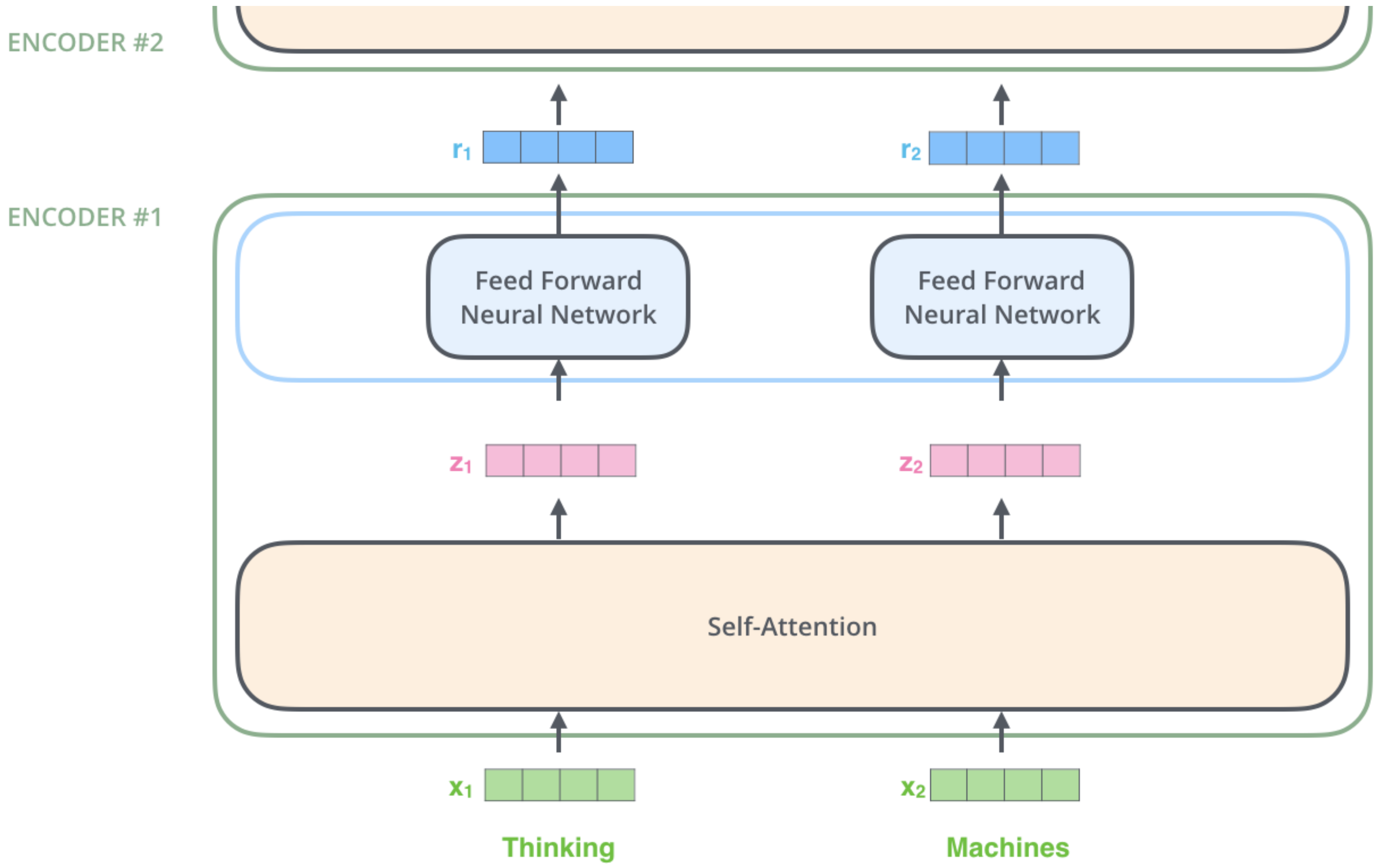




Transformers



Transformers



Transformers

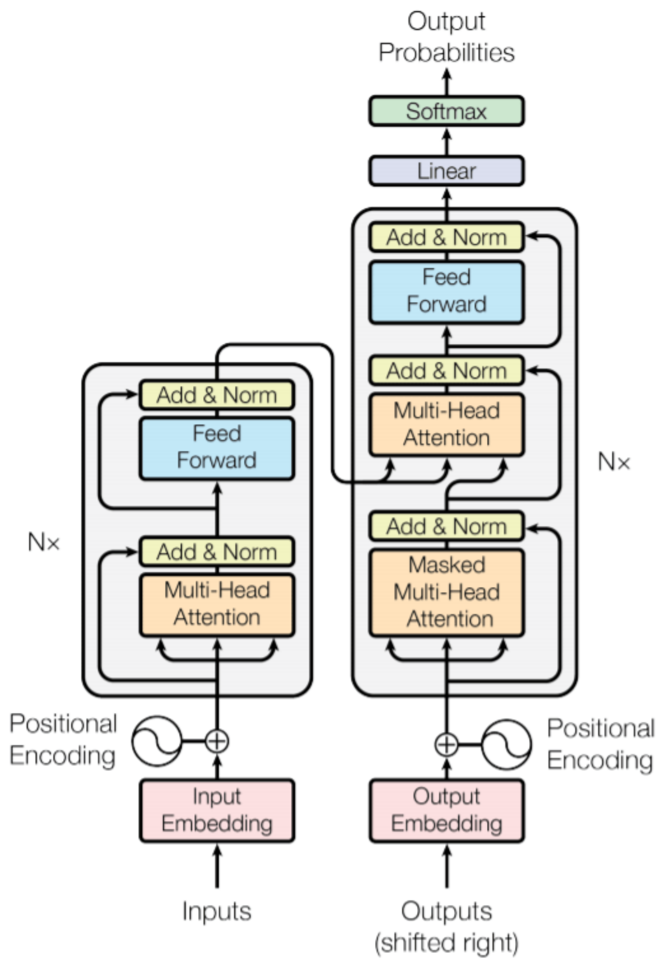


Figure 1: The Transformer - model architecture.

Transformers

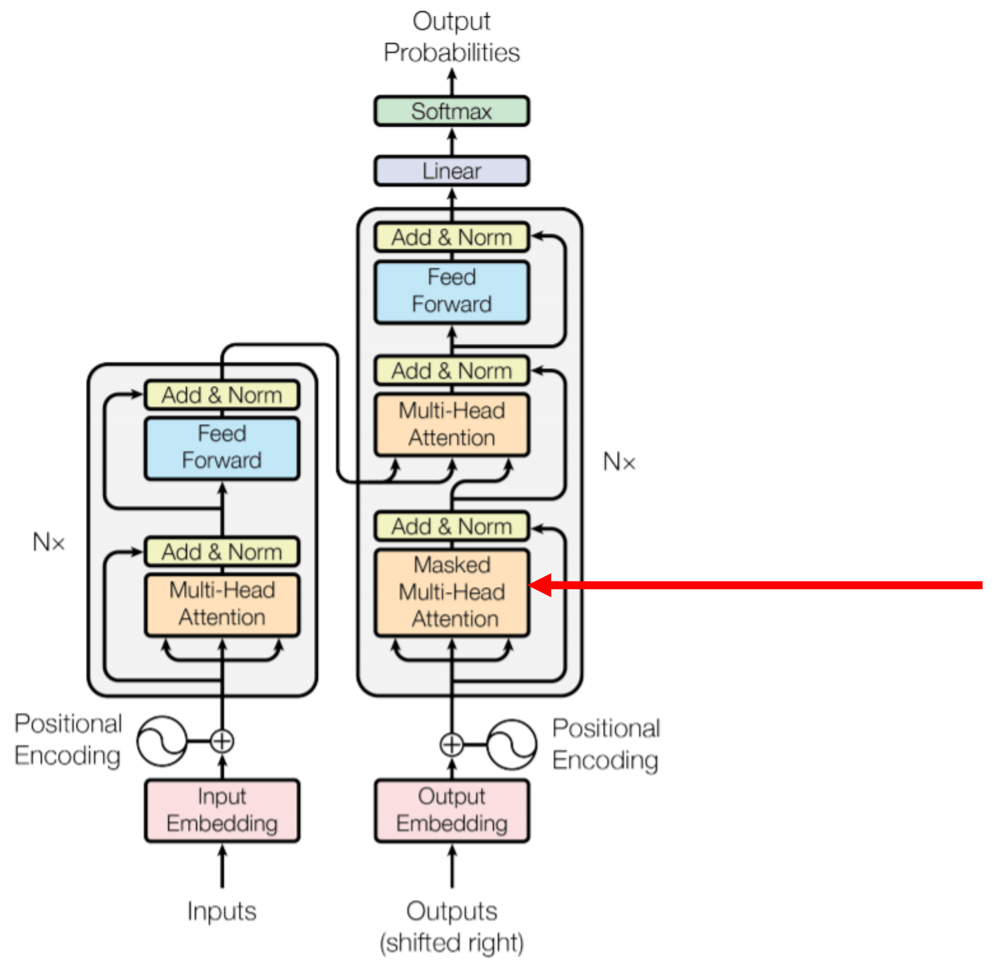
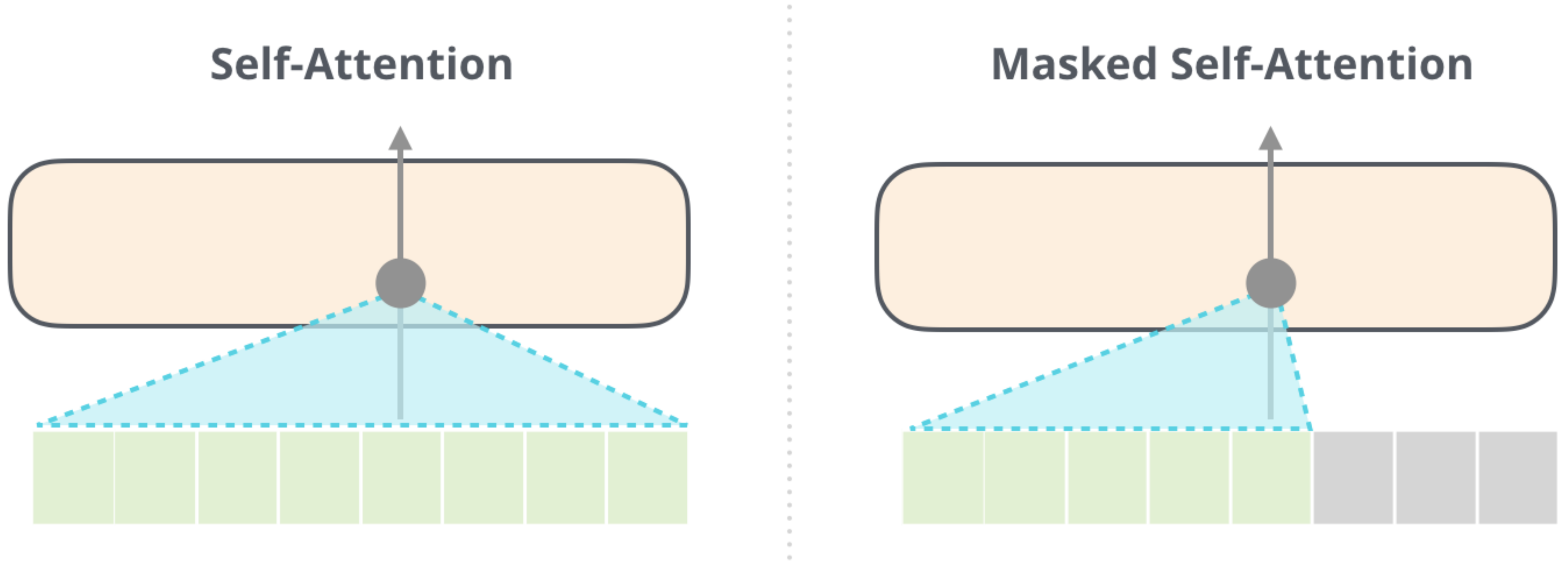


Figure 1: The Transformer - model architecture.

