Large Language Models

CS6960 MultiModal LLM Agents

Kenneth Marino

Slide credits: Ana Marasović, Alane Suhr, Dan Klein, Jay Alammar, Dan Jurafsky, and others

Quick Announcements

- Add/Drop Course
  - Class is pretty much at capacity
  - If you already know you will drop, please do that ASAP so people can get in off the waiting list
- HW0 due this Friday
  - It's really short and easy
  - Basically just asking you to read the intro unit to the HF Agents tutorial and write some boilerplate code
  - See the new instructions posted for this
- Papers List
  - Posted in Piazza
  - Grading rubric also up in assignments
  - Will open up people to sign up on Thursday during class

Recommended Readings

- NLP Courses and Tutorials
  - Utah NLP Course: https://utah-cs6340-nlp.notion.site/
  - Stanford NLP Course: https://web.stanford.edu/class/cs224n/
  - HuggingFace LLM Tutorial: https://huggingface.co/learn/llm-course/en/chapter1/1
- More resources
  - Nathan Lambert RLHF Intro: https://rlhfbook.com/c/14-reasoning#ref-wang2025ragen
  - Transformer paper: http://arxiv.org/abs/1706.03762
  - Annotated transformer: https://nlp.seas.harvard.edu/2018/04/03/attention.html

Warning (again)

- Again I can’t possibly cover everything about LLMs in a single lecture
- If you haven’t taken an NLP course, you probably will want to do some of the recommended readings/tutorials
- If you have taken e.g. Ana Marasović’s course, then this will be mostly review

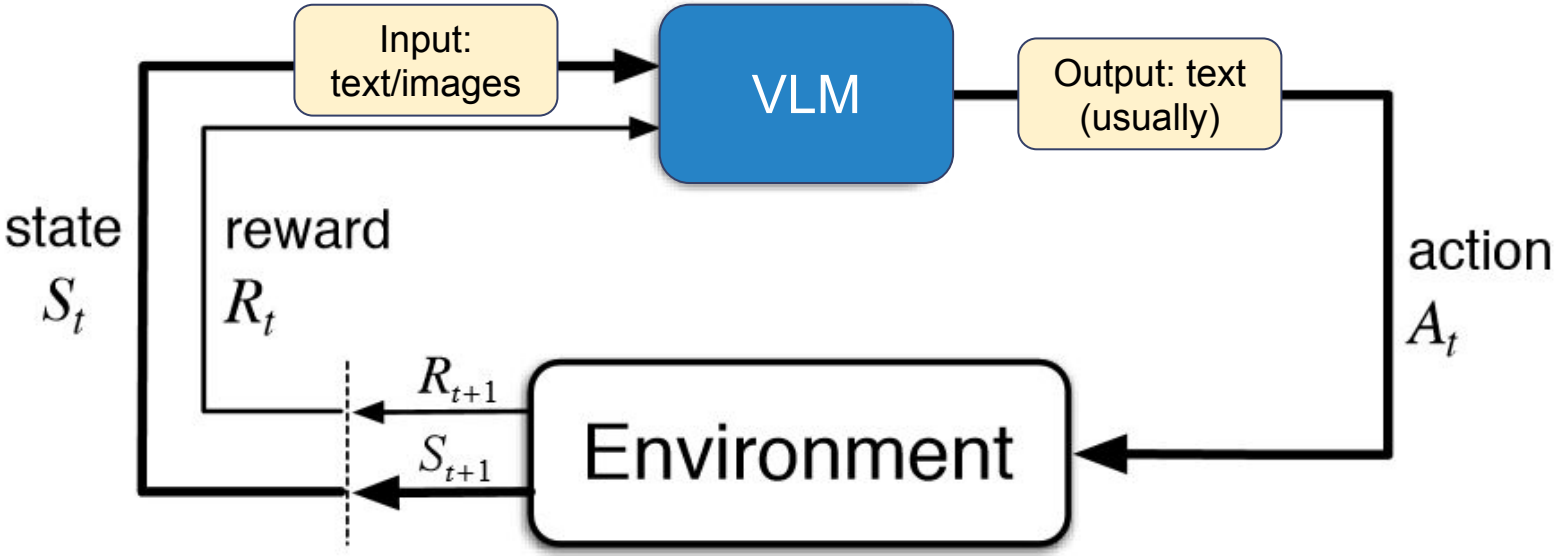
Any Questions

Recall: What are Agents?

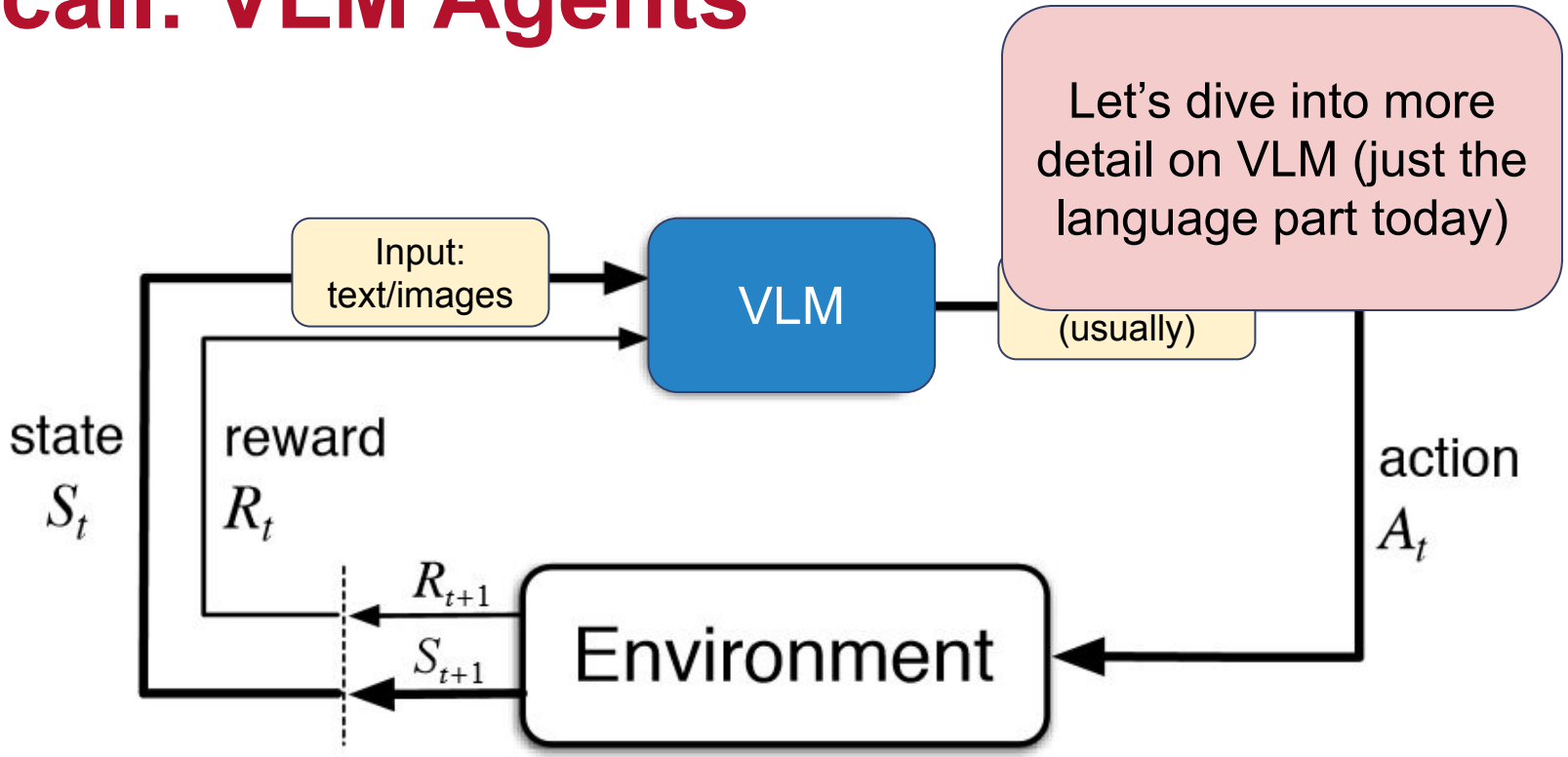
- VLM Agents

What Even is an Agent?

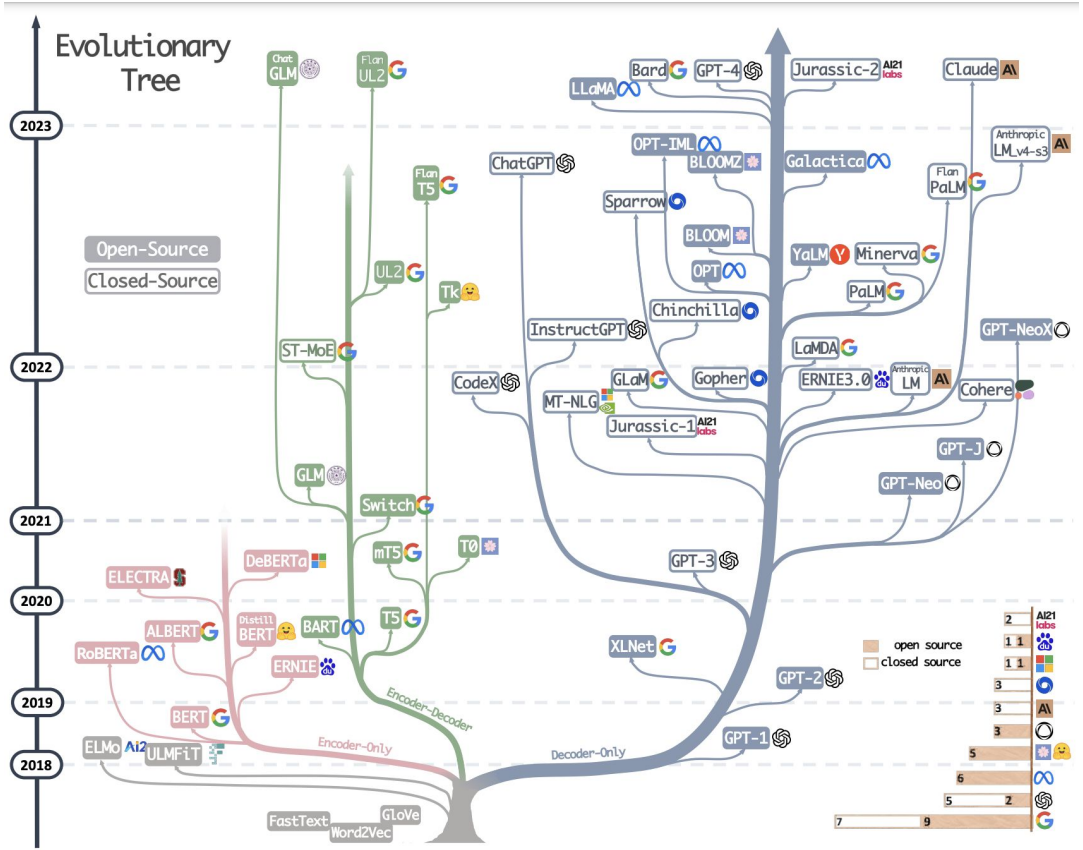
Recall: VLM Agents



Recall: VLM Agents



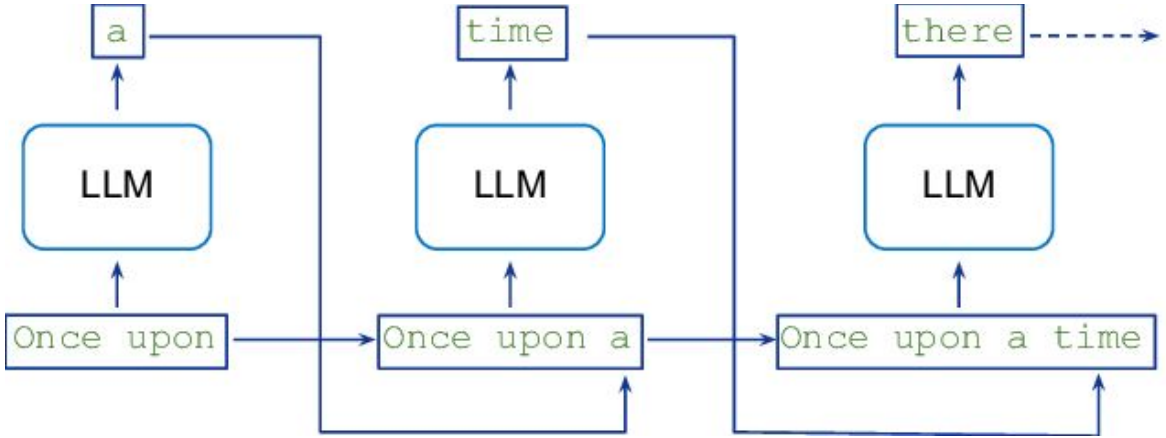
Lots of Kinds of LLMs



[Yang et al., 2023]

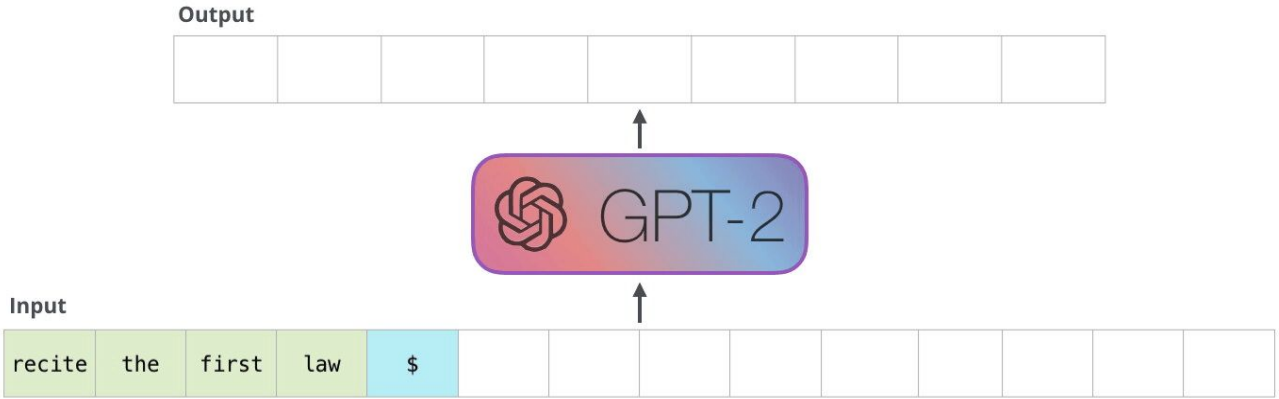
Autoregressive LLMs

- Conditional Generation
  - Generating text conditioned on previous text!
  - P(“a” | “Once upon”)
  - P(“time” | “Once upon a”)
  - P(“there” | “Once upon a time”)