Masked Attention

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to -infinity) before the softmax step in the self-attention calculation.



Transformers

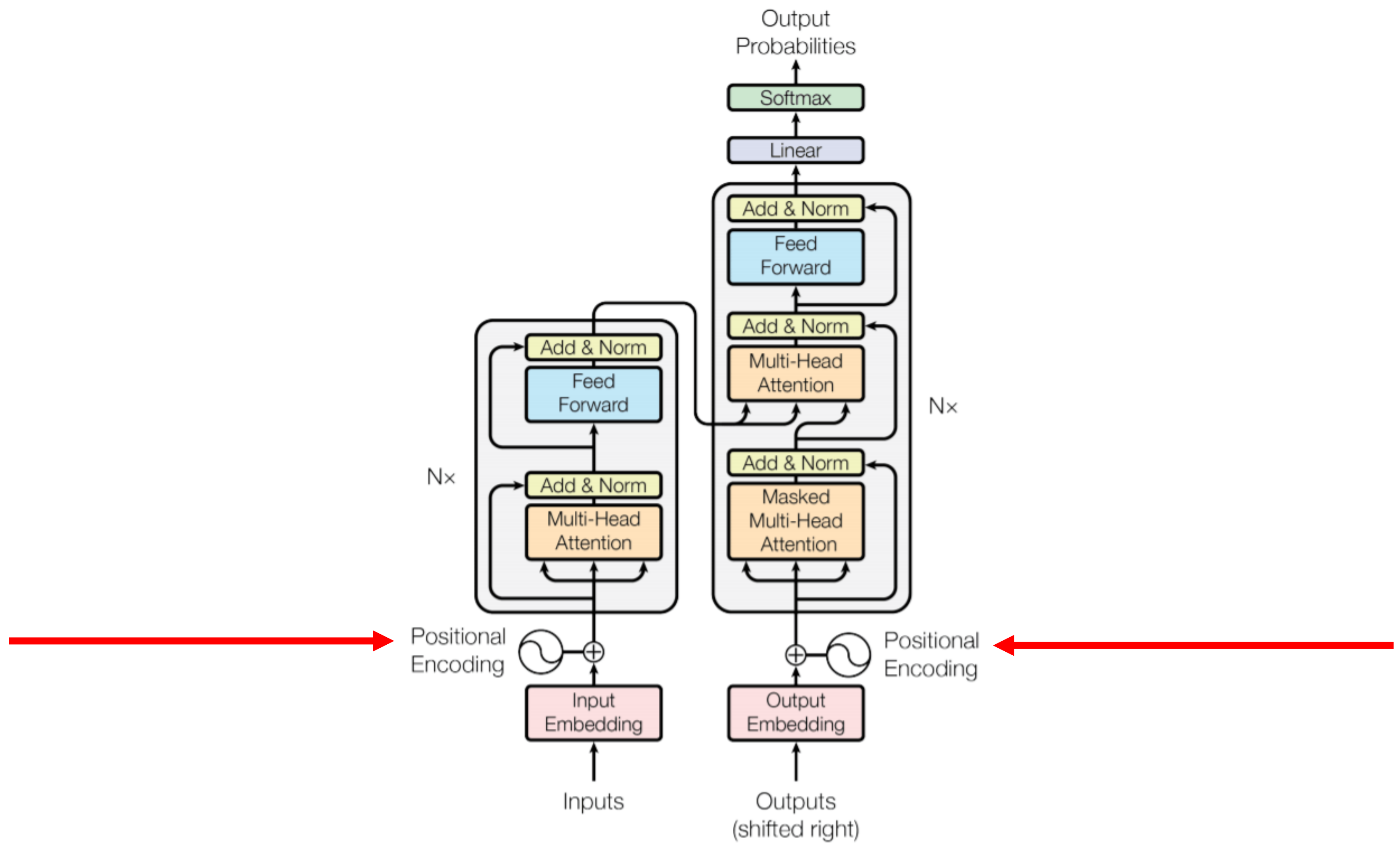
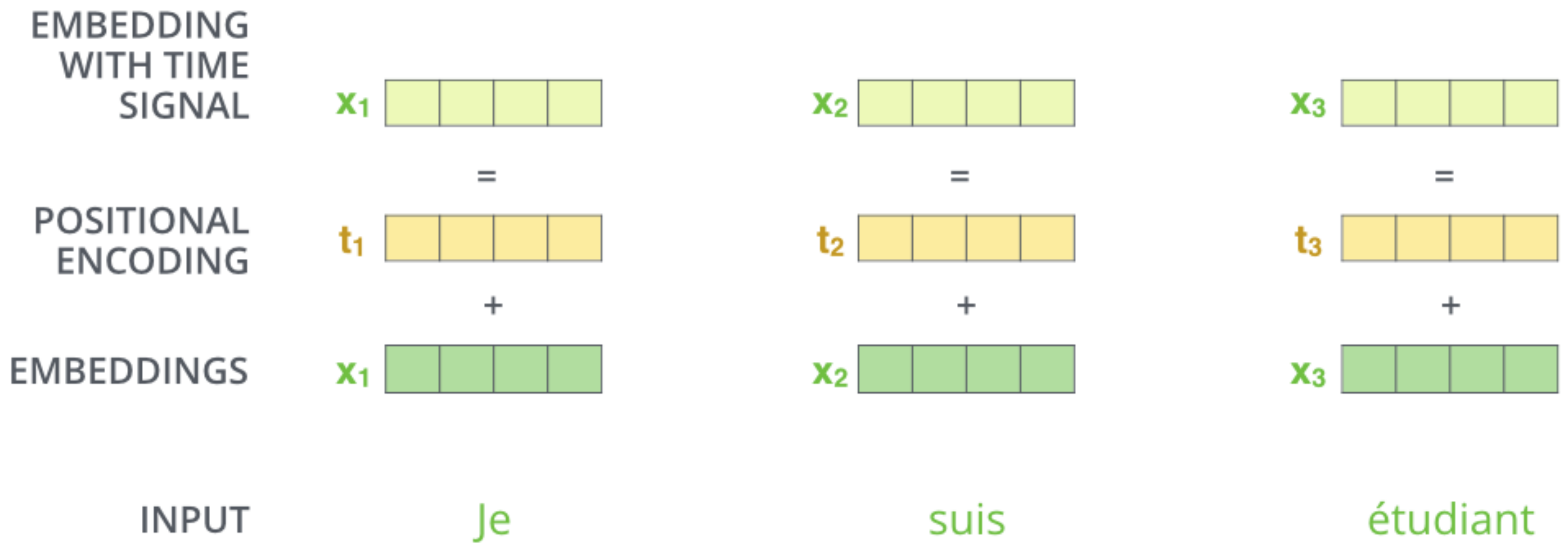


Figure 1: The Transformer - model architecture.

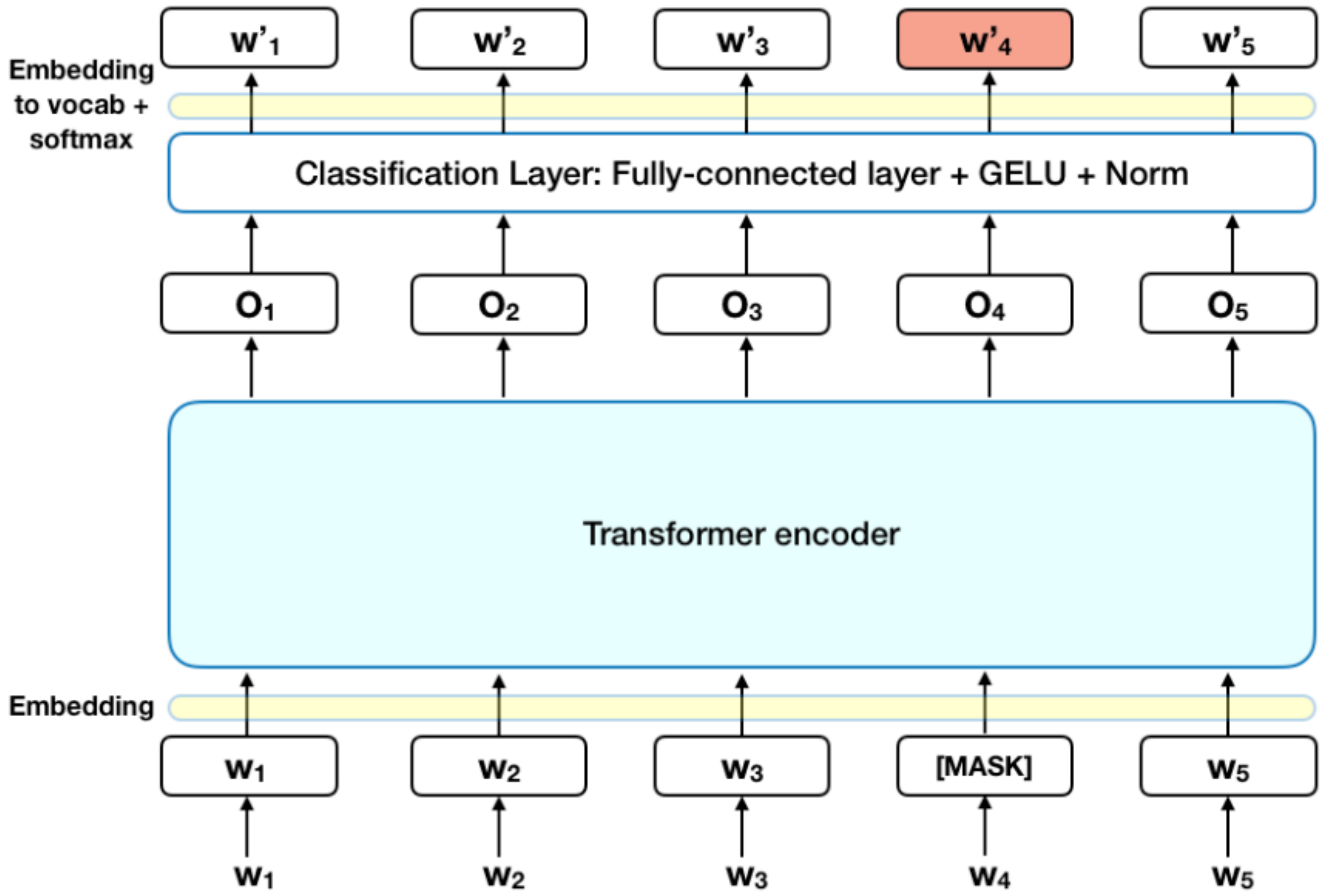
Positional Embeddings

Positional embeddings explicitly encode the position of each input element within the sequence.



Transformer-Inspired Models

Bidirectional Encoder Representations from Transformers (BERT) (2018)

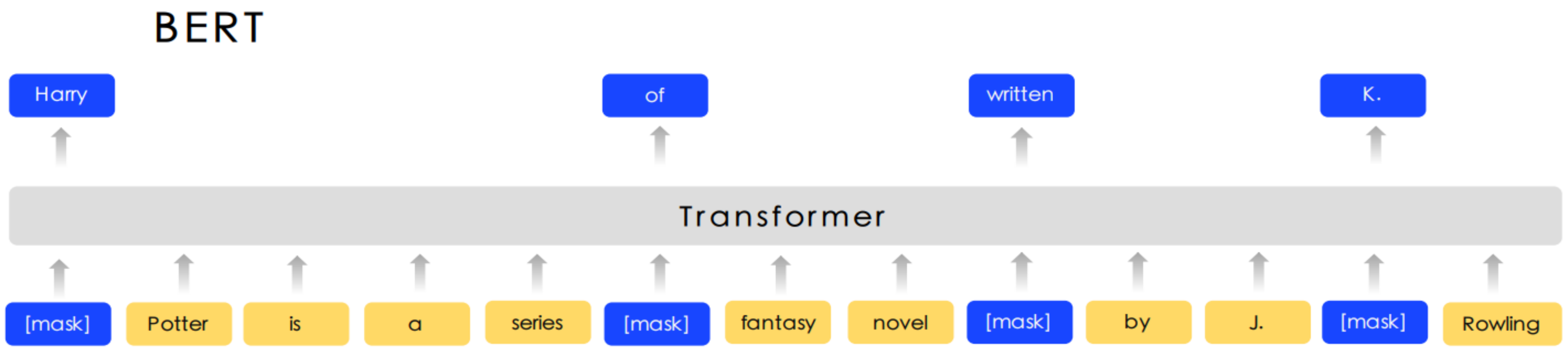

w'₁
w'₂
w'₃
w'₄
w'₅
Embedding to vocab + softmax
Classification Layer: Fully-connected layer + GELU + Norm
O₁
O₂
O₃
O₄
O₅
Transformer encoder
Embedding
w₁
w₂
w₃
[MASK]
w₅
w₁
w₂
w₃
w₄
w₅

BERT

- Composed of Transformer encoder layers
- BERT was pre-trained on the Toronto BookCorpus (800M words) and English Wikipedia (2,500M words)
- The power of the BERT models largely comes from how it is pre-trained: using labels automatically derived from the text rather than human-generated labels
    - This enables the bottleneck of this method to be the amount of training data rather than the amount of human labels
    - The broad term for this technique is **self-supervised learning**

BERT Pre-Training Task #1: Masked Language Modeling

Randomly mask out words and predict the missing words



BERT Pre-Training Task #2: Next Sentence Prediction

Predict whether one sentence comes after the next (binary classification)

Sentence 1	Sentence 2	Next Sentence?
I have a class	I will be back by 6	✓
I have a class	Zebra is a animal	✗

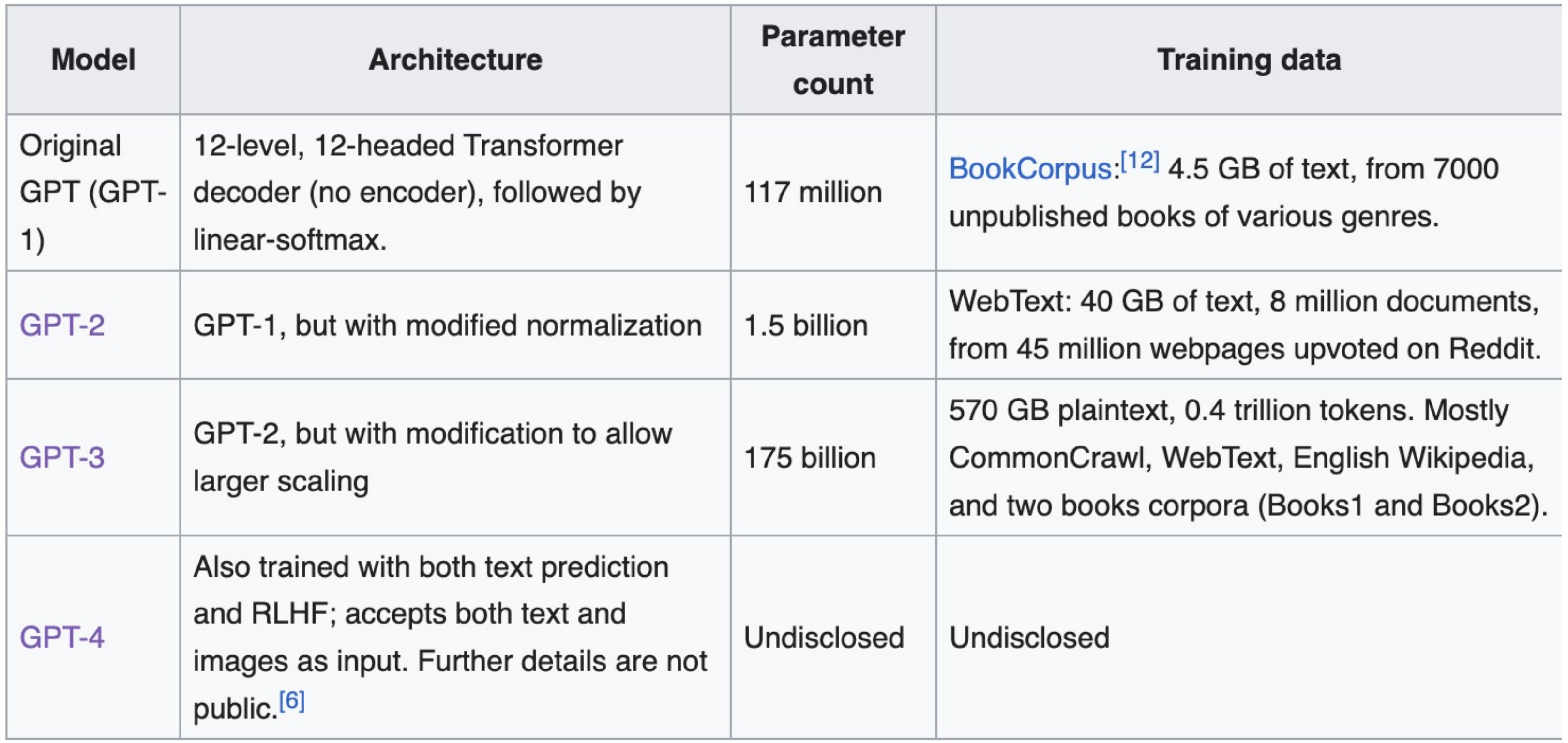
Large Pre-Trained Language Models

- BERT was the start of an emerging trend: large language models
- Large language models pre-trained on as much text as can be collected and stored on a computer
- Transfer learning over pre-trained language models like BERT often yields state-of-the-art performance

GPT: another large language Transformer model

Generative Pre-trained Transformer (GPT)

- GPT-1 (2018, proof of concept not released publicly)
- GPT-2 (2019)
- GPT-3 (2020)
- GPT-4 (2023)



Model	Architecture	Parameter count	Training data
Original GPT (GPT-1)	12-level, 12-headed Transformer decoder (no encoder), followed by linear-softmax.	117 million	BookCorpus:[12] 4.5 GB of text, from 7000 unpublished books of various genres.
GPT-2	GPT-1, but with modified normalization	1.5 billion	WebText: 40 GB of text, 8 million documents, from 45 million webpages upvoted on Reddit.
GPT-3	GPT-2, but with modification to allow larger scaling	175 billion	570 GB plaintext, 0.4 trillion tokens. Mostly CommonCrawl, WebText, English Wikipedia, and two books corpora (Books1 and Books2).
GPT-4	Also trained with both text prediction and RLHF; accepts both text and images as input. Further details are not public.[6]	Undisclosed	Undisclosed

GPT is a Transformer Decoder

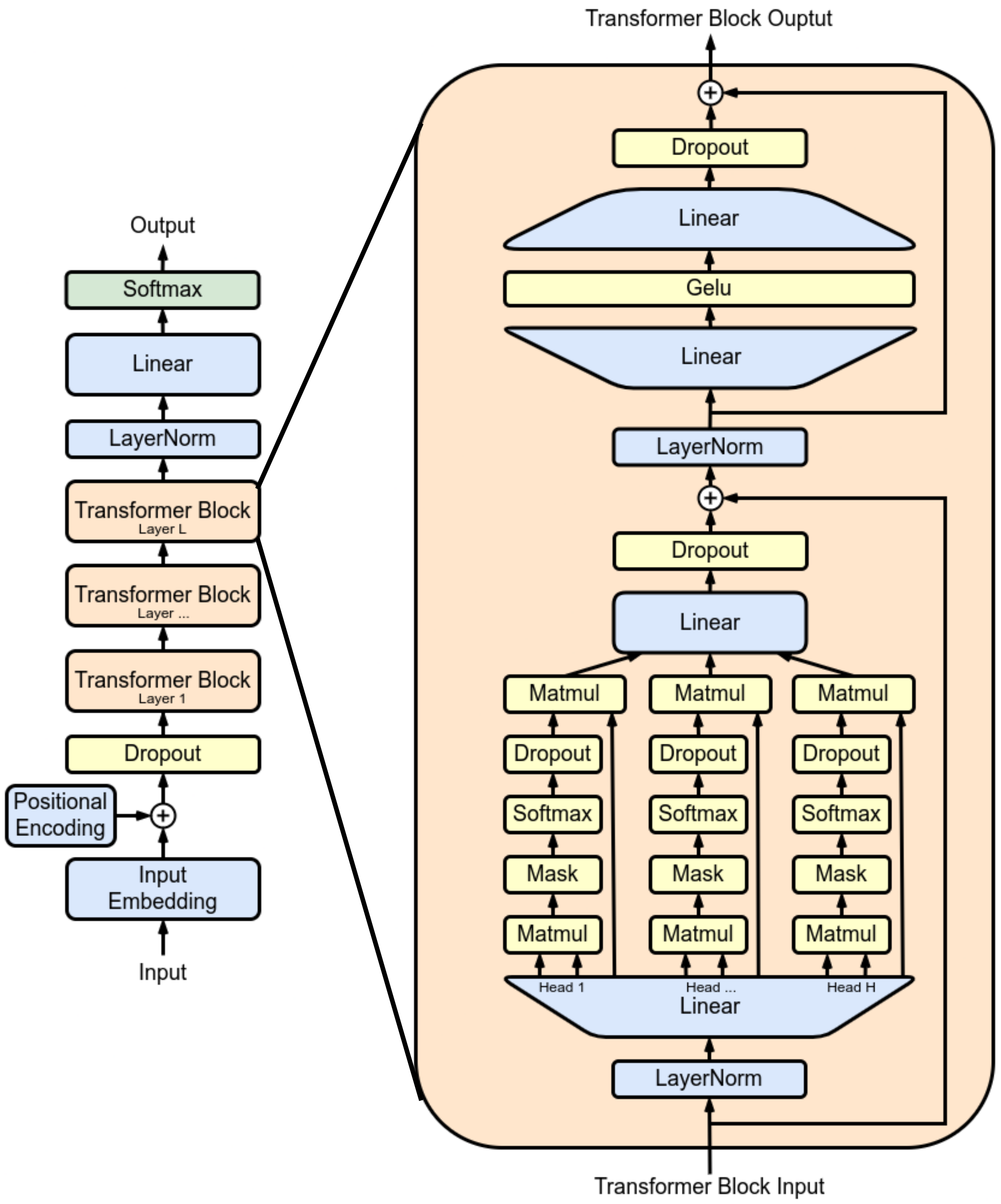
Transformer Encoders (e.g., BERT)

- Use all sequence tokens to attend to each token in the sequence (unmasked attention)

Transformer Decoders (e.g., GPT)

- Use only the tokens preceding the current token to attend to each token in the sequence (masked attention)
- This makes it natural for **autoregressive modeling** (predicting the next time step from the previous time steps)