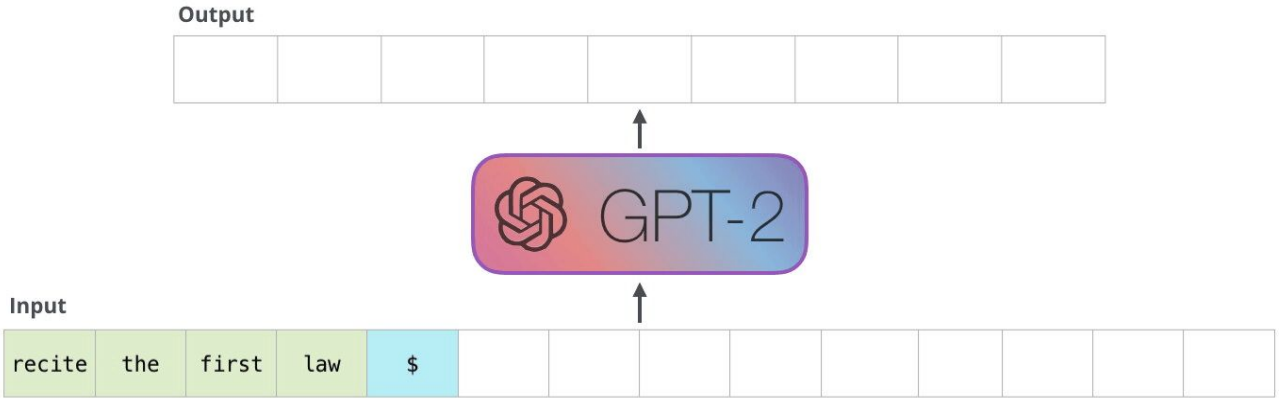
Autoregressive LLMs

- LLM Inference
  - Given a sequence of input tokens, an LLM generates a single token at each step
  - The generated token will be appended after the previous input tokens and fed into the LLM again to generate the token for the next step



Autoregressive LLMs

- LLM Inference
  - Assigns probabilities to sequences of words
  - Generate text by sampling possible next words
  - **Are trained by learning to guess the next word**



Transformer (the first one)

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

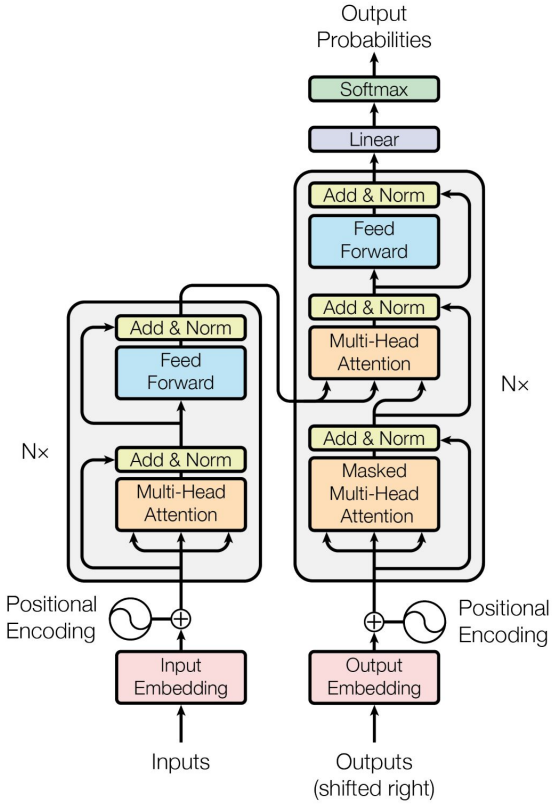
Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

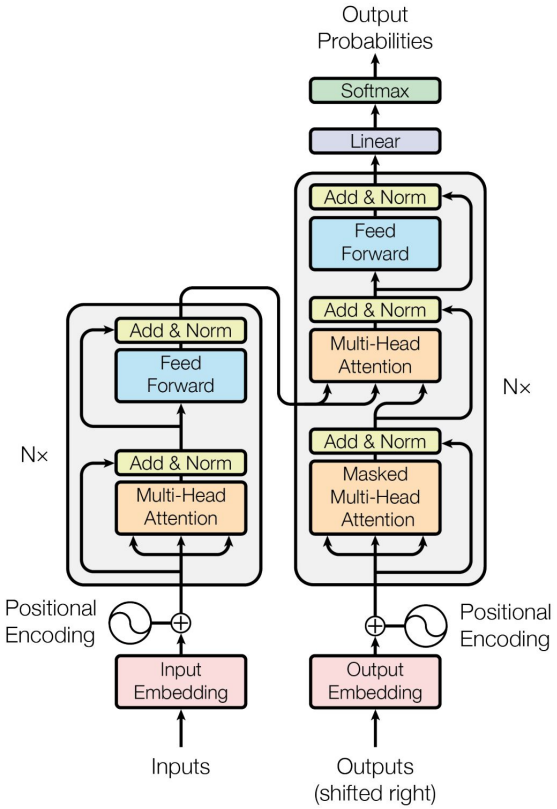
Illia Polosukhin* ‡ illia.polosukhin@gmail.com



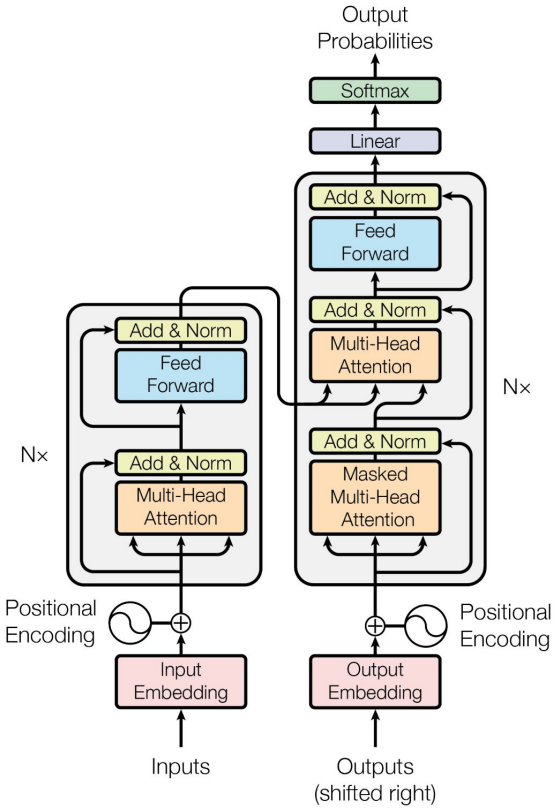
Motivation for Transformers

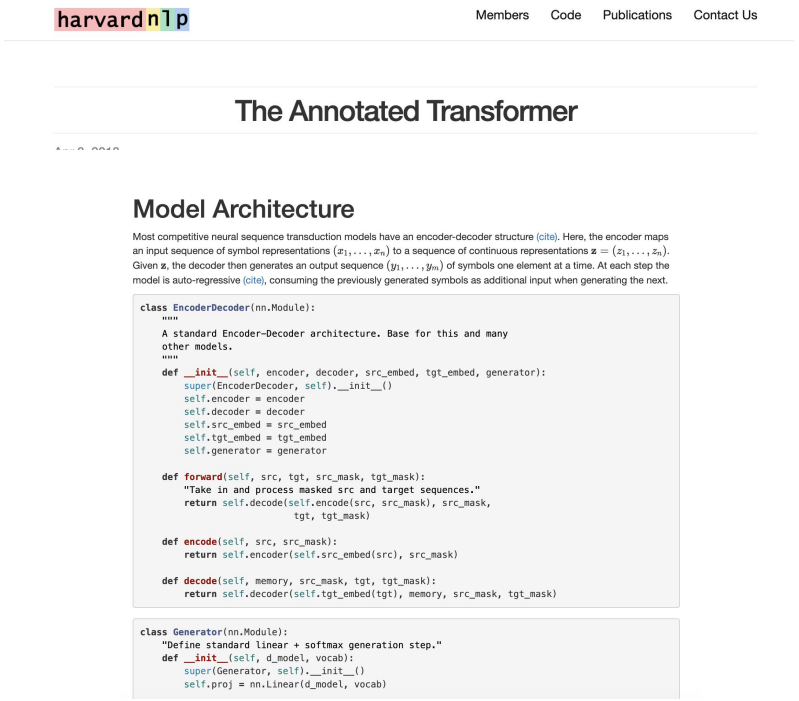
- High-level insights
  - Originally proposed to handle long sequences (especially language)
  - Compared to recurrent models, attention model does not suffer the gradient vanishing or explosion problem
  - Enabled the current renaissance of NLP

Original Transformer Architecture



Original Transformer Architecture





harvardnlp Members Code Publications Contact Us

The Annotated Transformer

Model Architecture

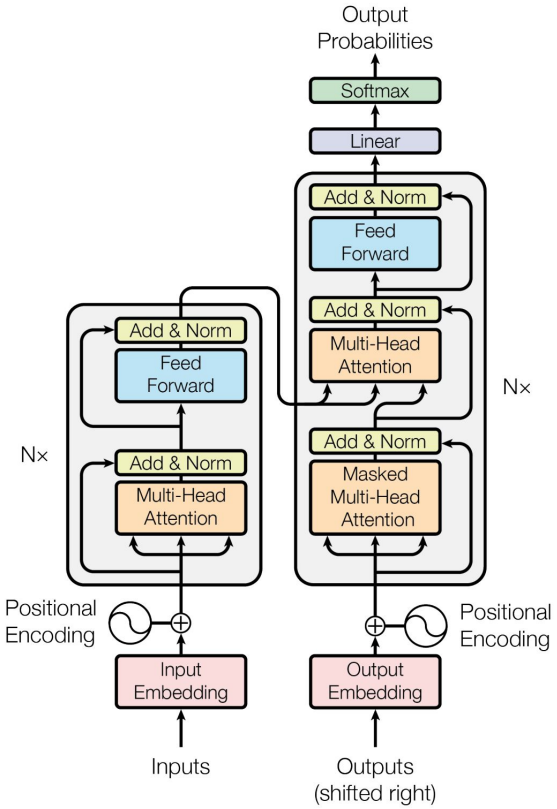
Most competitive neural sequence transduction models have an encoder-decoder structure (cite). Here, the encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$. Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, \ldots, y_m)$ of symbols one element at a time. At each step the model is auto-regressive (cite), consuming the previously generated symbols as additional input when generating the next.

```
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask,
                           tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)
```

For thorough walkthrough, see
https://nlp.seas.harvard.edu/2018/04/03/attention.html

Original Transformer Architecture



Parts of Transformers:

- Encoder/Decoder
- Attention
- Self-attention
- Multi-headed attention
- Word Embeddings
- Positional Encoding
- Encoder blocks
- Decoder blocks

Encoder/Decoder

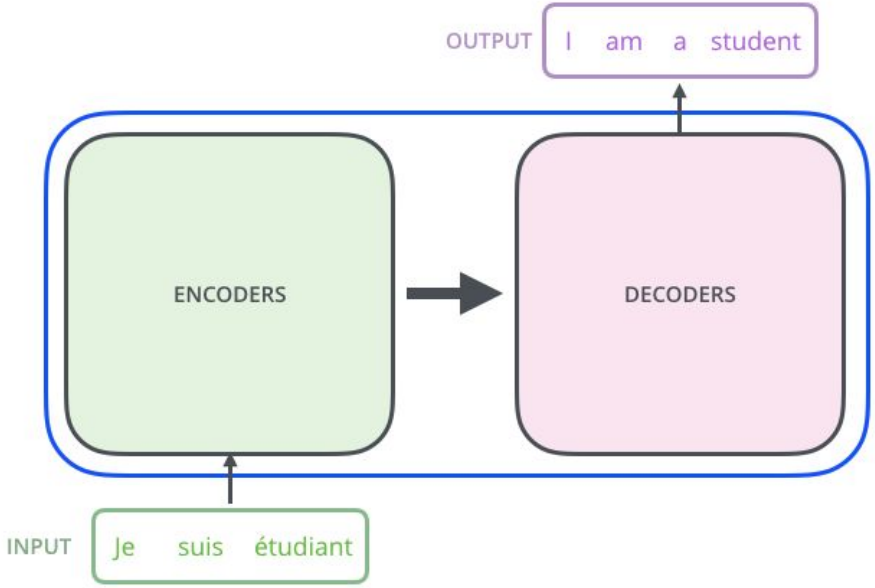
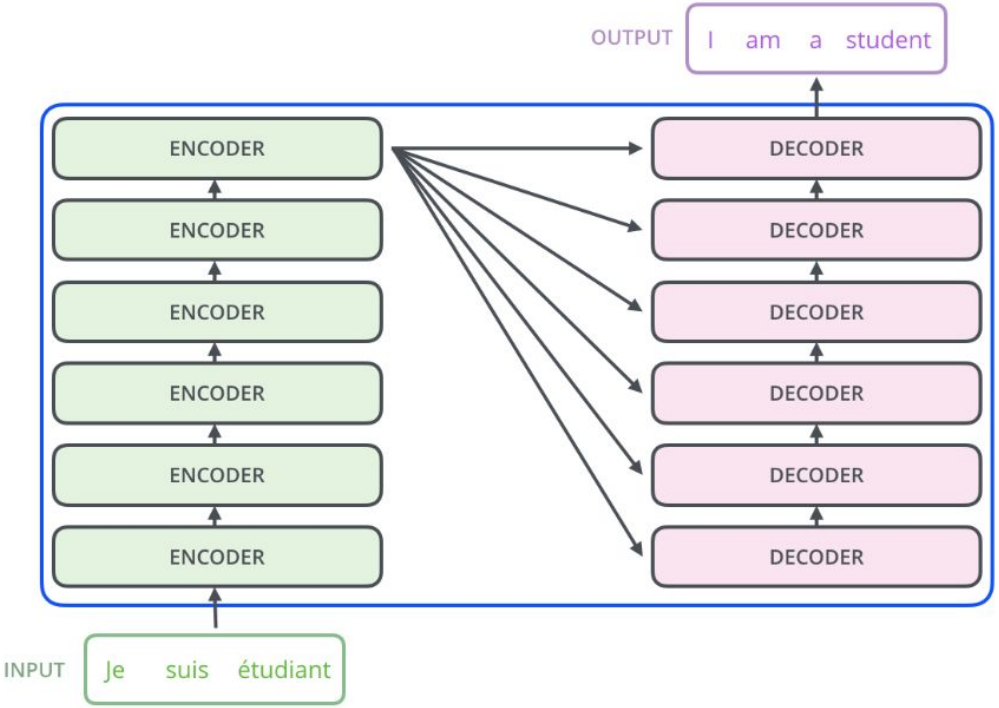


Figure: Jay Alammar

Repeated Blocks

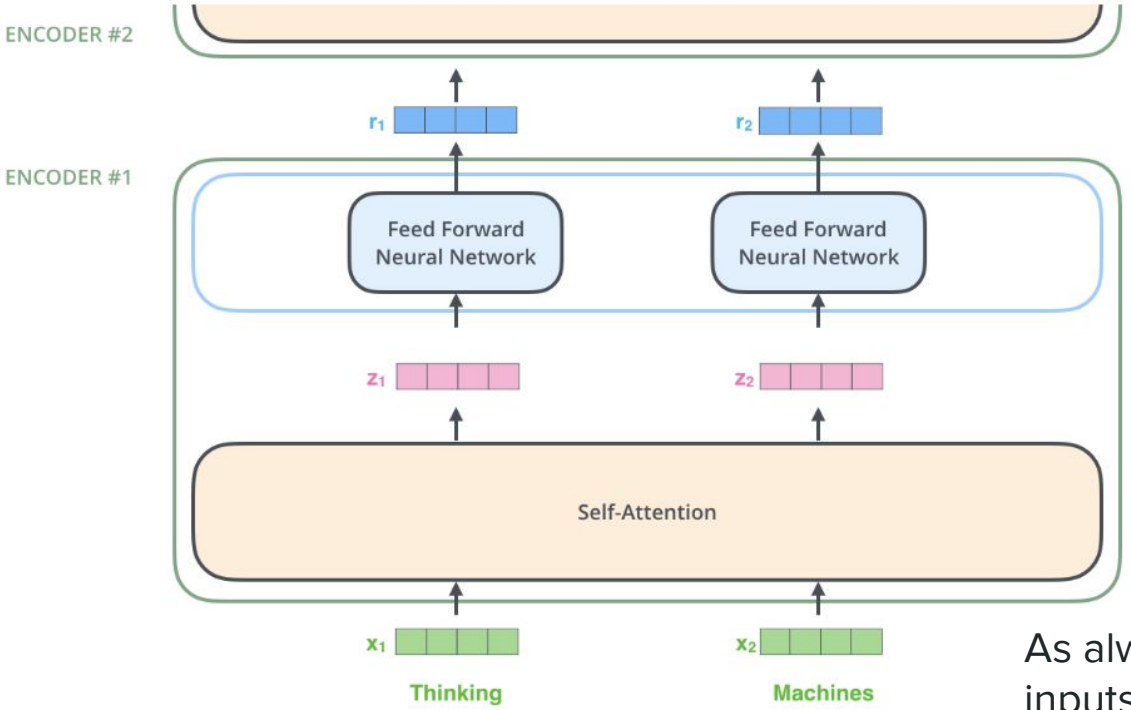
A stack of encoder blocks



A stack of decoder blocks

Figure: Jay Alammar

Each encoder block consists of multi-headed self-attention & FFNN



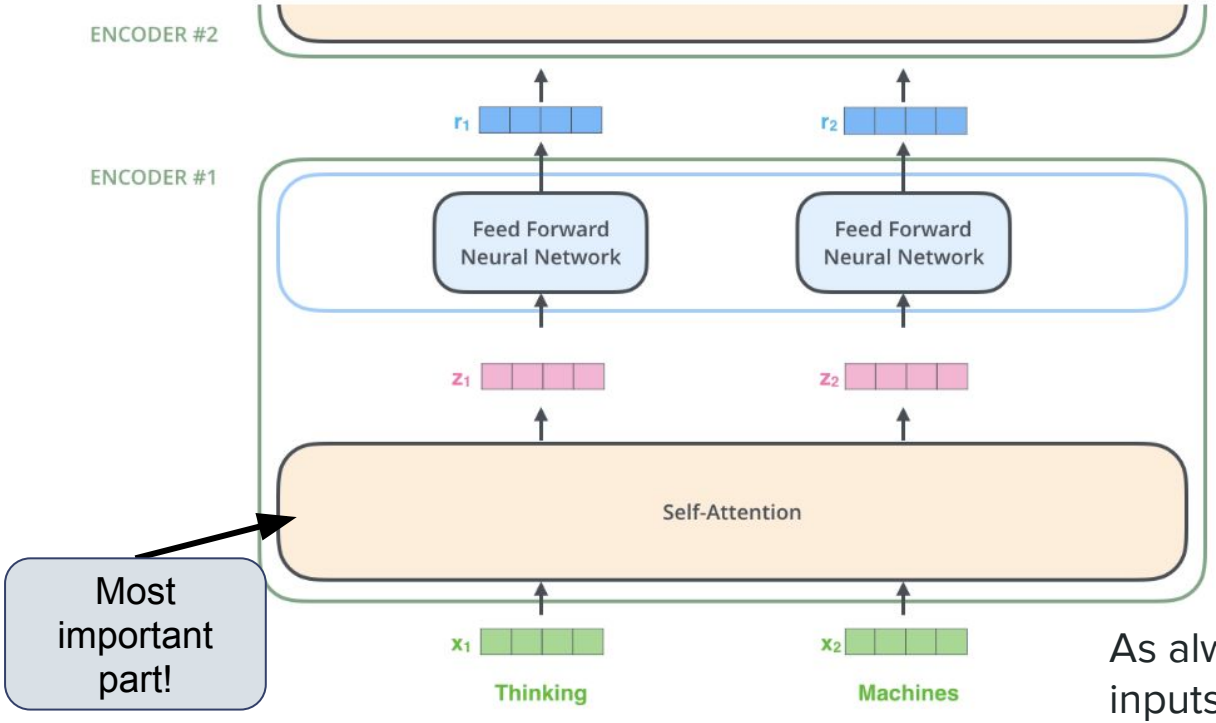
As always, deeper layers get outputs of the previous layers as inputs

Input to each encoder block has the same size as original token embeddings

As always, in the first layer inputs are static token emeddings

Figure: Jay Alammar

Each encoder block consists of multi-headed self-attention & FFNN



As always, deeper layers get outputs of the previous layers as inputs

Input to each encoder block has the same size as original token embeddings

As always, in the first layer inputs are static token emeddings

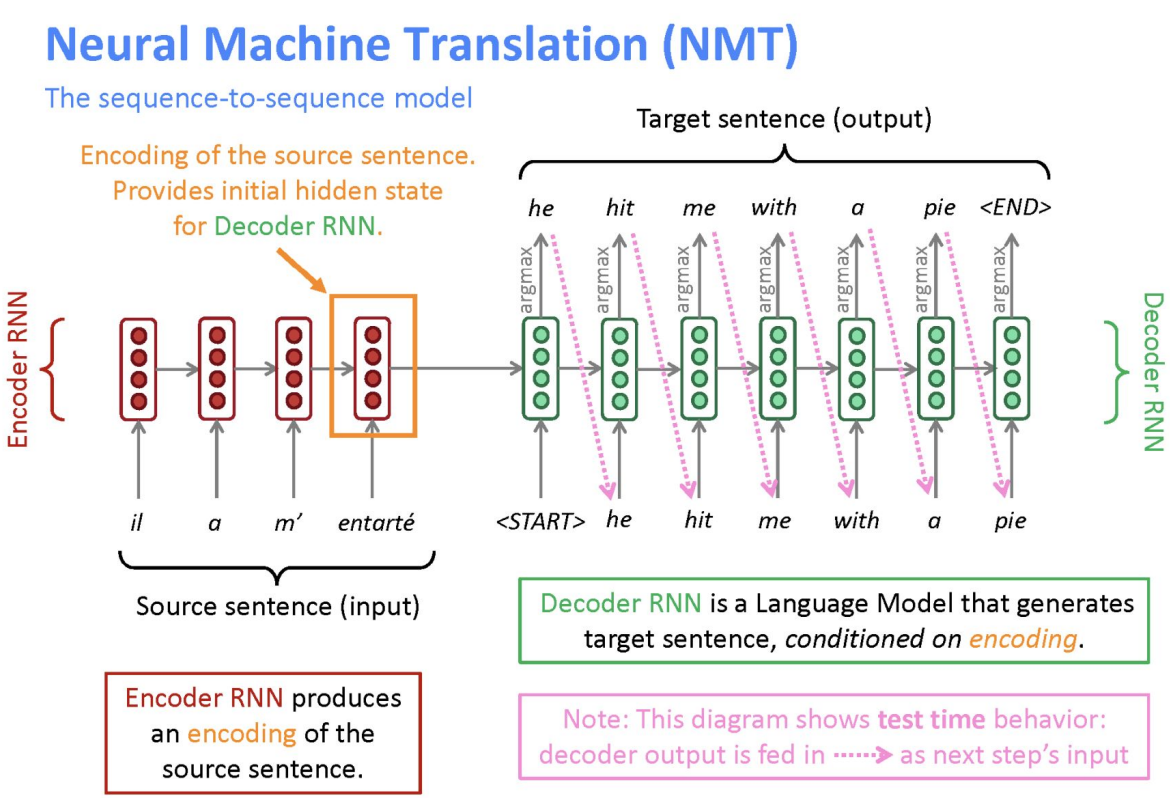
Figure: Jay Alammar

Why Attention

Turned out to be a bottleneck

Needs to encode all info in the source

What if the source sequence is long?



Source: Alane Suhr & Dan Klein

Attention

Attention provides a solution to the bottleneck problem

Proposed in Bahdanau et al. (2015): Neural Machine Translation by Jointly Learning to Align and Translate

Core idea: On each step of the decoder, use **direct connection to the encoder** to **focus on a particular part** of the source sequence

Attention

- Translate “The animal didn't cross the street because it was too tired”

Source: https://jalammar.github.io/illustrated-transformer/

Attention

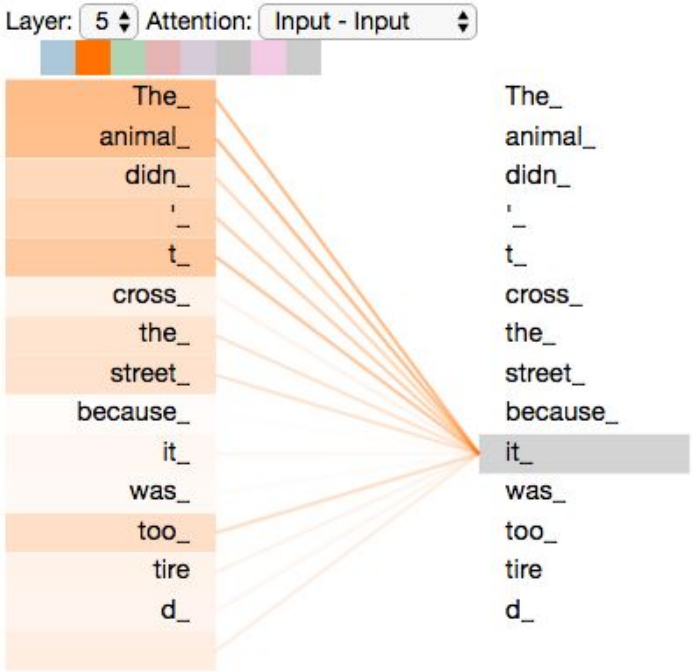
- Translate “The animal didn't cross the street because it was too tired”

What does the word “it” refer to?

Source: https://jalammar.github.io/illustrated-transformer/

Attention

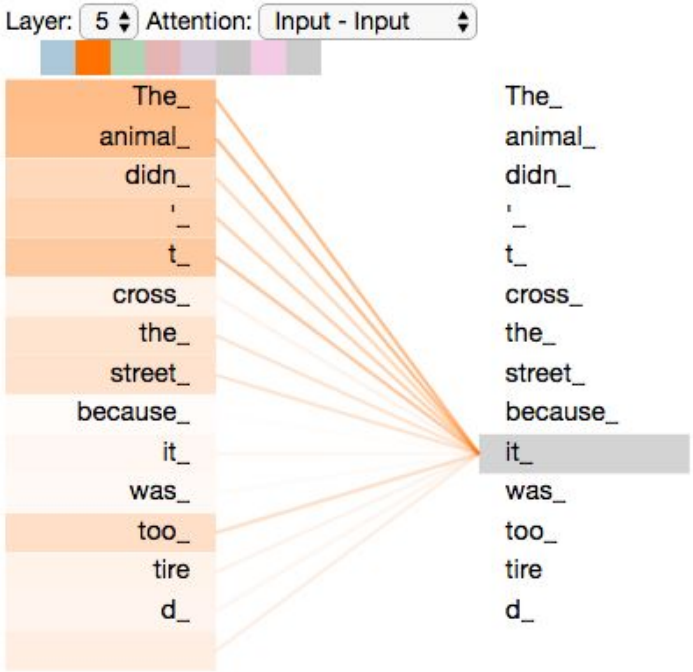
- Translate “The animal didn't cross the street because it was too tired”



Source: https://jalammar.github.io/illustrated-transformer/

Attention

- Translate “The animal didn't cross the street because it was too tired”



Idea: when we get to the word “it” we want to pay more “attention” to the most relevant prior words

Source: https://jalammar.github.io/illustrated-transformer/

Self-Attention

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

Source: https://jalammar.github.io/illustrated-transformer/

Self-Attention

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

Companies have spent literally billions of dollars because of this one equation

Source: https://jalammar.github.io/illustrated-transformer/

Self-Attention

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

What are the inputs Q, K, V for self-attention?