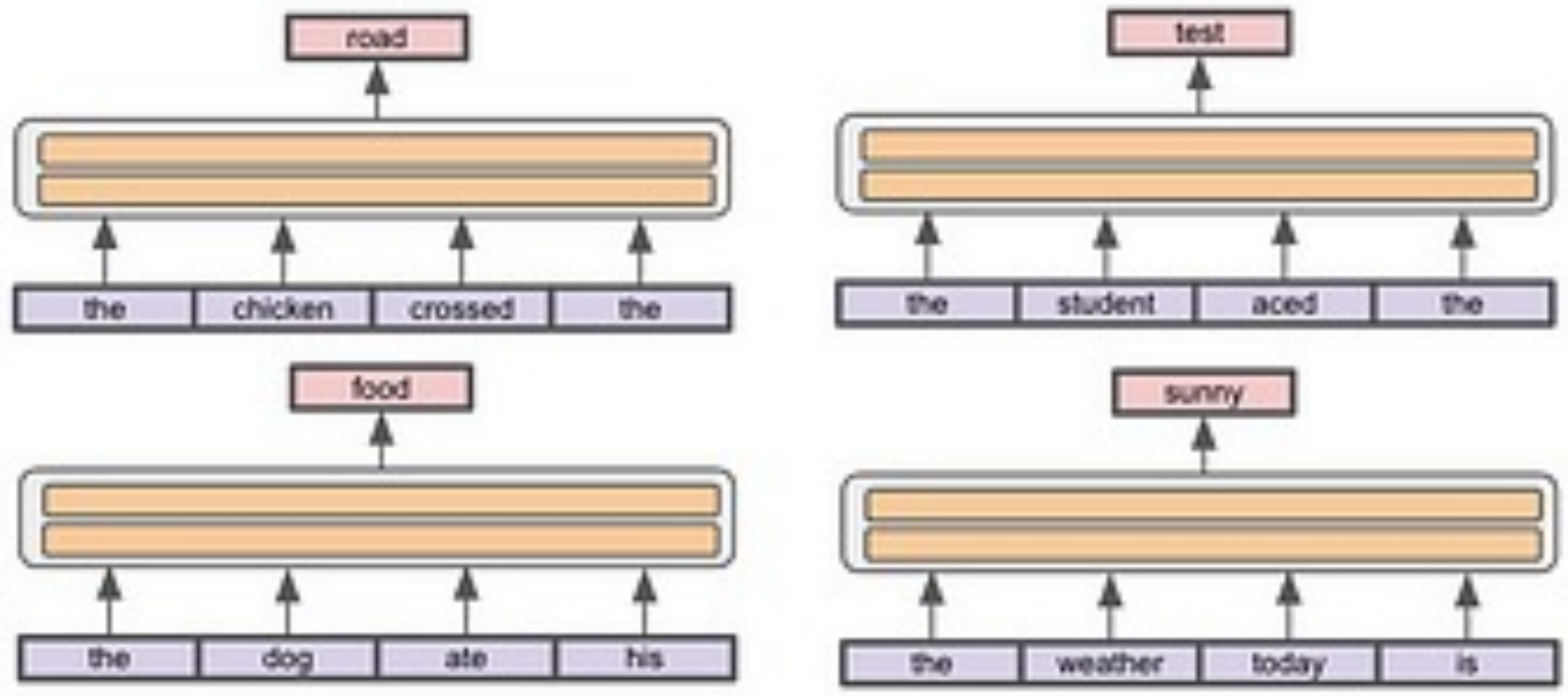
GPT Architecture



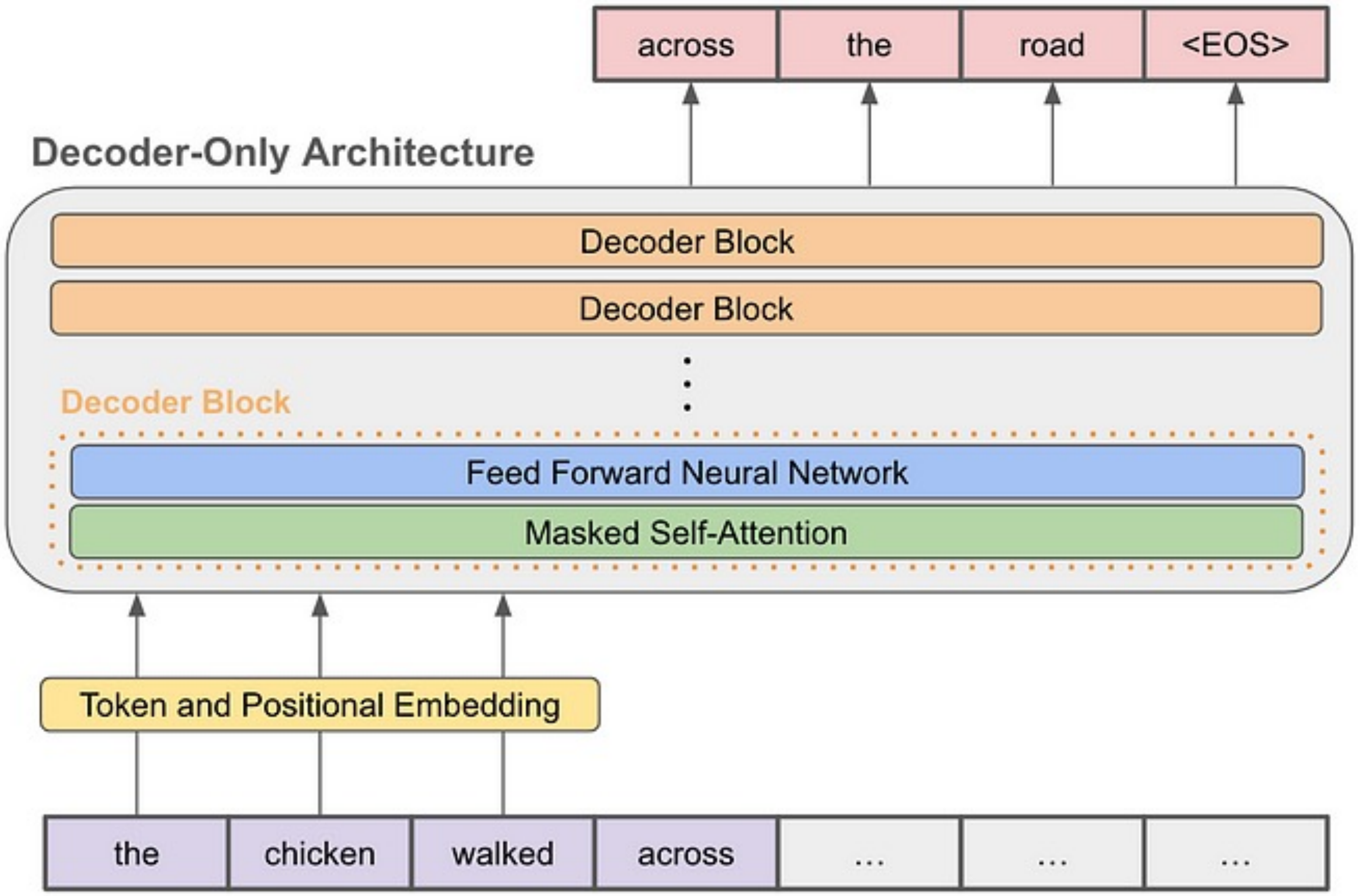
Generative Pre-training for GPT

Predict the next word.

Just like BERT pre-training, doesn't require human labels! Just need text.



Generative Pre-training for GPT



Generative Pre-training for GPT

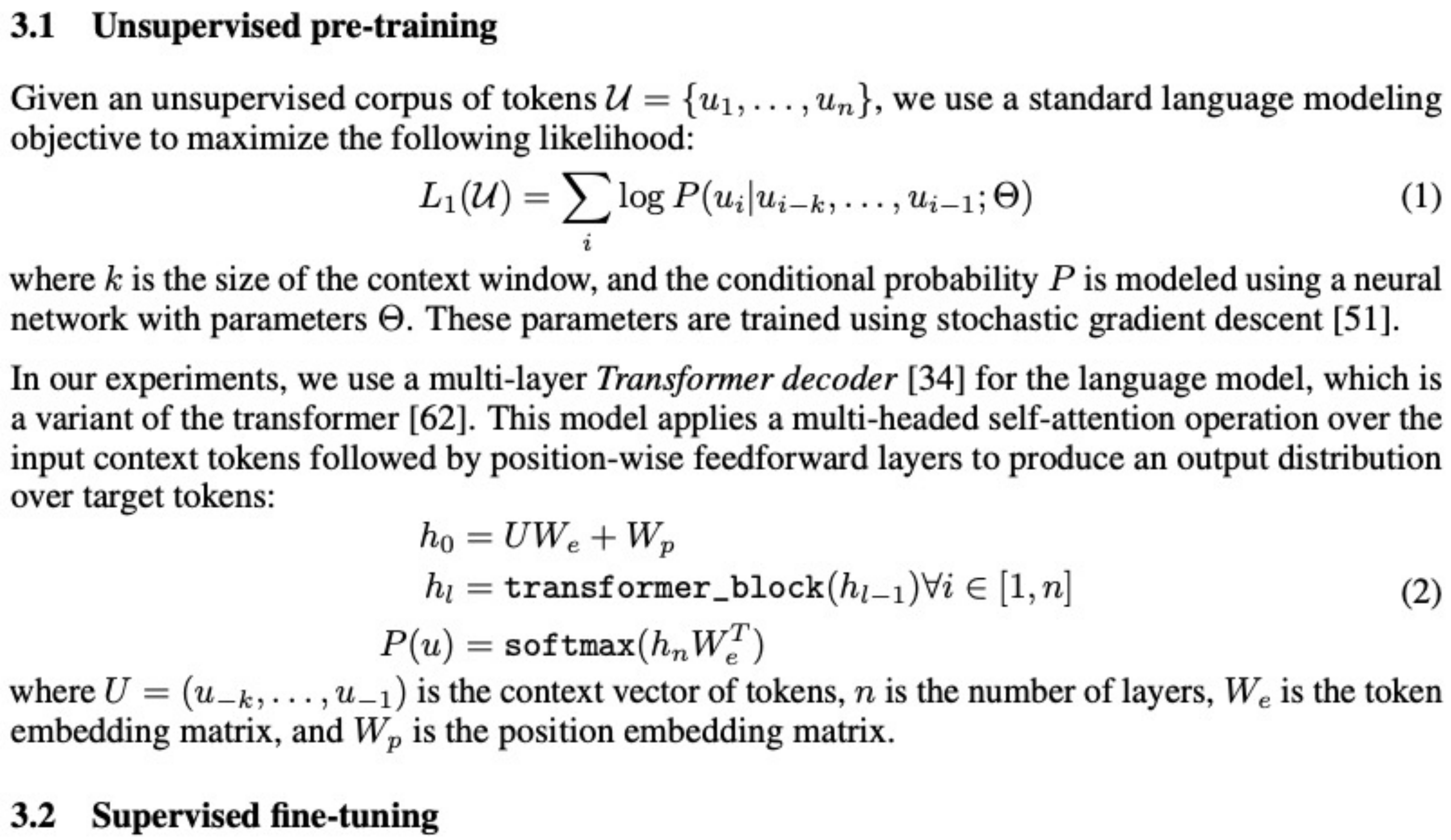
1. Sample text from the pre-training corpus

2. Predict the next token with our model

3. Use stochastic gradient descent (SGD) or any other optimizer to increase the probability of the correct next token

4. Repeat

At this point in the class, you should be able to understand the original GPT paper’s description of this process. 🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉



3.1 Unsupervised pre-training

Given an unsupervised corpus of tokens $\mathcal{U} = \{u_1, \ldots, u_n\}$, we use a standard language modeling objective to maximize the following likelihood:

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta) \tag{1}$$

where k is the size of the context window, and the conditional probability P is modeled using a neural network with parameters Θ. These parameters are trained using stochastic gradient descent [51].

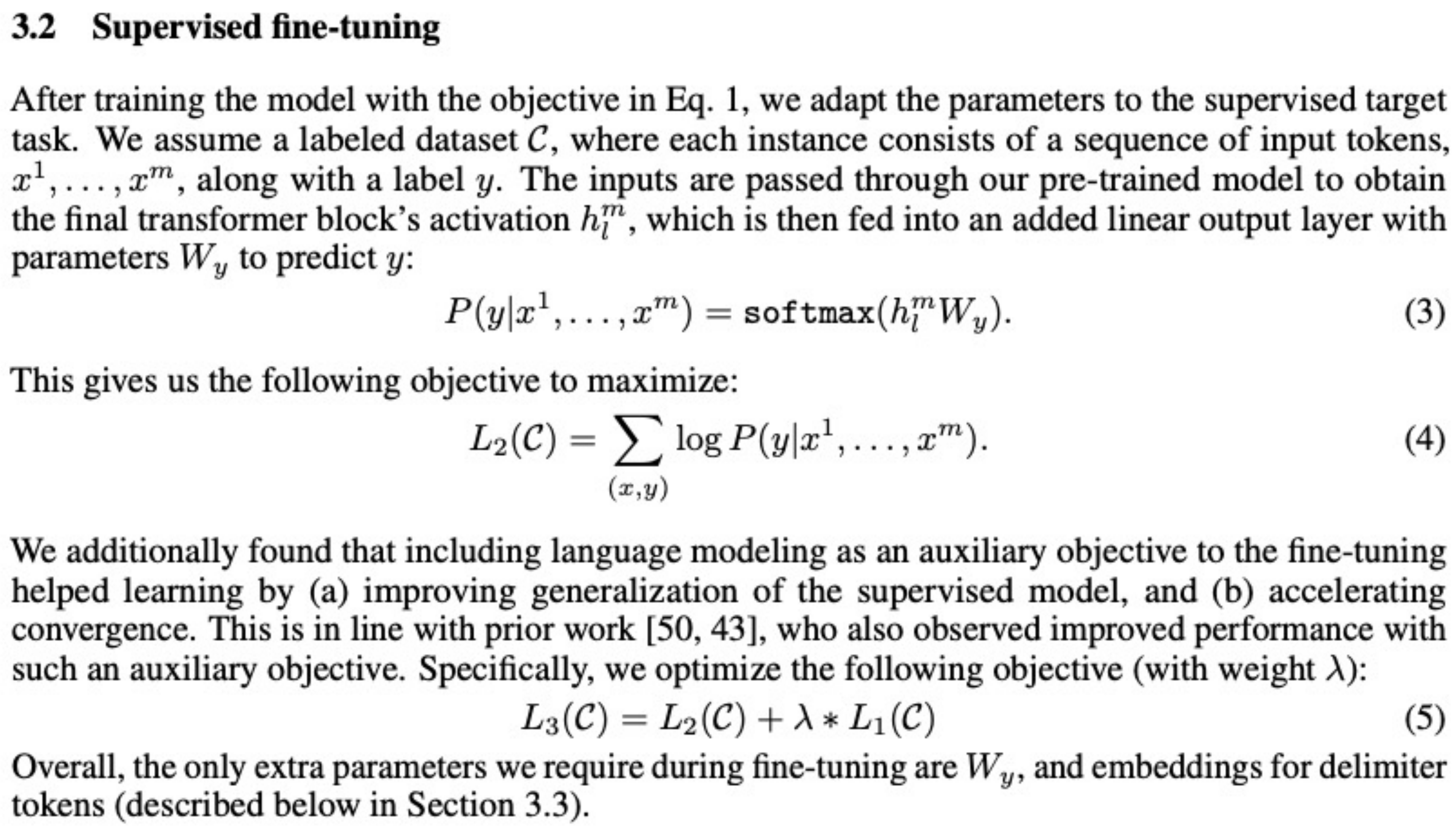
In our experiments, we use a multi-layer *Transformer decoder* [34] for the language model, which is a variant of the transformer [62]. This model applies a multi-headed self-attention operation over the input context tokens followed by position-wise feedforward layers to produce an output distribution over target tokens:

$$\begin{aligned} h_0 &= UW_e + W_p \\ h_l &= \texttt{transformer\_block}(h_{l-1}) \forall i \in [1, n] \\ P(u) &= \texttt{softmax}(h_n W_e^T) \end{aligned} \tag{2}$$

where $U = (u_{-k}, \ldots, u_{-1})$ is the context vector of tokens, n is the number of layers, W_e is the token embedding matrix, and W_p is the position embedding matrix.

3.2 Supervised fine-tuning

The authors then fine-tune to several tasks



3.2 Supervised fine-tuning

After training the model with the objective in Eq. 1, we adapt the parameters to the supervised target task. We assume a labeled dataset $\mathcal{C}$, where each instance consists of a sequence of input tokens, $x^1, \ldots, x^m$, along with a label y. The inputs are passed through our pre-trained model to obtain the final transformer block's activation h_l^m, which is then fed into an added linear output layer with parameters W_y to predict y:

$$P(y|x^1, \ldots, x^m) = \texttt{softmax}(h_l^m W_y). \tag{3}$$

This gives us the following objective to maximize:

$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y|x^1, \ldots, x^m). \tag{4}$$

We additionally found that including language modeling as an auxiliary objective to the fine-tuning helped learning by (a) improving generalization of the supervised model, and (b) accelerating convergence. This is in line with prior work [50, 43], who also observed improved performance with such an auxiliary objective. Specifically, we optimize the following objective (with weight λ):

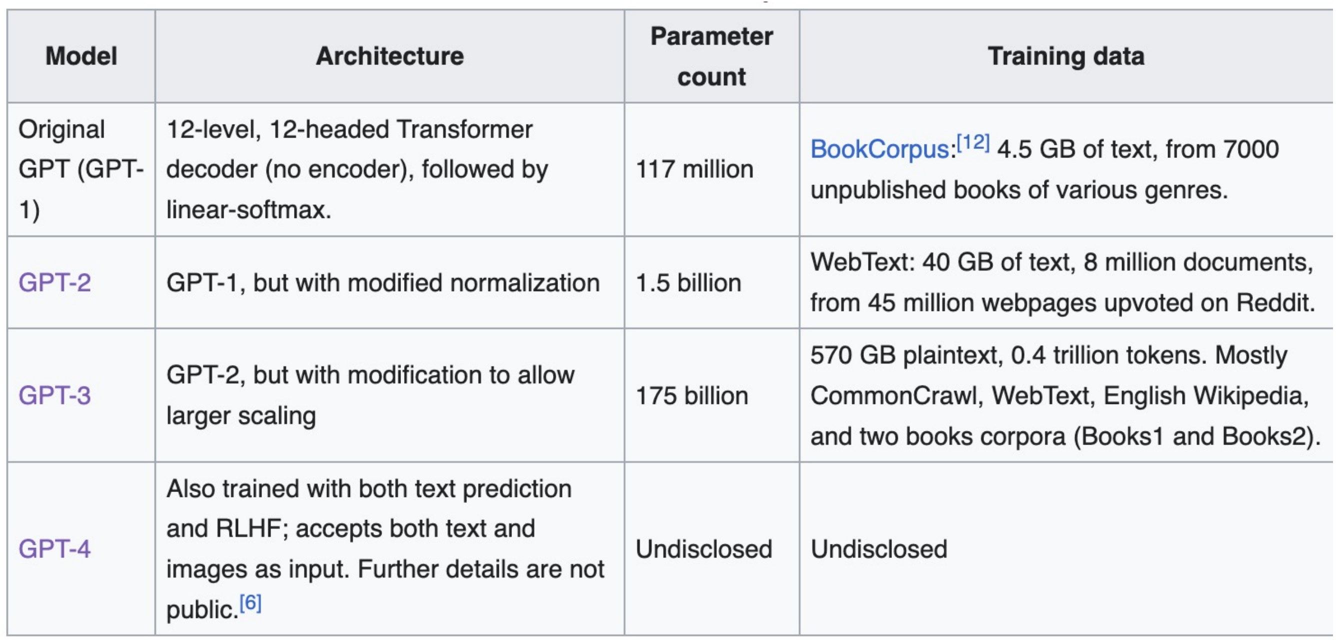
$$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda * L_1(\mathcal{C}) \tag{5}$$

Overall, the only extra parameters we require during fine-tuning are W_y, and embeddings for delimiter tokens (described below in Section 3.3).

GPT1 vs 2 vs 3 vs 4

Each successive GPT version is:

- Much larger
- Trained on a larger dataset
- Has minor architectural differences to allow for scaling



Model	Architecture	Parameter count	Training data
Original GPT (GPT-1)	12-level, 12-headed Transformer decoder (no encoder), followed by linear-softmax.	117 million	BookCorpus:[12] 4.5 GB of text, from 7000 unpublished books of various genres.
GPT-2	GPT-1, but with modified normalization	1.5 billion	WebText: 40 GB of text, 8 million documents, from 45 million webpages upvoted on Reddit.
GPT-3	GPT-2, but with modification to allow larger scaling	175 billion	570 GB plaintext, 0.4 trillion tokens. Mostly CommonCrawl, WebText, English Wikipedia, and two books corpora (Books1 and Books2).
GPT-4	Also trained with both text prediction and RLHF; accepts both text and images as input. Further details are not public.[6]	Undisclosed	Undisclosed

GPT-2 Architecture and Details

- 12-layer decoder-only Transformer with 12 masked self-attention heads
- the learning rate was increased linearly from zero over the first 2,000 updates, to a maximum of 2.5×10−4, and annealed to 0 using a cosine schedule

“We train for 100 epochs on minibatches of 64 randomly sampled, contiguous sequences of 512 tokens. Since layernorm is used extensively throughout the model, a simple weight initialization of N(0,0.02) was sufficient. We used a bytepair encoding (BPE) vocabulary with 40,000 merges and residual, embedding, and attention dropouts with a rate of 0.1 for regularization. We also employed a modified version of L2 regularization proposed in Loshchilov et al. 2017, with w = 0.01 on all non bias or gain weights.

[...]

We used learned position embeddings instead of the sinusoidal version proposed in the original work.

[...]

Unless specified, we reuse the hyperparameter settings from unsupervised pre-training. We add dropout to the classifier with a rate of 0.1. For most tasks, we use a learning rate of 6.25e-5 and a batchsize of 32. Our model finetunes quickly and 3 epochs of training was sufficient for most cases. We use a linear learning rate decay schedule with warmup over 0.2% of training. λ was set to 0.5.”

GPT-2 Architecture and Details

You already know all this stuff!!!

- 12-layer de sformer with 12 masked self-attention heads
- the learning rat linearly from zero over the first 2,000 updates, to a maximum of 2.5×10−4, and annealed to 0 u schedule

“We train for 100 epochs on m 4 randomly sampled, contiguous sequences of 512 tokens. Since layernorm is used extensively thr odel, a simple weight initialization of N(0,0.02) was sufficient. We used a bytepair encoding (BPE) v 40,000 merges and residual, embedding, and attention dropouts with a rate of 0.1 for regulariz employed a modified version of L2 regularization proposed in Loshchilov et al. 2017, with w n bias or gain weights.