Source: https://jalammar.github.io/illustrated-transformer/

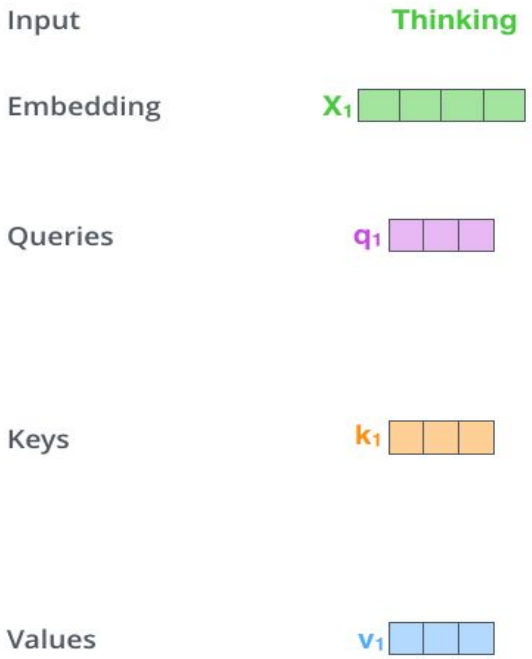
Self-attention – Query, Key, Value

- Create three vectors from the embedding of each word (query, key, value)

Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Query, Key, Value

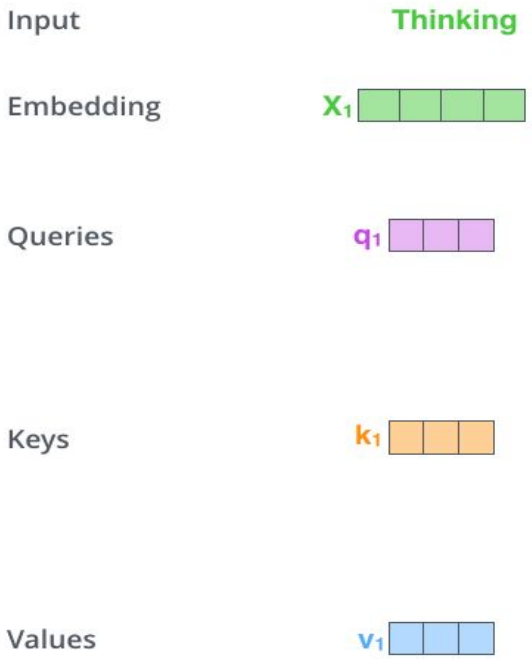
- Create three vectors from the embedding of each word (query, key, value)



Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Query, Key, Value

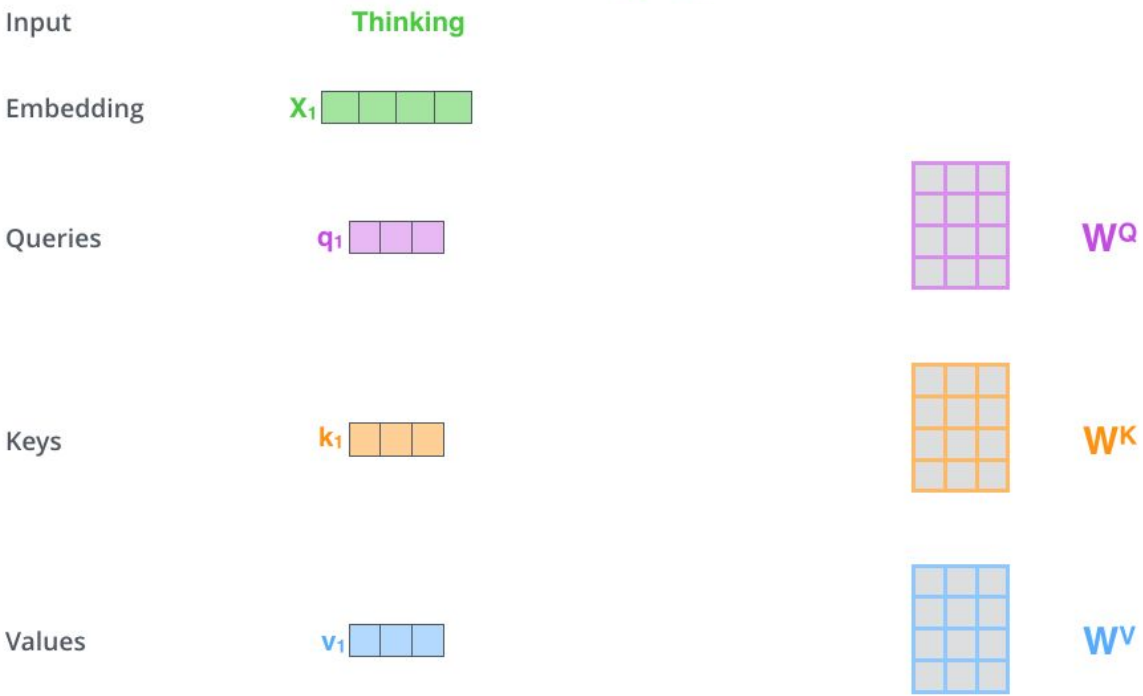
- Create three vectors from the embedding of each word (query, key, value)



Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Query, Key, Value

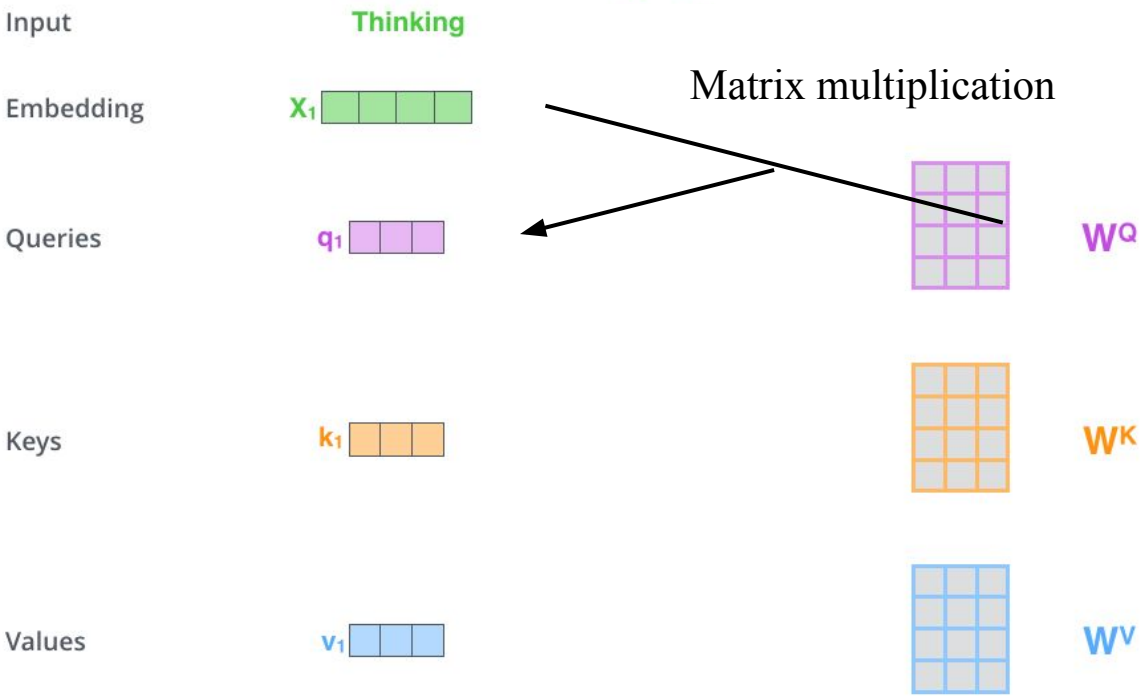
- Create three vectors from the embedding of each word (query, key, value)



Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Query, Key, Value

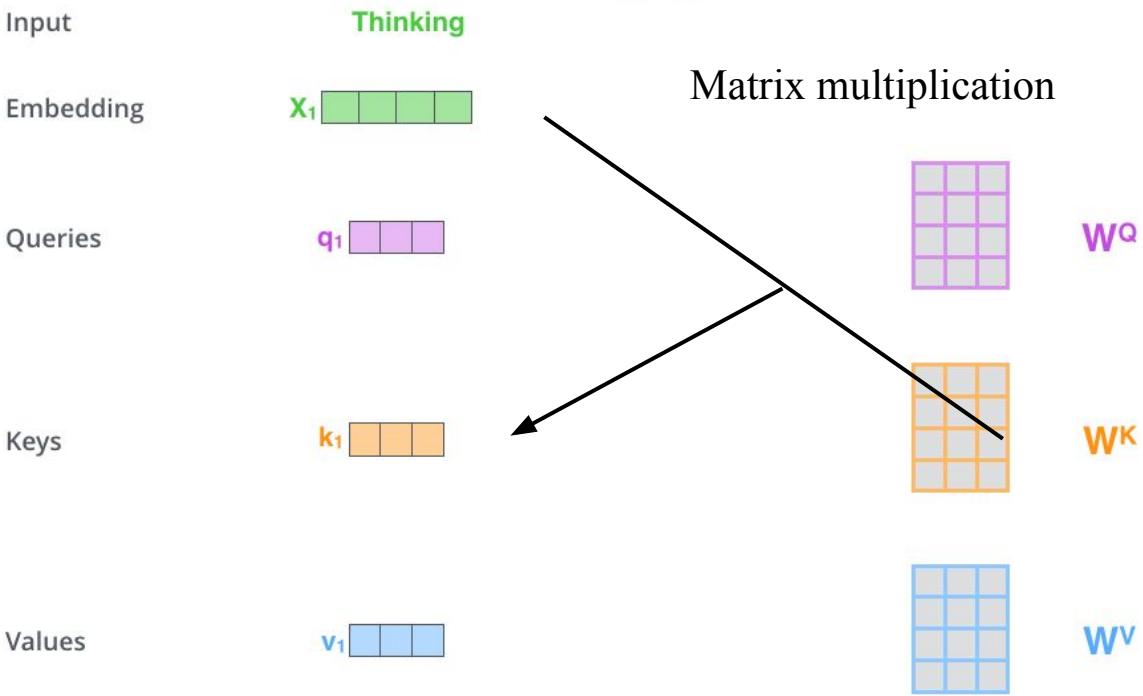
- Create three vectors from the embedding of each word (query, key, value)



Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Query, Key, Value

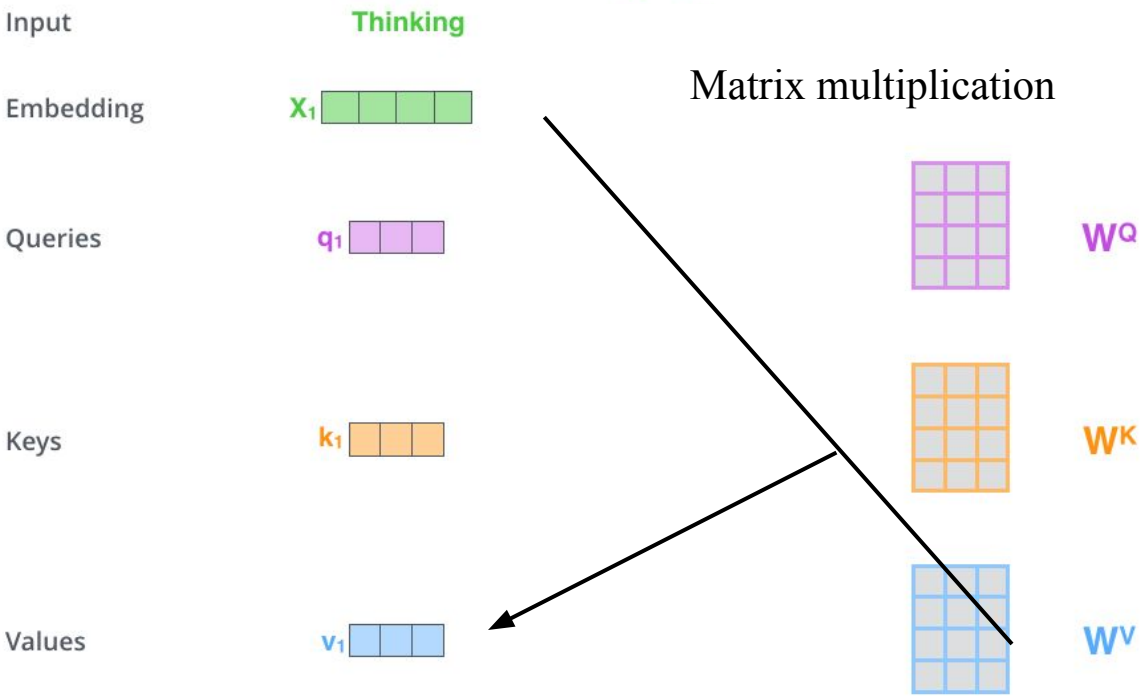
- Create three vectors from the embedding of each word (query, key, value)



Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Query, Key, Value

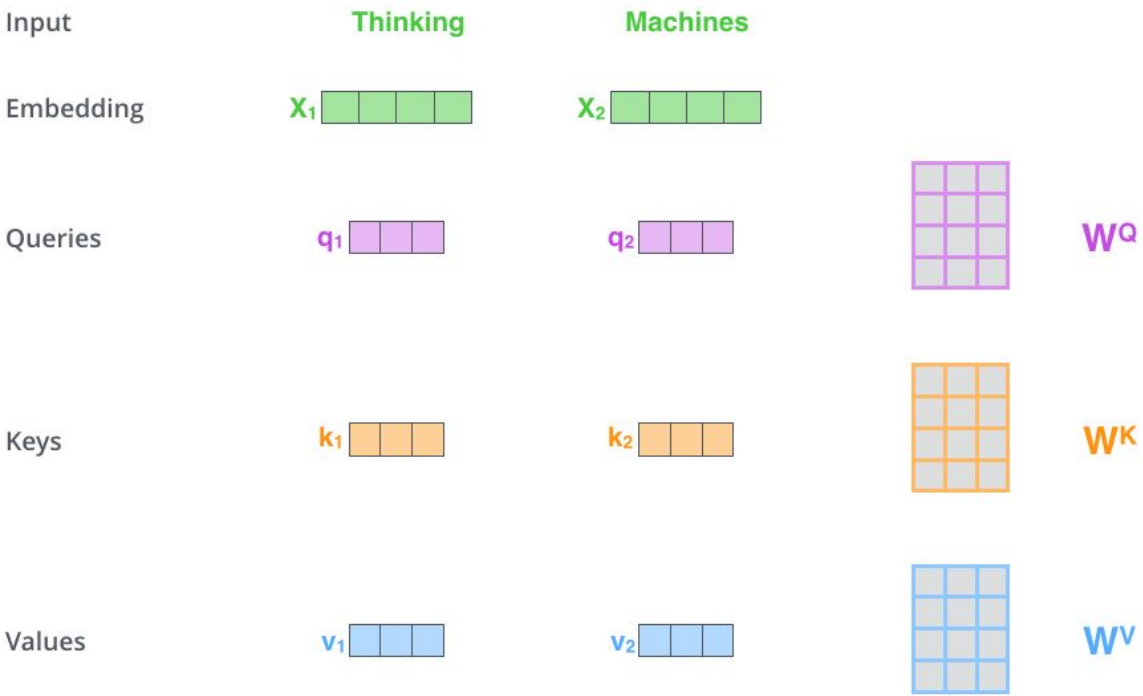
- Create three vectors from the embedding of each word (query, key, value)



Source: https://jalammar.github.io/illustrated-transformer/

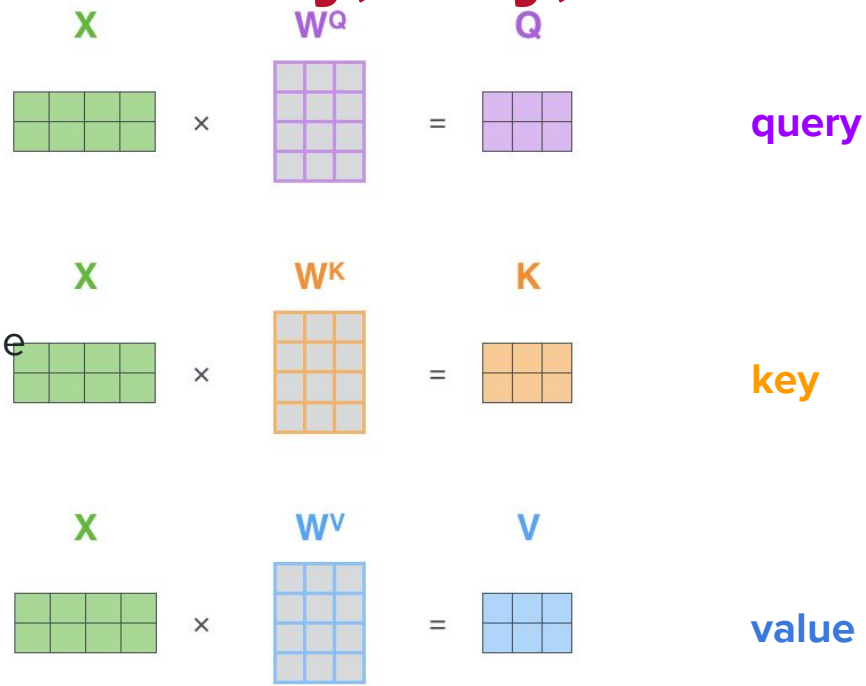
Self-attention – Query, Key, Value

- Create three vectors from the embedding of each word (query, key, value)



Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Query, Key, Value



X represents entire input sequence (here of the len 2)

Row 1 is the embedding of the 1st token in the input

Row 2 is the embedding of the 2nd token in the input

i-th row is the embedding of the i-th token in the input

W are weights we are learning

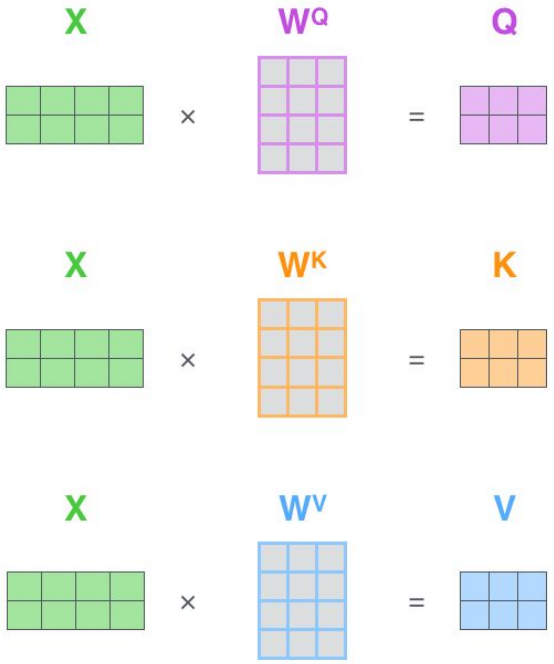
We multiply the input representation with the query, key, and value **weight matrices** to get query, key, and values

Think about them as different linear representations of the input that will be used to compute new input representations that depend on pairwise token importances

Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Matrix Calculation

- Create three vectors from the embedding of each word (query, key, value)

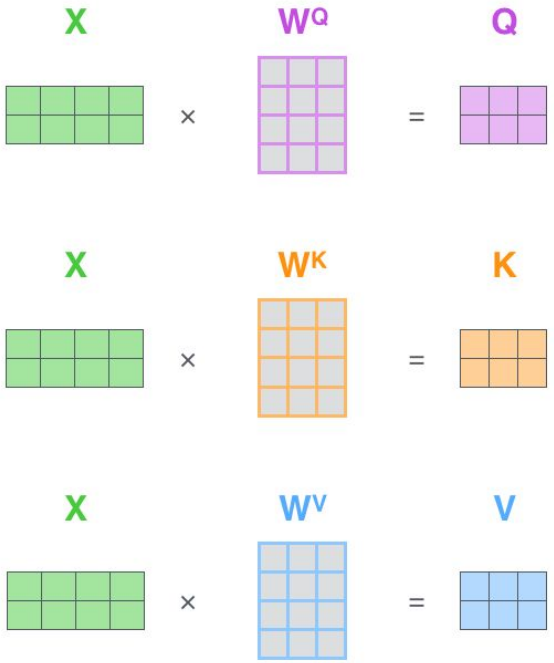


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

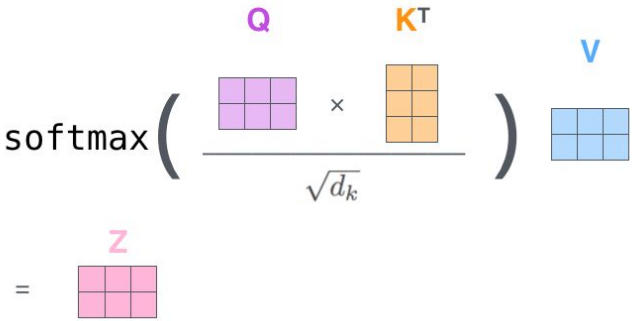
Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Matrix Calculation

- Create three vectors from the embedding of each word (query, key, value)



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



Produce Output

Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Matrix Calculation

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right)$$

let's visualize

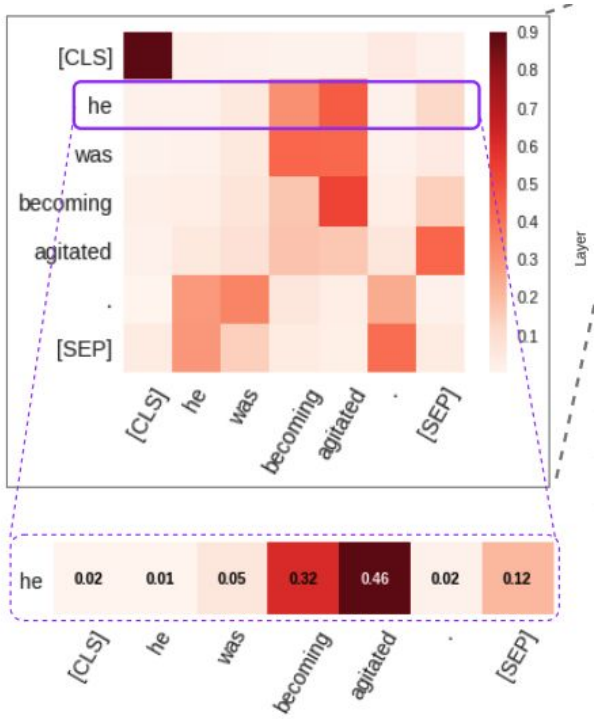
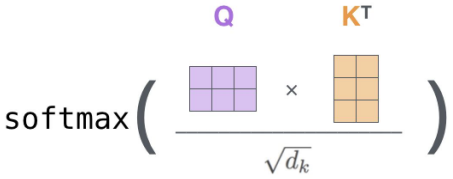


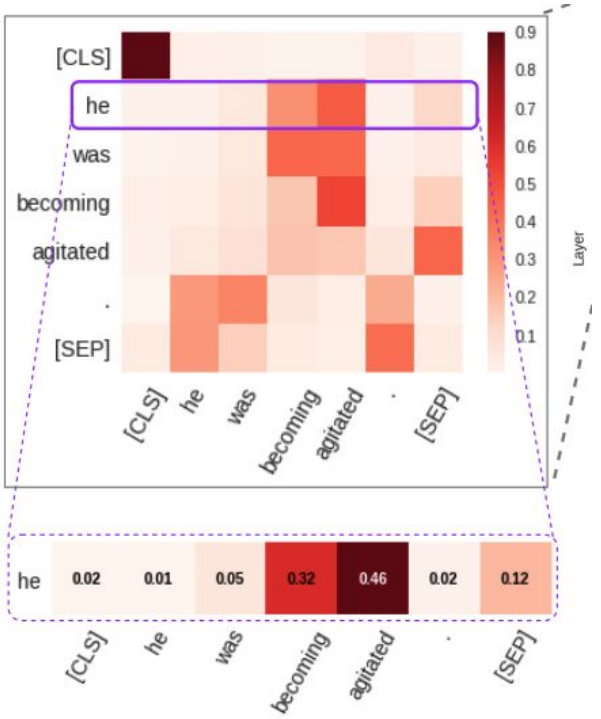
Figure on the right: [Kovaleva et al., 2019]

Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Matrix Calculation



let's visualize

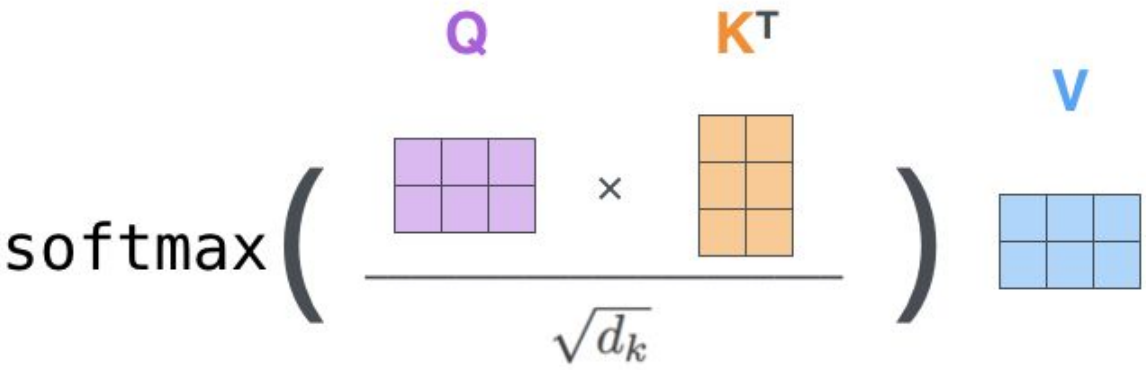


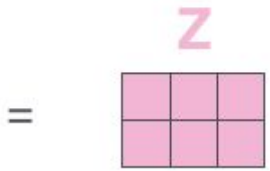
Essentially this is saying, which tokens in our sequence should we pay attention to

Figure on the right: [Kovaleva et al., 2019]

Source: https://jalammar.github.io/illustrated-transformer/

Self-attention – Matrix Calculation



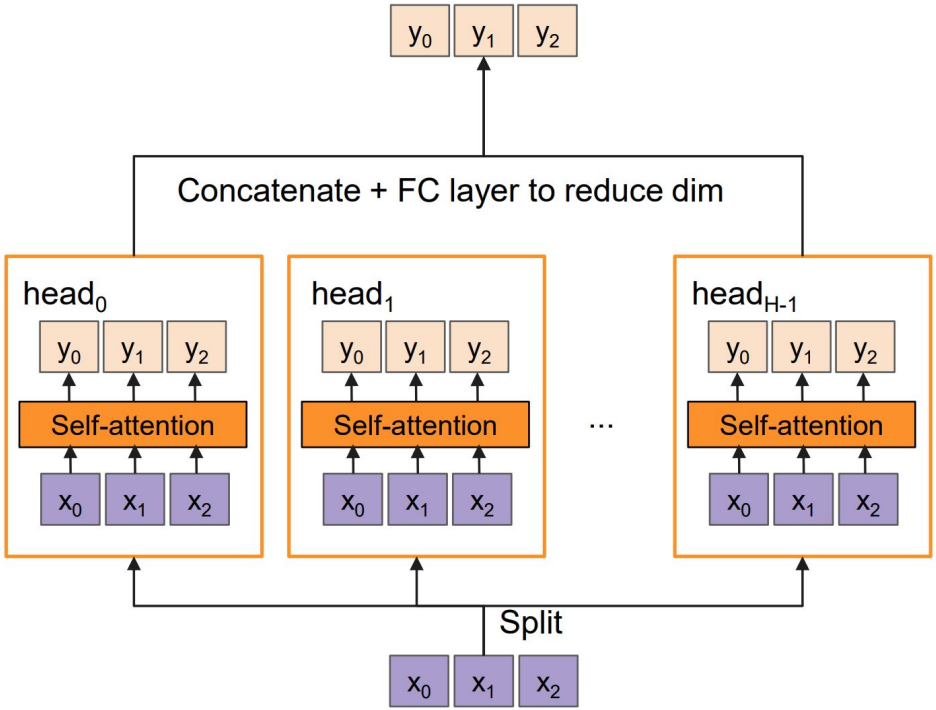


Each row is the new representation of the corresponding token in the input, now capturing this token's relationships with other tokens in the input