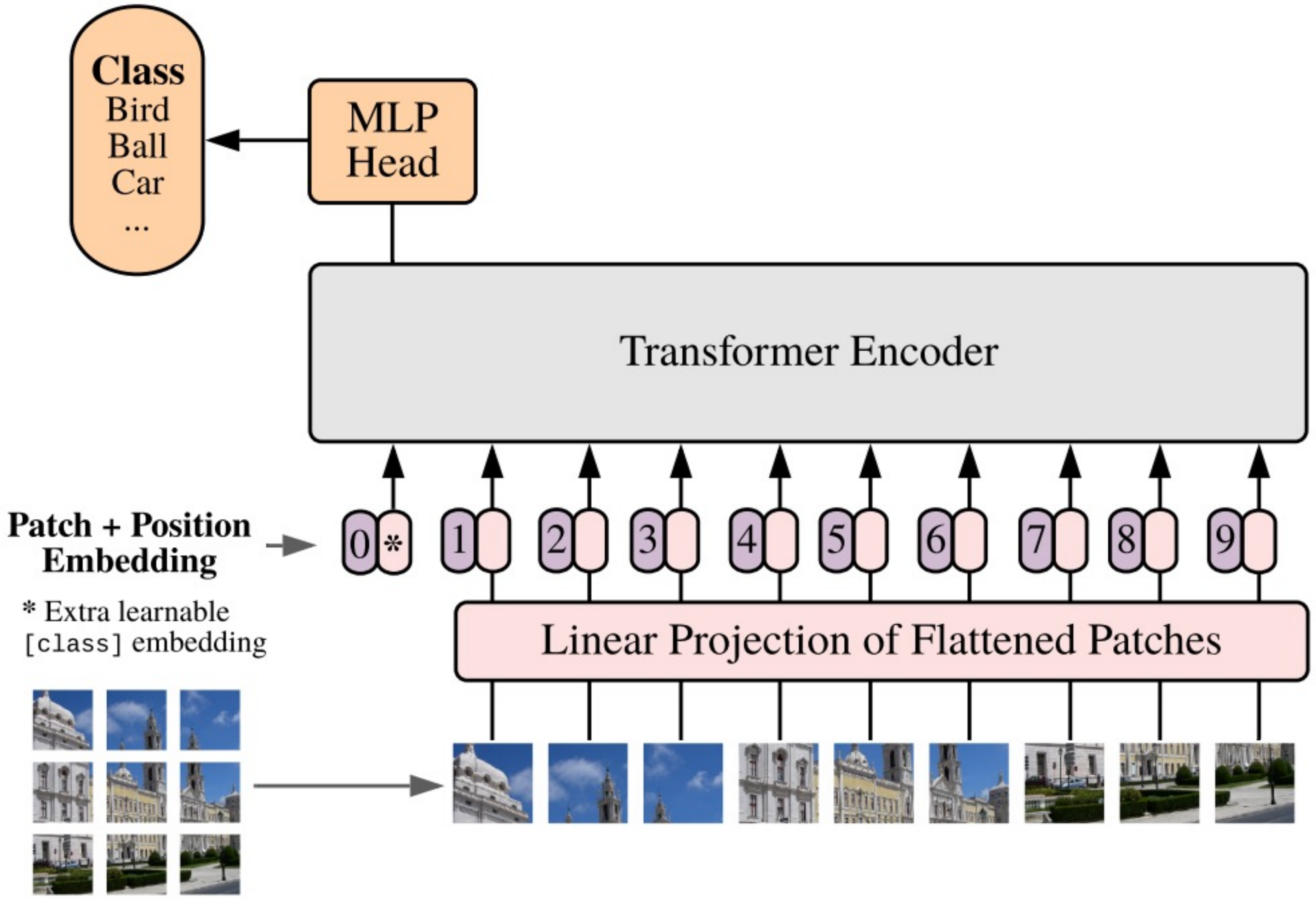
[...]

We used learned position embeddings instead of the sinu proposed in the original work.

[...]

Unless specified, we reuse the hyperparameter settings from unsupervis . We add dropout to the classifier with a rate of 0.1. For most tasks, we use a learning rate of 6.25e-5 ze of 32. Our model finetunes quickly and 3 epochs of training was sufficient for most cases. We us ning rate decay schedule with warmup over 0.2% of training. λ was set to 0.5.”

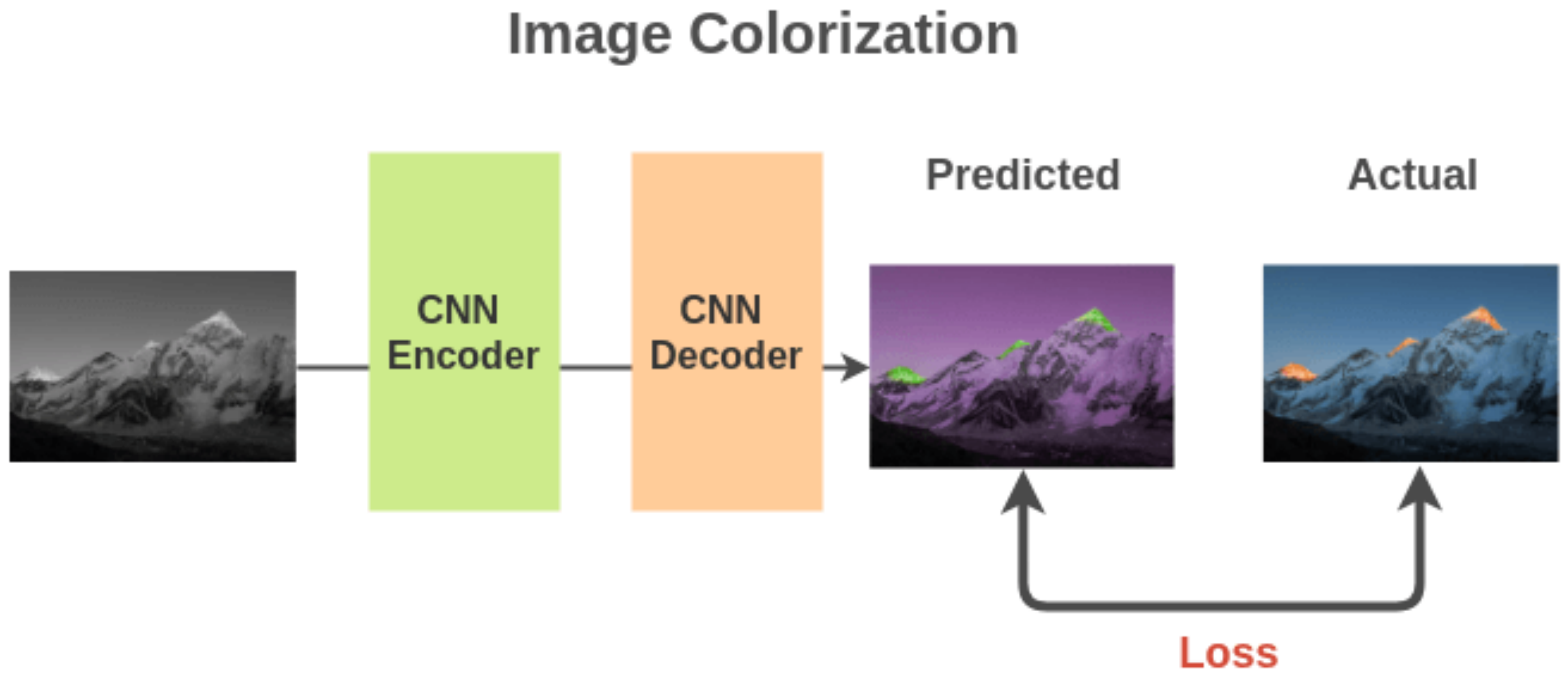
Transformers have also yielded state-of-the-art computer vision performance


Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
0 *
1
2
3
4
5
6
7
8
9
* Extra learnable [class] embedding
Linear Projection of Flattened Patches

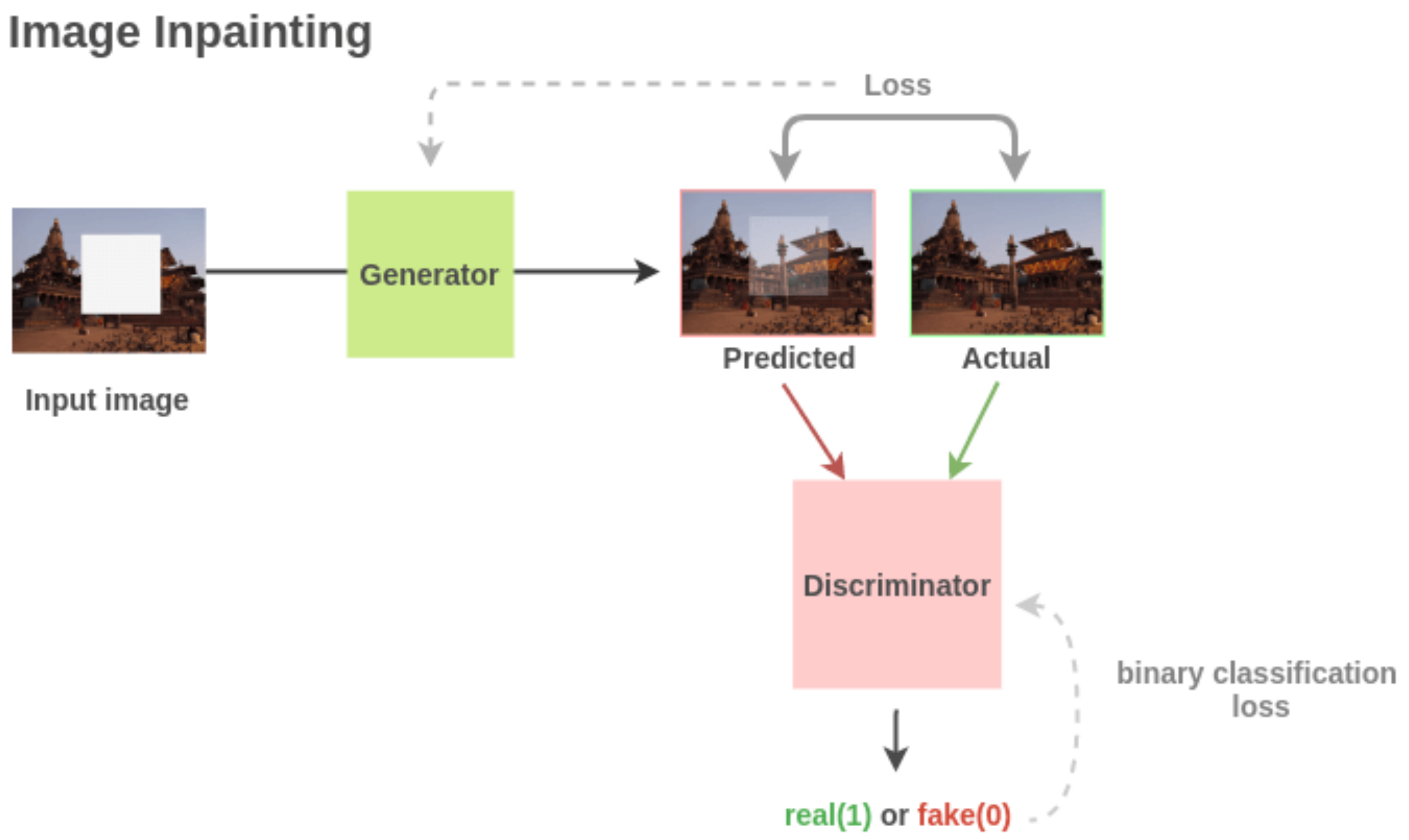
Self-Supervised Learning

- Generalization of the pre-training in BERT and GPT models
- Pre-training a neural network’s weights using **y_train** that **does not come from human labels** is a new trend called “self-supervised learning”
- BERT and GPT were the first popular example of this, in the context of NLP
- Now common practice in a variety of domains

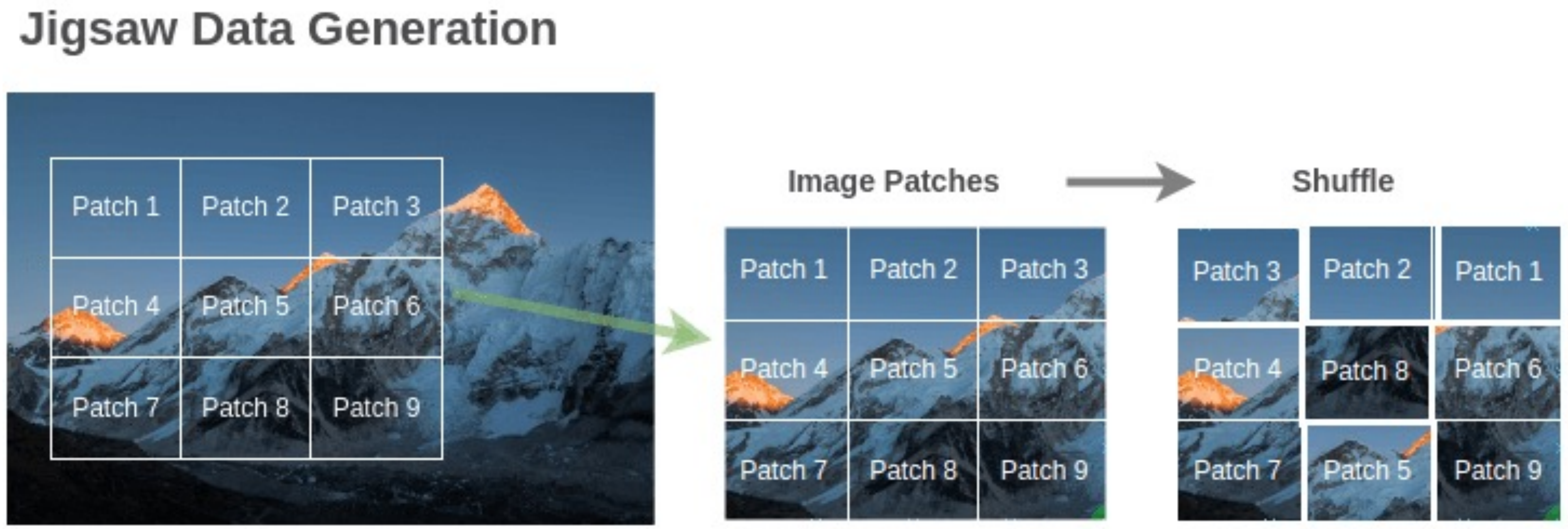
Self-Supervised Learning for Images: Colorization



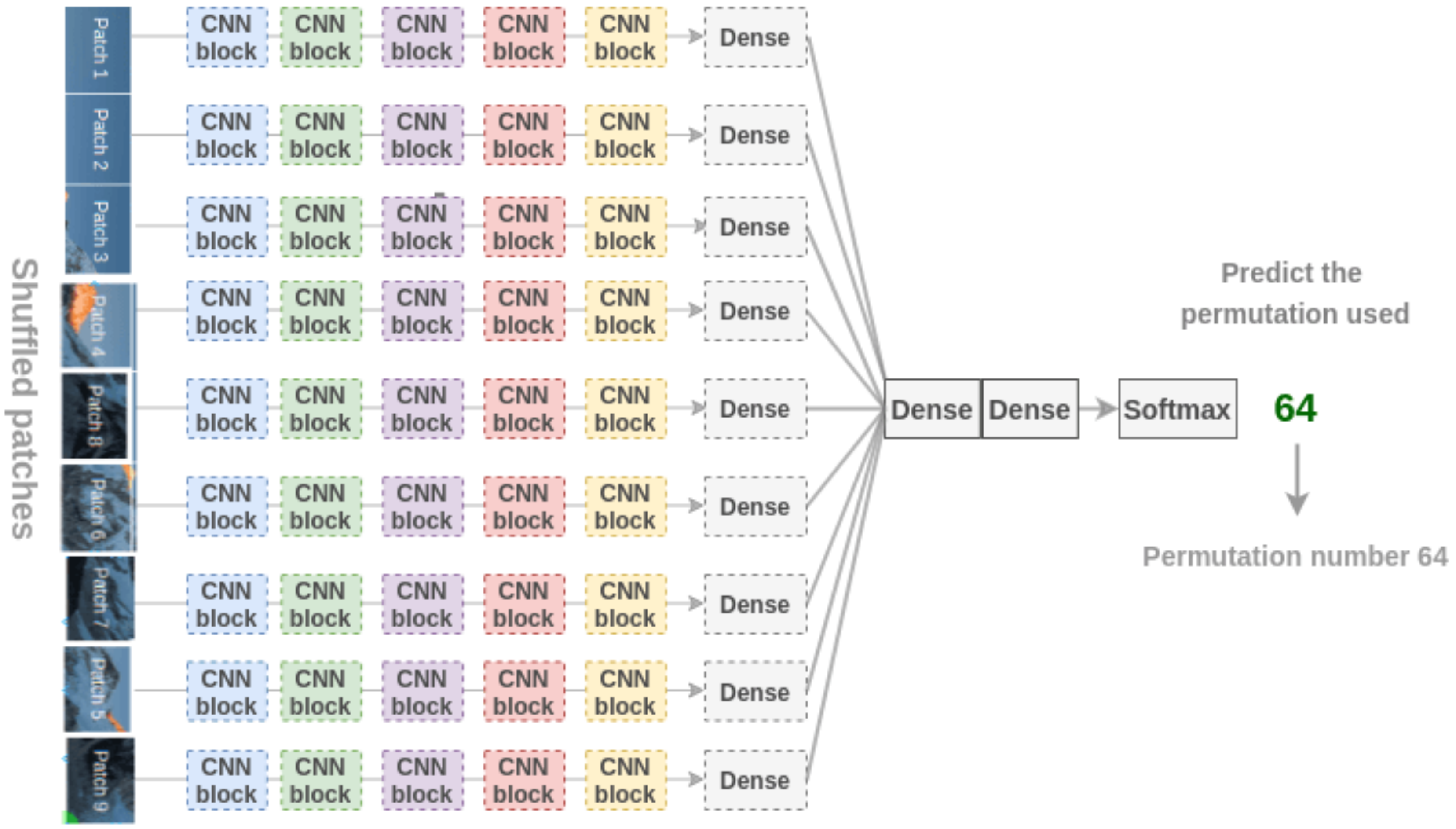
Self-Supervised Learning for Images: Inpainting



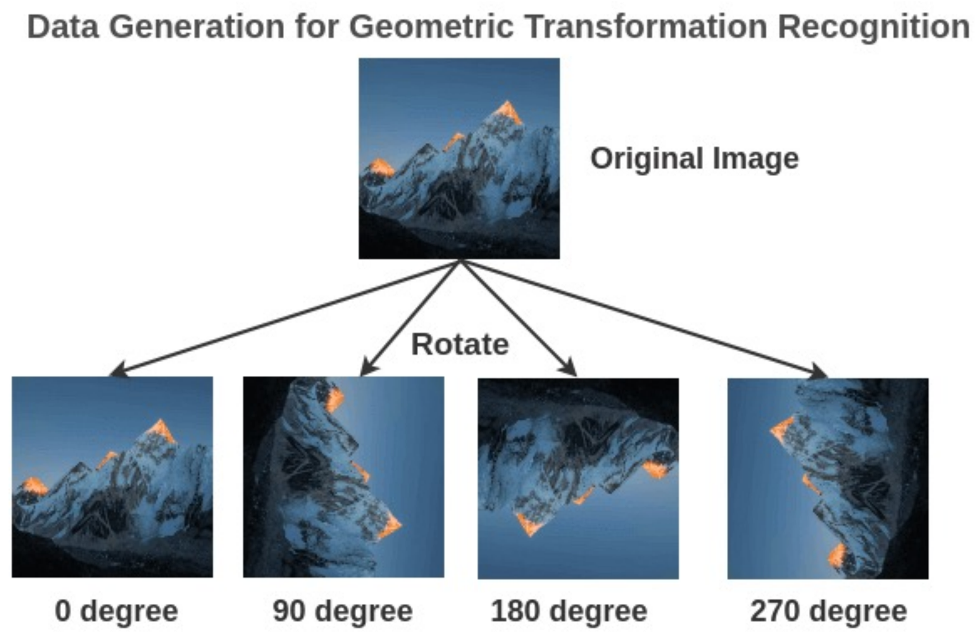
Self-Supervised Learning for Images: Jigsaw

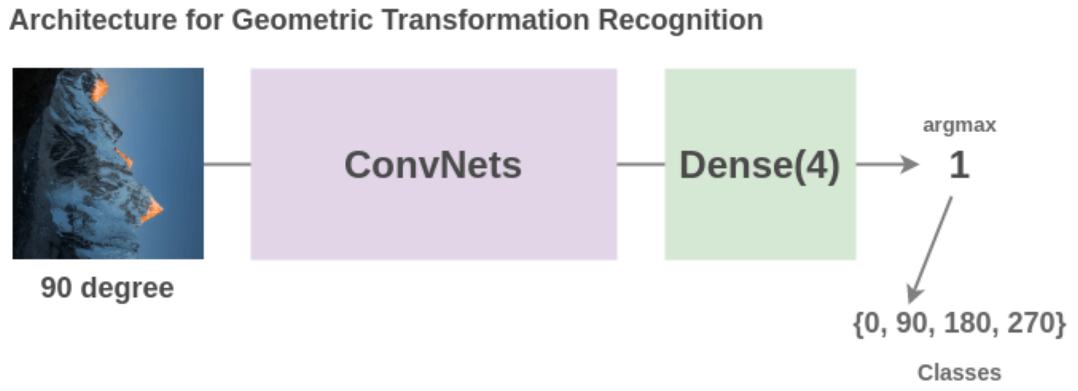


Self-Supervised Learning for Images: Jigsaw

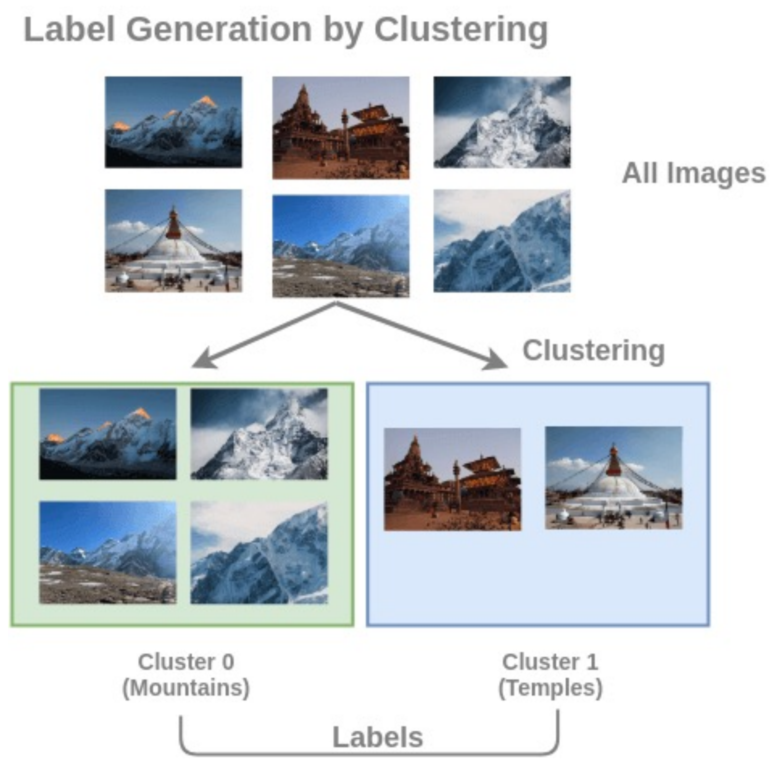


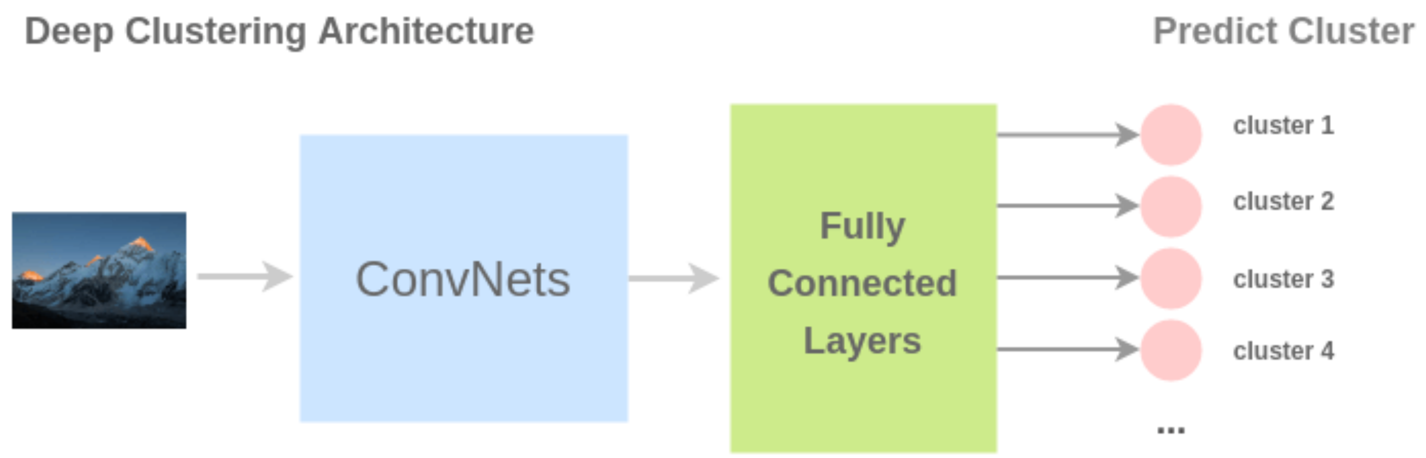
Self-Supervised Learning for Images: Geometric Transformations