Source: https://jalammar.github.io/illustrated-transformer/

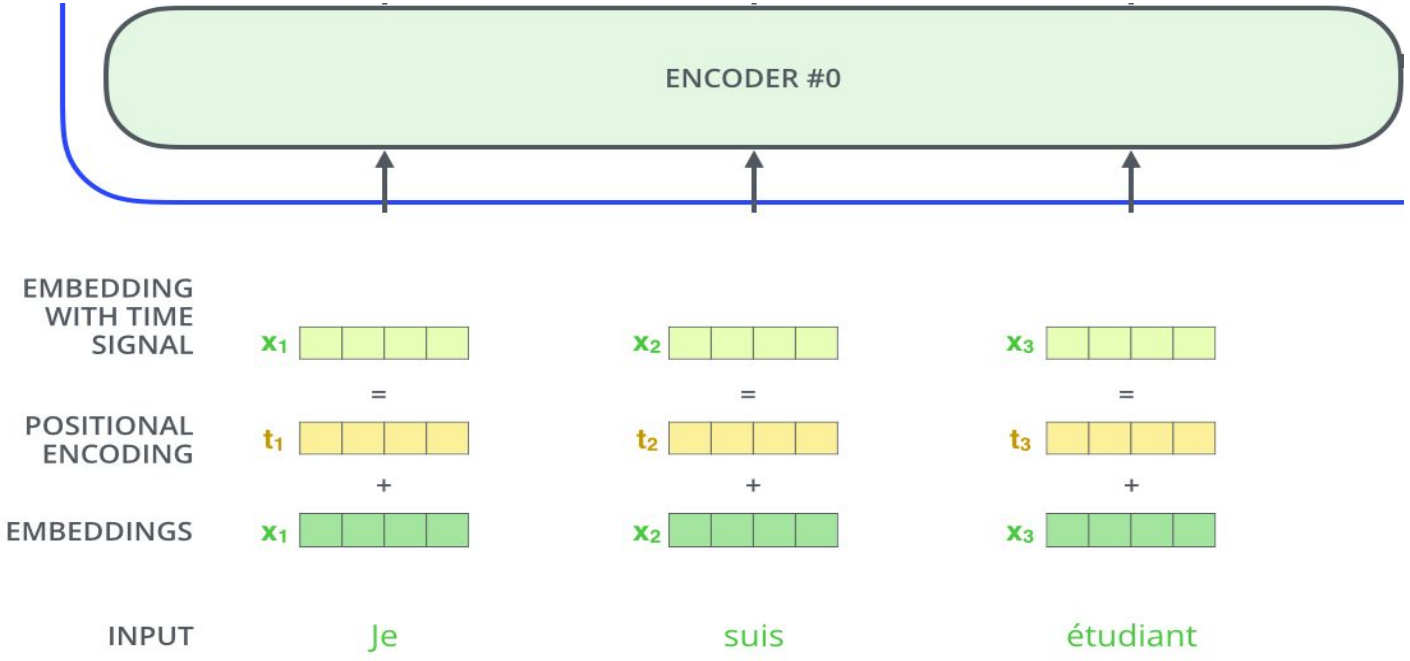
Multi-head self-attention layer

- Multiple self-attention "heads" in parallel
- We want to have multiple sets of queries/keys/values calculated in the layer. This is a similar idea to having multiple conv filters learned in a layer



Positional Encoding

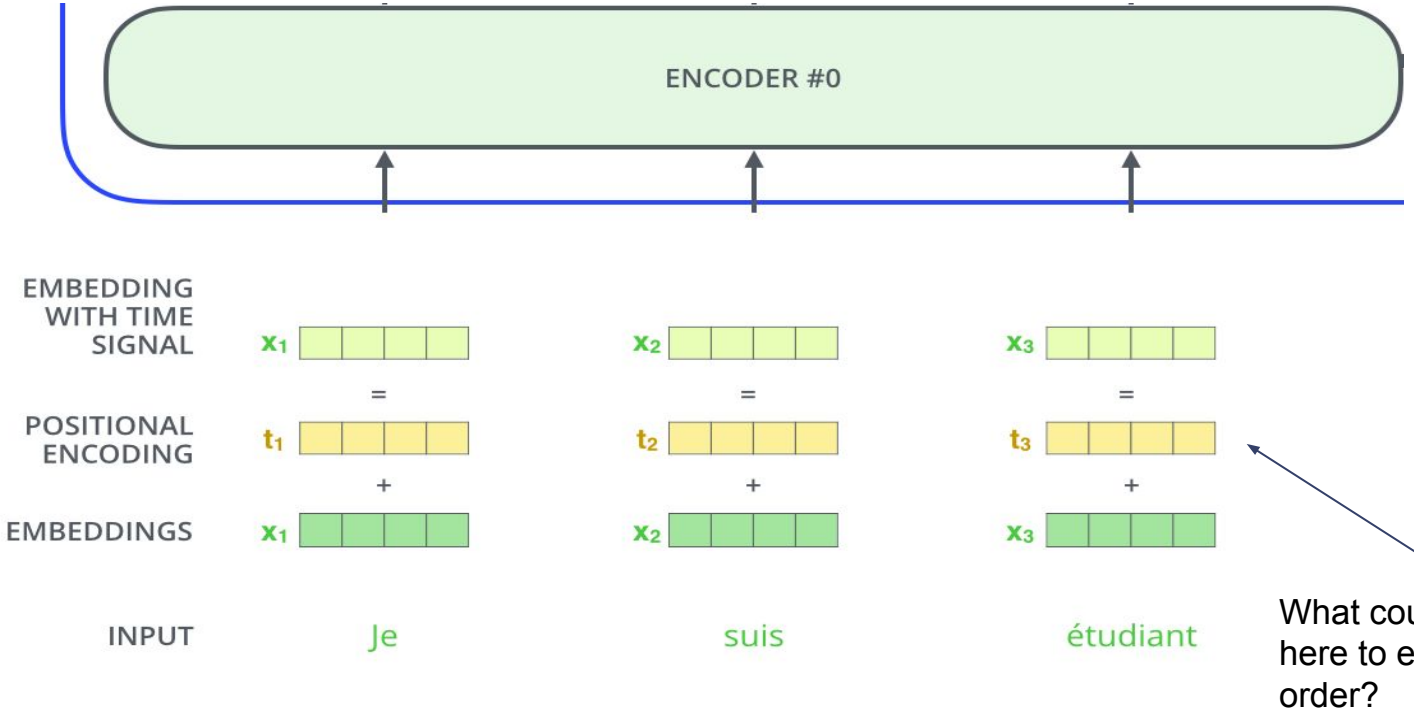
- Encode the order of the sequence



Source: https://jalammar.github.io/illustrated-transformer/

Positional Encoding

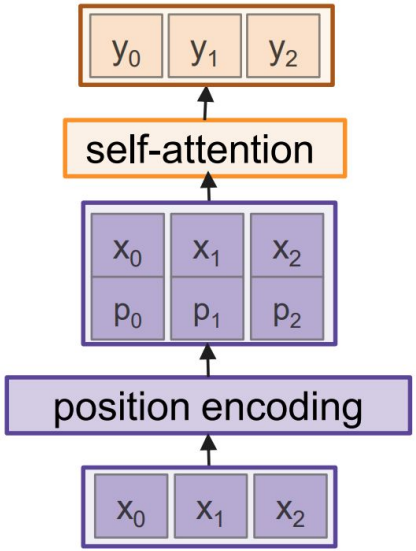
- Encode the order of the sequence



Source: https://jalammar.github.io/illustrated-transformer/

Positional Encoding

- Concatenate or add special positional encoding p_j to each input vector x_j
- We use a function $pos: N \rightarrow R^d$ to process the position j of the vector into a d-dimensional vector

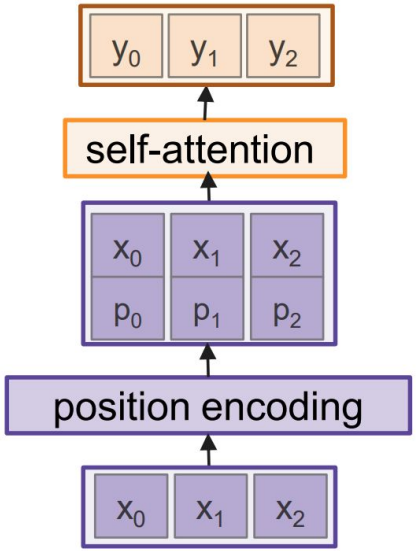


Possible desirable properties of *pos*(.):

1. It should output a **unique** encoding for each timestep (word's position in a sentence)
2. **Distance** between any two time-steps should be consistent across sentences with different lengths.
3. The model should generalize to **longer** sentences without any efforts. Its values should be bounded.
4. It must be **deterministic**.

Positional Encoding

- Concatenate or add special positional encoding p_j to each input vector x_j
- We use a function $pos: N \rightarrow R^d$ to process the position j of the vector into a d-dimensional vector



Options for *pos*(.):

1. Learn a lookup table:
   - Learn parameters to use for *pos*(t) for t ε [0, T)
   - Lookup table contains T x d parameters.
2. Design a fixed function with the desired properties

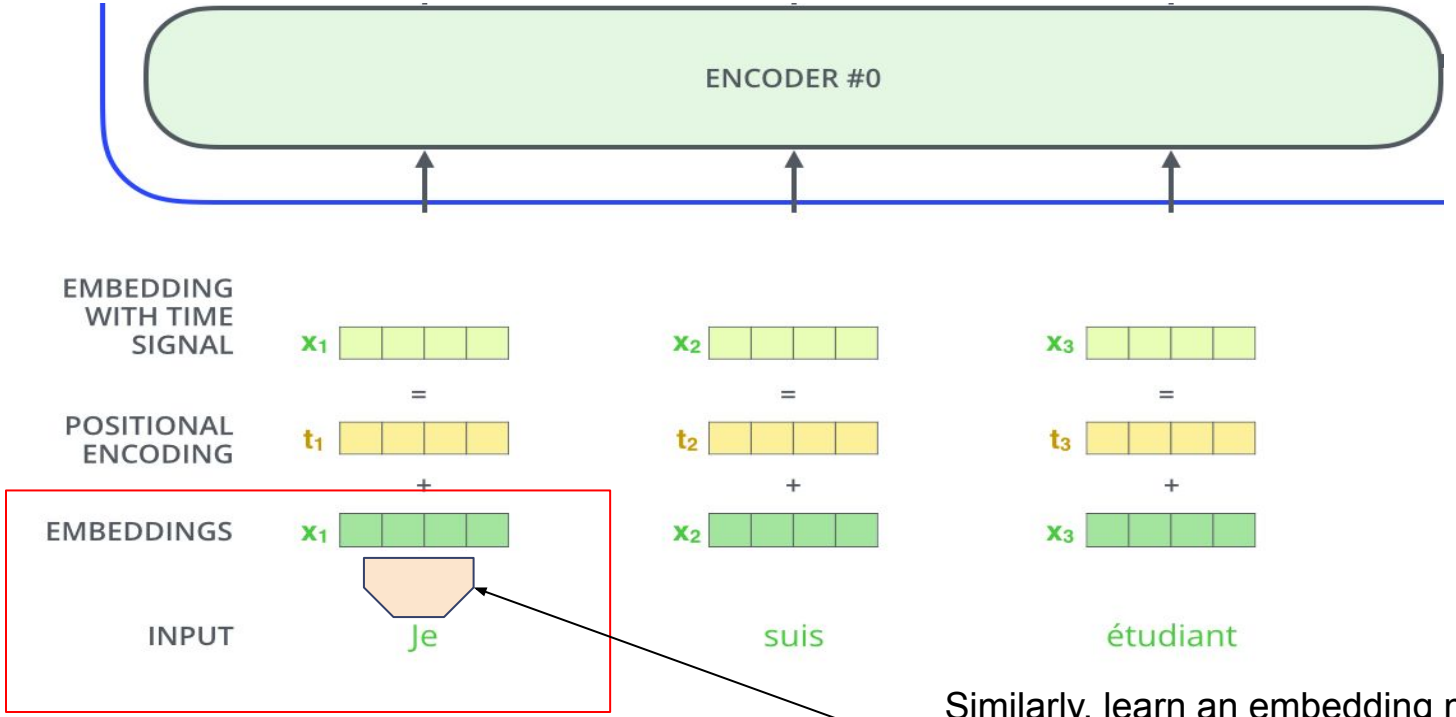
$$p(t) = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \vdots \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_d$$

Intuition:

0:	0	0	0	0	8:	1	0	0	0
1:	0	0	0	1	9:	1	0	0	1
2:	0	0	1	0	10:	1	0	1	0
3:	0	0	1	1	11:	1	0	1	1
4:	0	1	0	0	12:	1	1	0	0
5:	0	1	0	1	13:	1	1	0	1
6:	0	1	1	0	14:	1	1	1	0
7:	0	1	1	1	15:	1	1	1	1

where $10000^{2\kappa/u}$

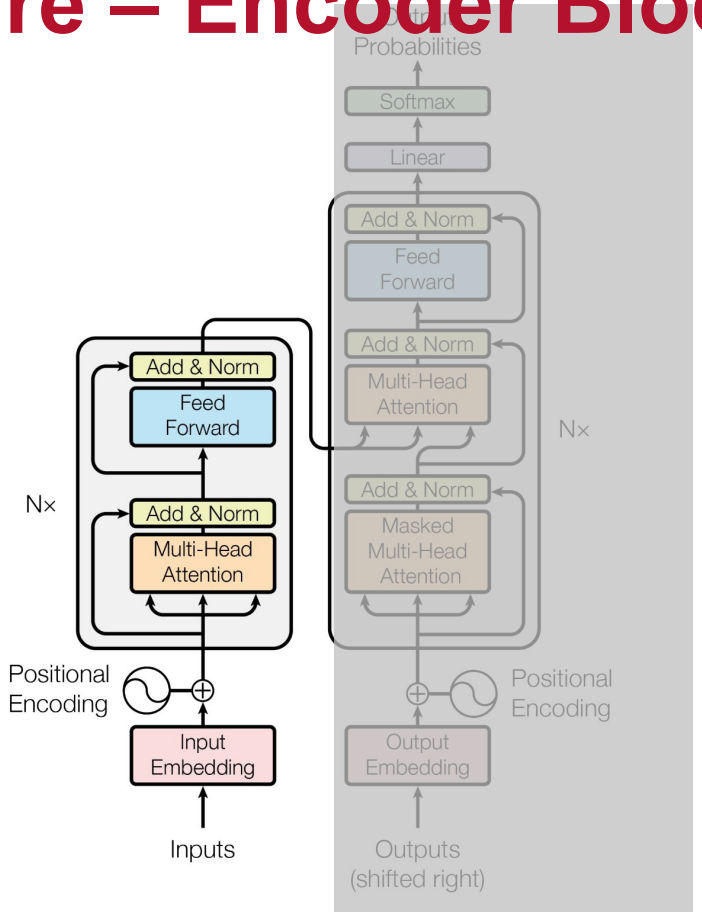
Also embed tokens into vectors



Similarly, learn an embedding matrix for each word in Vocab. Learn entirely end-to-end