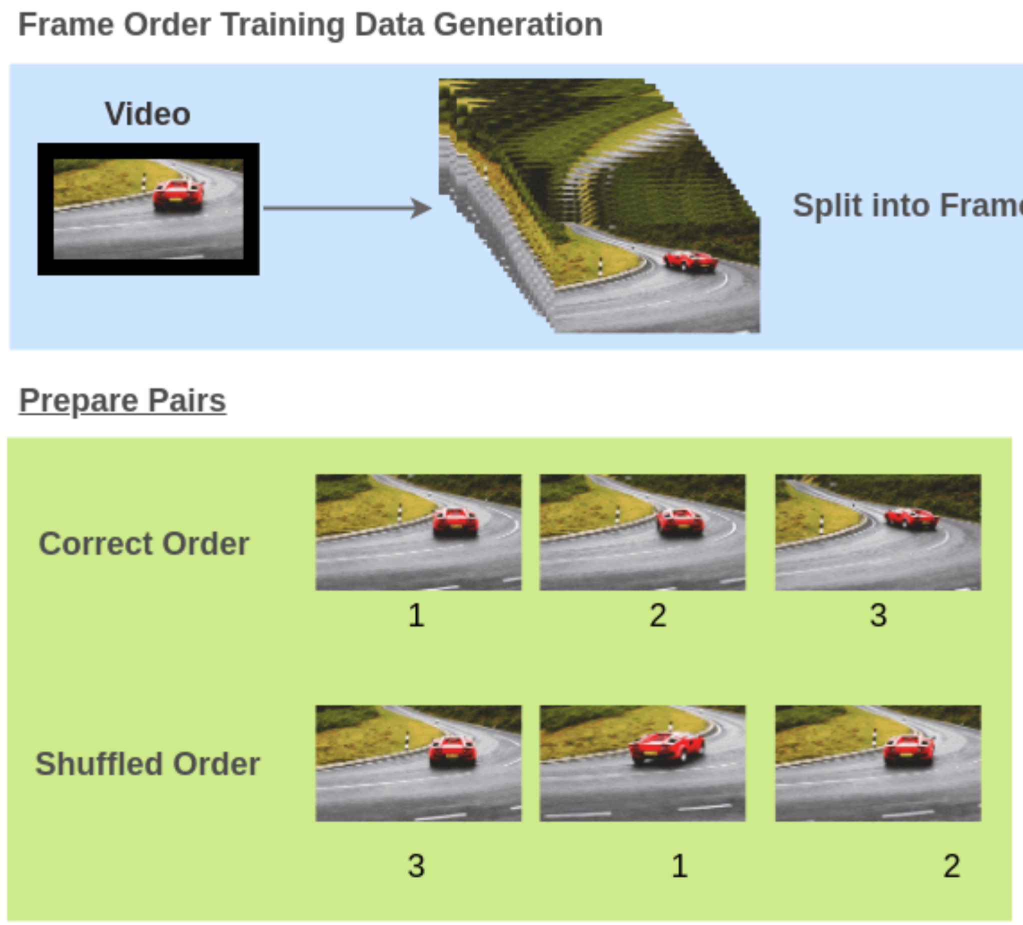


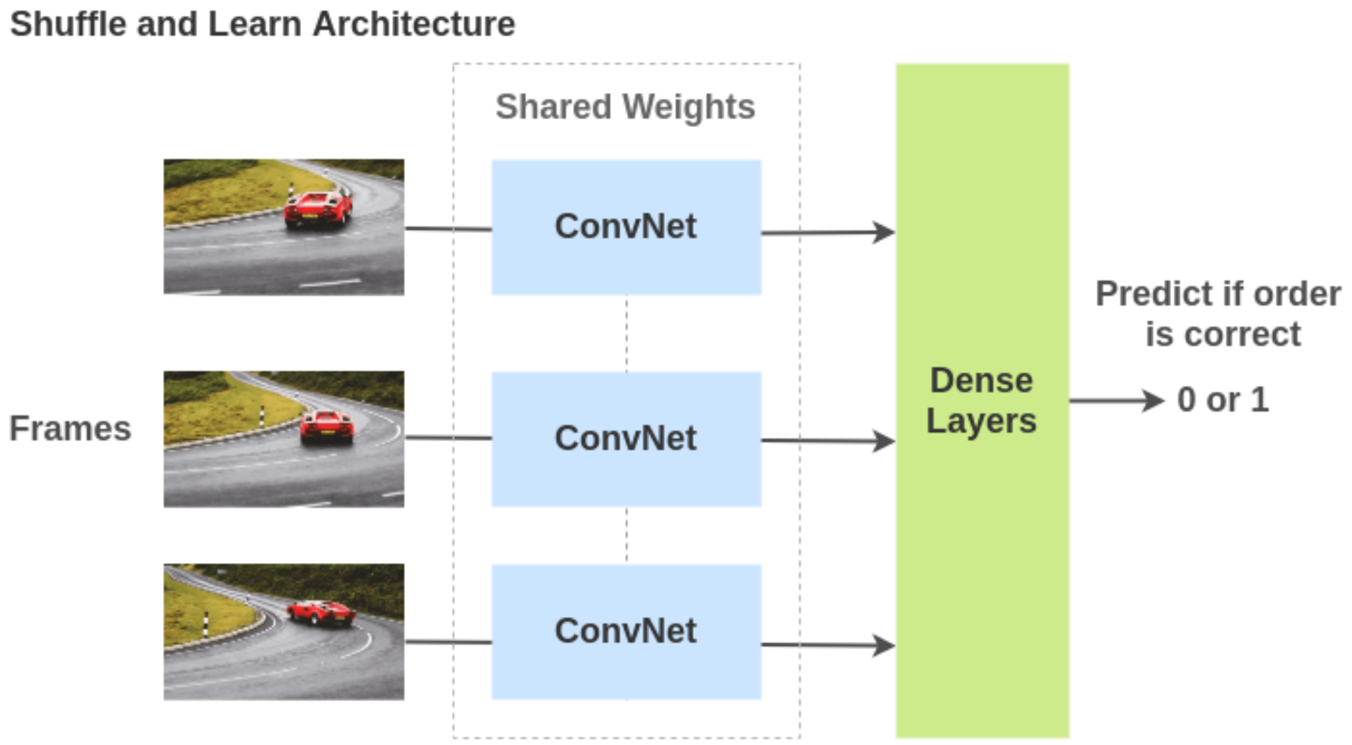
Self-Supervised Learning for Images: Image Clustering



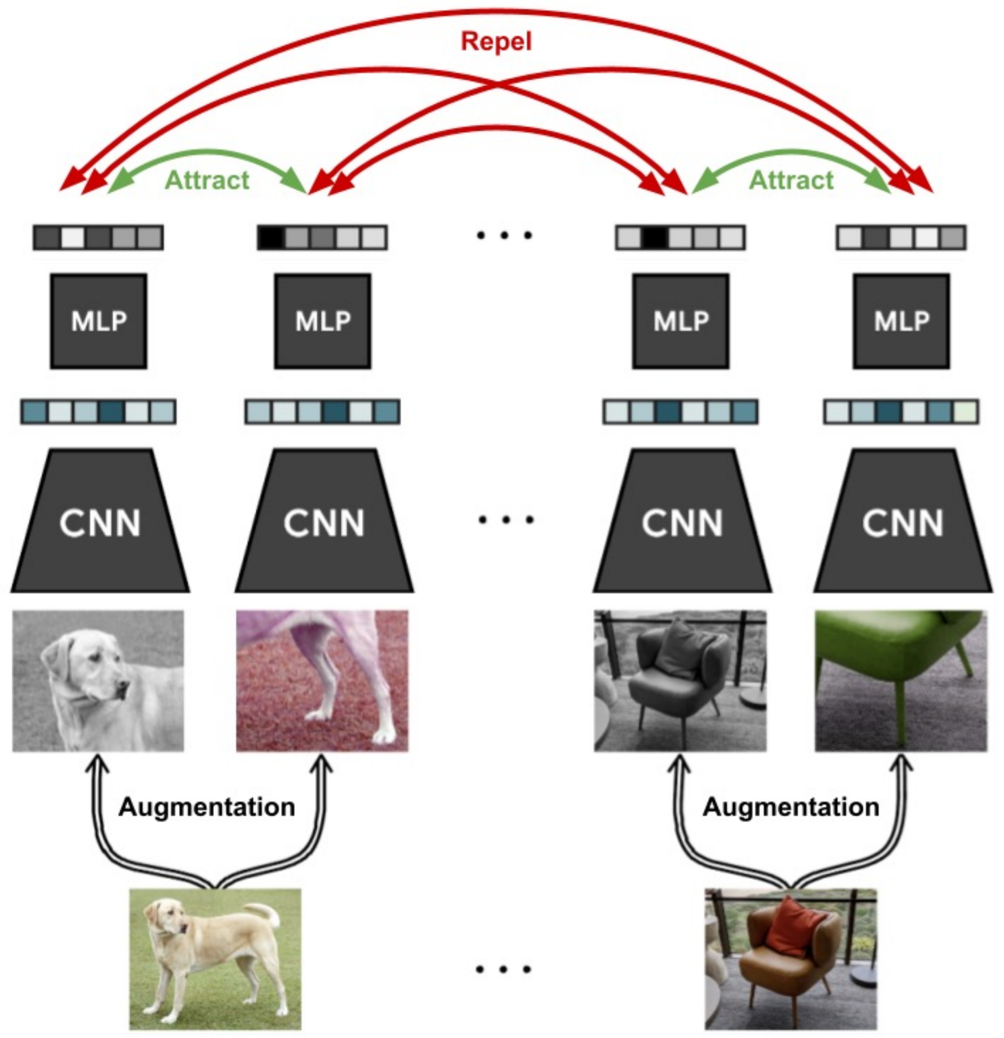


Self-Supervised Learning for Videos: Frame Ordering





Contrastive Self-Supervised Learning



Some Big Takeaways

- There are countless ways to design a deep neural network
  - The popular ones are often the ones that happened to be trained with lots of computational power
  - So don’t memorize all of the architectures we learned about – just the big ideas and core building blocks (which are reused across network architectures)
- There are a small set of core deep learning building blocks to remember
  - Dense layers, Convolutional+Pooling layers, Recurrent layers (incuding LSTMs), Attention layers, and variations/hyperparameters of all of these (including activation functions and which to use when)
- Feature representation matters
  - This is why we have transfer learning, self-supervised pre-training, etc.
- The neural network will learn what you tell it to learn via the loss function (not specific to deep learning)