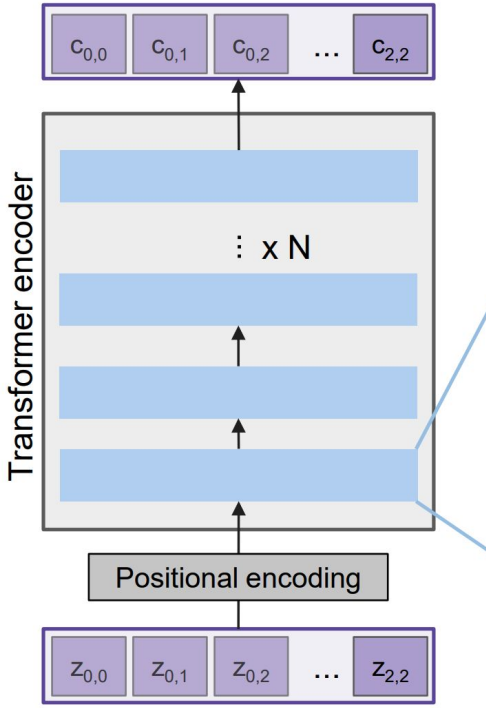
Source: https://jalammar.github.io/illustrated-transformer/

Transformer Architecture – Encoder Block



Source: https://jalammar.github.io/illustrated-transformer/

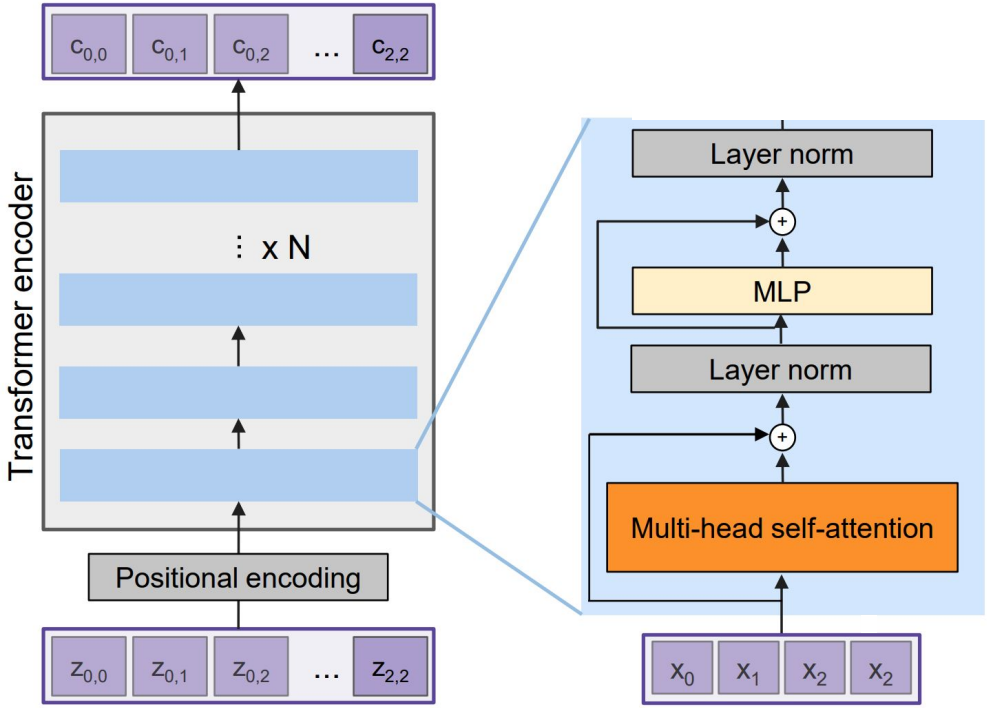
Transformer Architecture – Encoder Block



Made up of N encoder blocks.

E.g., $N = 6$, $D_q = 512$

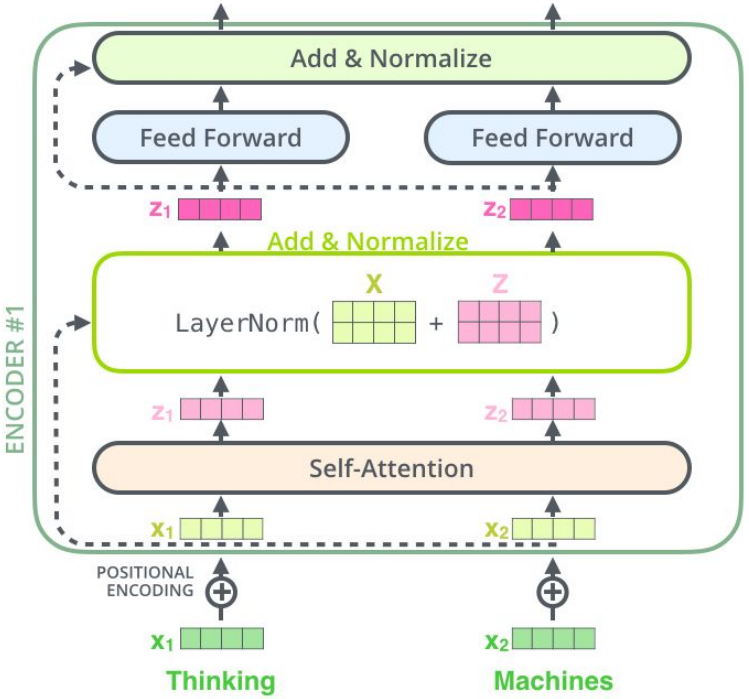
Transformer Architecture – Encoder Block



Let's dive into one encoder block

Transformer Architecture – Encoder Block

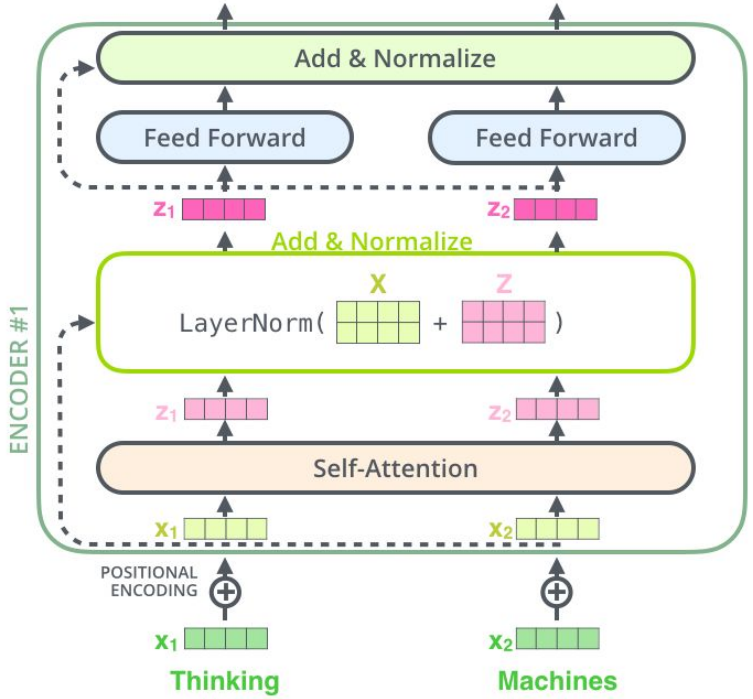
- Multi-head attention



Source: https://jalammar.github.io/illustrated-transformer/

Transformer Architecture – Encoder Block

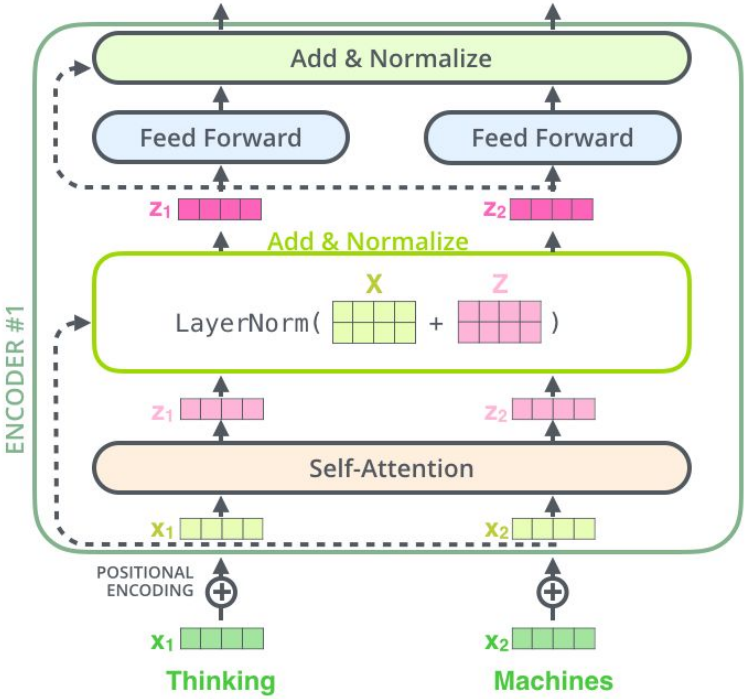
- Multi-head attention
- Residual connection
  - A mechanism introduced in CNN
  - Help with learning with deep layers



Source: https://jalammar.github.io/illustrated-transformer/

Transformer Architecture – Encoder Block

- Multi-head attention
- Residual connection
  - A mechanism introduced in CNN
  - Help with learning with deep layers
- Layer Norm
  - Normalization layer that is robust to varied input length



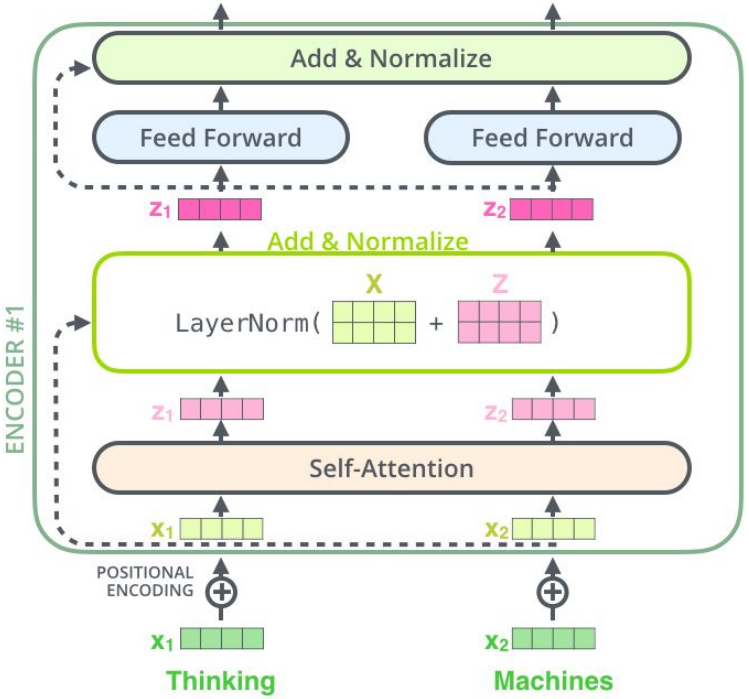
Source: https://jalammar.github.io/illustrated-transformer/

Transformer Architecture – Encoder Block

- Multi-head attention
- Residual connection
  - A mechanism introduced in CNN
  - Help with learning with deep layers
- Layer Norm
  - Normalization layer that is robust to varied input length
- Feed forward network

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

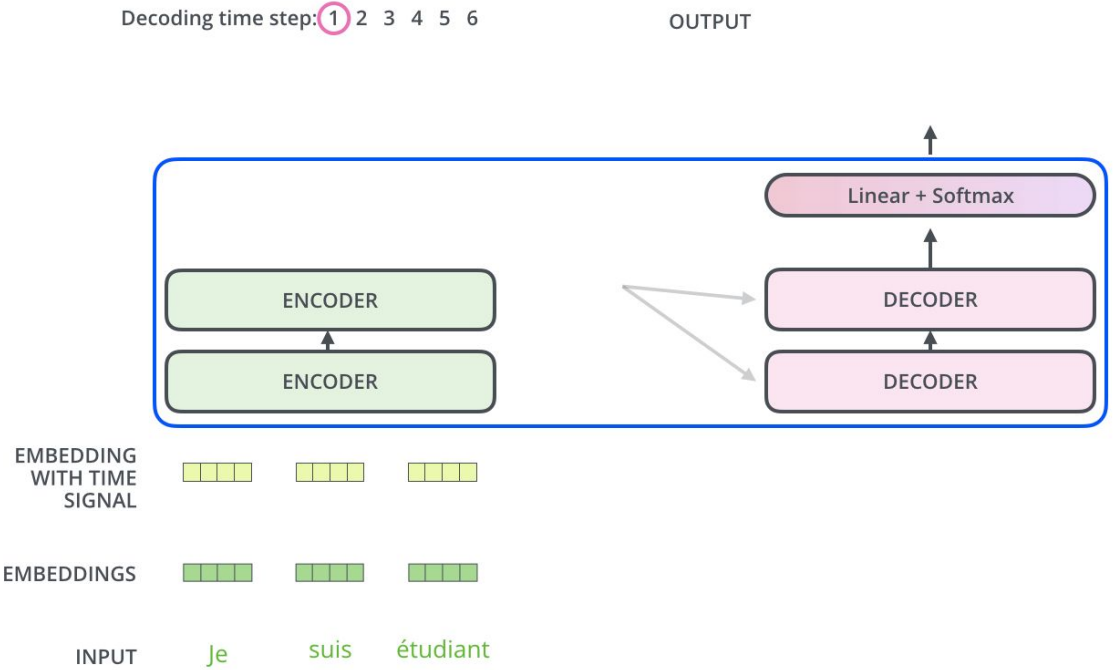
  - Fully-connected linear layers
  - First increase the dimensionality and then reduce to the original dimensionality



Source: https://jalammar.github.io/illustrated-transformer/

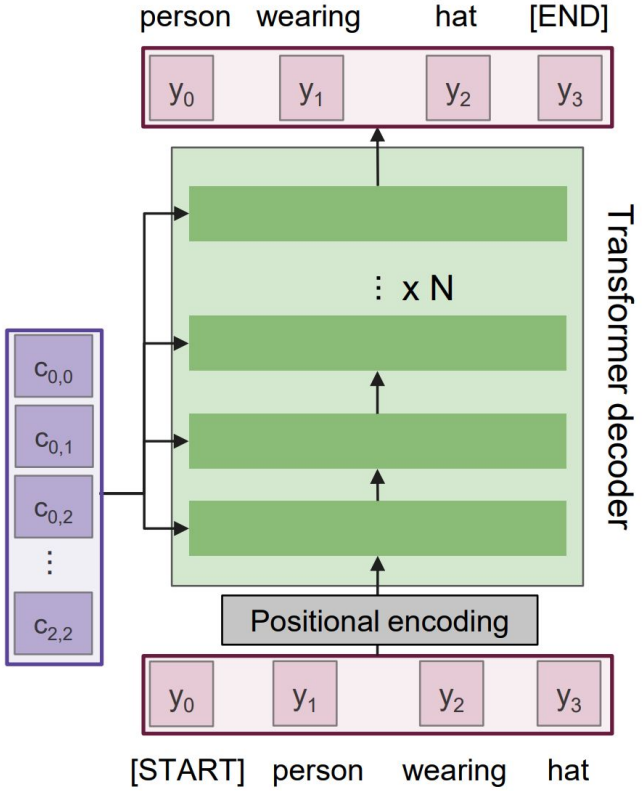
Transformer Architecture Decoder

- The output of the top encoder is used by each decoder to focus on appropriate places in the input sequence



Source: https://jalammar.github.io/illustrated-transformer/

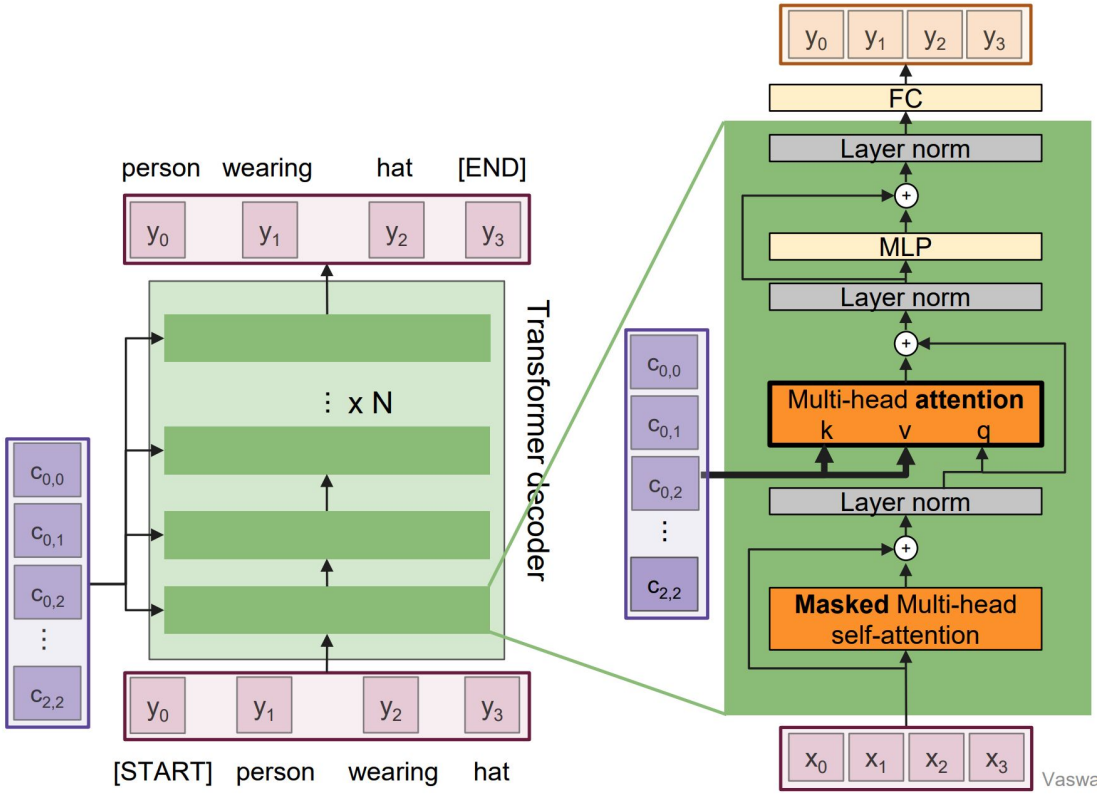
Transformer Architecture – Decoder



Made up of N decoder blocks.

E.g., $N = 6$, $D_q = 512$

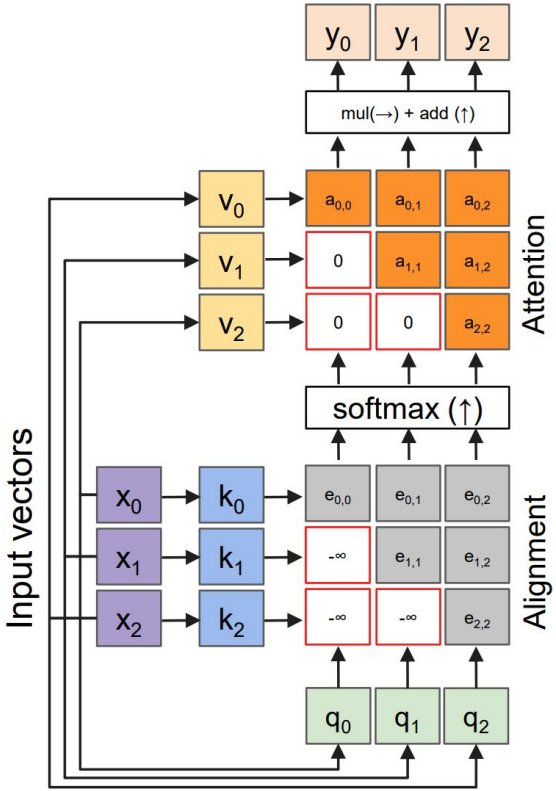
Transformer Architecture – Decoder



Quite Similar to Encoder

Main difference - attends to encoder tokens as well

Masked Self-Attention



Outputs:
context vectors: **y** (shape: D_v)

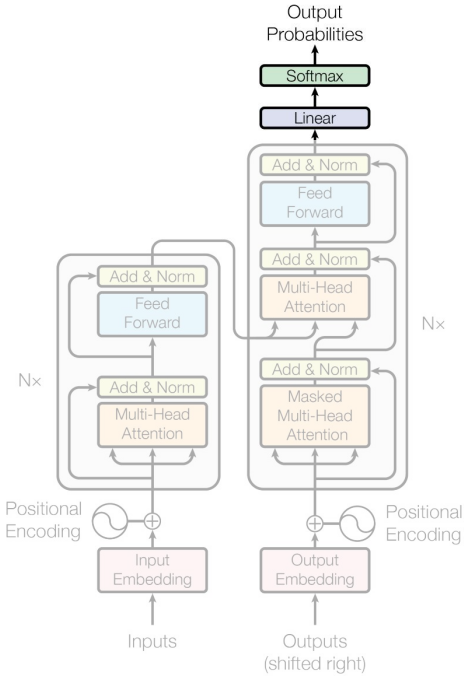
Operations:
Key vectors: $\mathbf{k} = \mathbf{x}W_k$
Value vectors: $\mathbf{v} = \mathbf{x}W_v$
Query vectors: $\mathbf{q} = \mathbf{x}W_q$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:
Input vectors: **x** (shape: N x D)

- Allows us to parallelize attention across time
- Don't need to calculate the context vectors from the previous timestep first!

- Prevent vectors from looking at future vectors.
- Manually set alignment scores to –infinity (-nan)

Transformer Final Layer



Assume we have already generated K tokens, generate the next one.

The decoder was used to gather all information necessary to predict a probability distribution for the next token (K), over the whole vocab.

Simple:

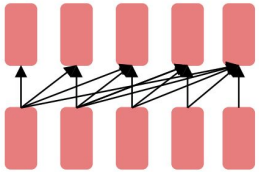
- linear projection of token K
- SoftMax normalization

Slide credit: Ana Marasović

Transformers - Summary

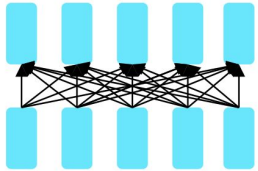
- The **general attention layer** is a new type of layer that can be used to design new neural network architectures
- Transformers are a type of layer that uses **self-attention** and layer norm.
  - It is highly scalable and highly parallelizable
  - **Faster** training, **larger** models, **better** performance across vision and language tasks
  - They are quickly replacing RNNs, LSTMs, and may(?) even replace convolutions.

Many Architectures for Large Language Models



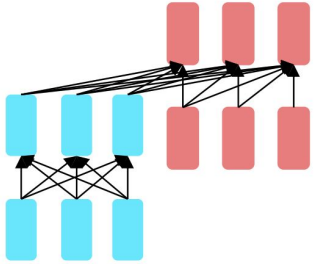
Decoders

GPT, Claude,
Llama
Mixtral



Encoders

BERT family,
HuBERT

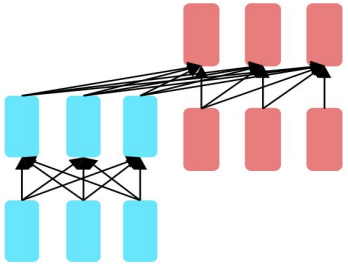


Encoder-decoders

Flan-T5, Whisper

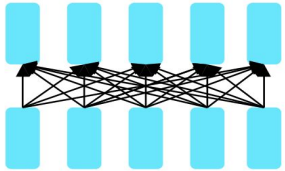
Encoder-Decoders

- Input sequence is encoded and then encoded sequence is decoded by another part of the network
- This is the architecture we just looked at
- Trained to map from one sequence to another
- Very popular for:
  - machine translation (map from one language to another)
  - speech recognition (map from acoustics to words)



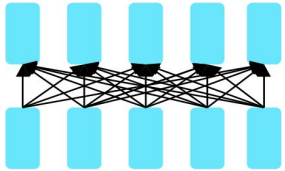
Encoder-Only

- Just encode the sequence (no next token prediction)
- Do something with that encoded sequence
  - Classification
  - Predict a masked token at one element of the sequence
- Many varieties!
- Popular: Masked Language Models (MLMs)
- BERT family



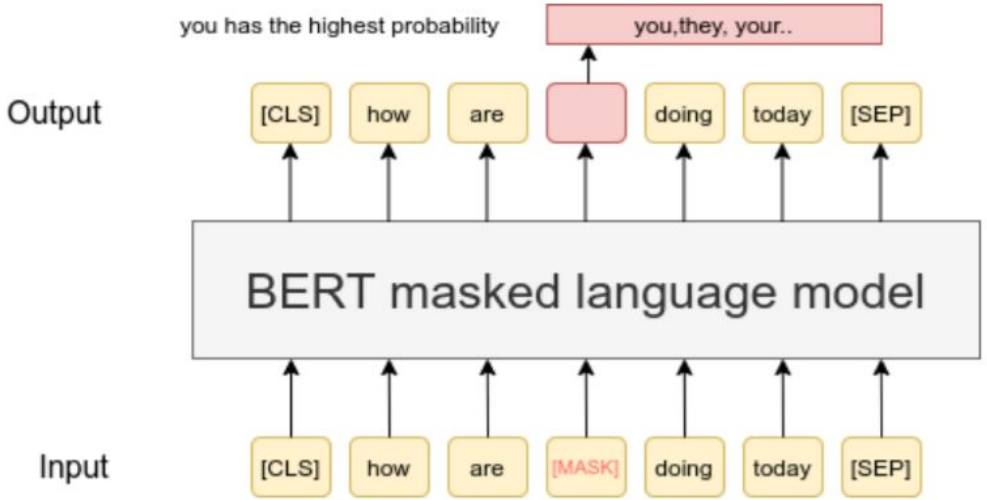
Encoder-Only

- Trained by predicting words from surrounding words on both sides
- Are usually **finetuned** (trained on supervised data) for classification tasks.



BERT

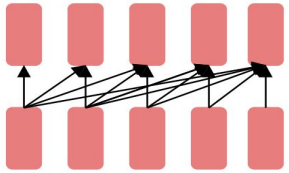
- Mask some tokens in the input and use the hidden representation to predict the masked tokens
- Equivalent to perform a classification task, the number of classes is the total number of tokens



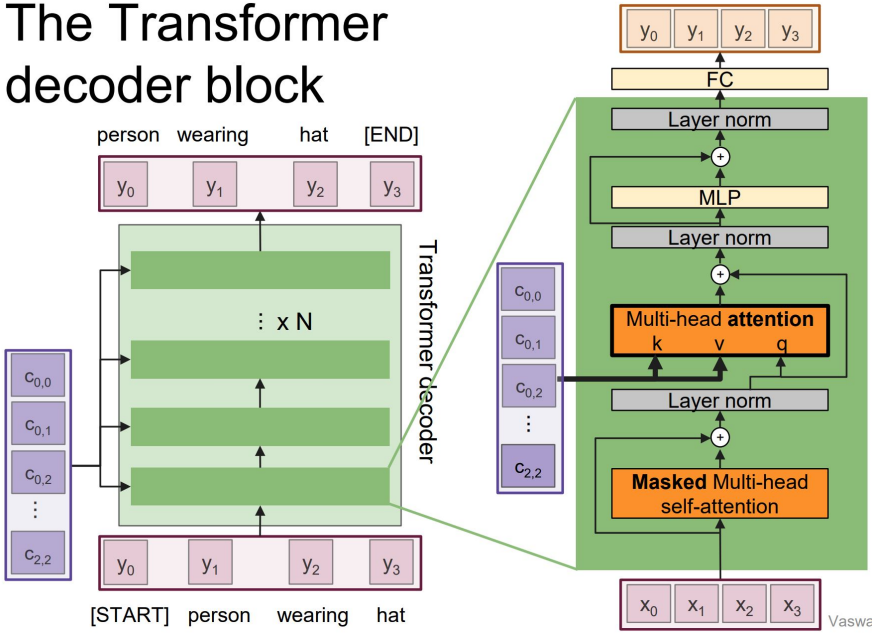
Decoder-Only

Also called:

- Causal LLMs
- Autoregressive LLMs
- Left-to-right LLMs
- Predict words left to right



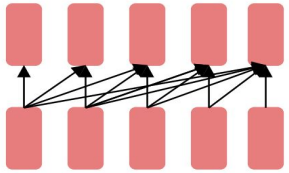
The Transformer decoder block



Decoder-Only

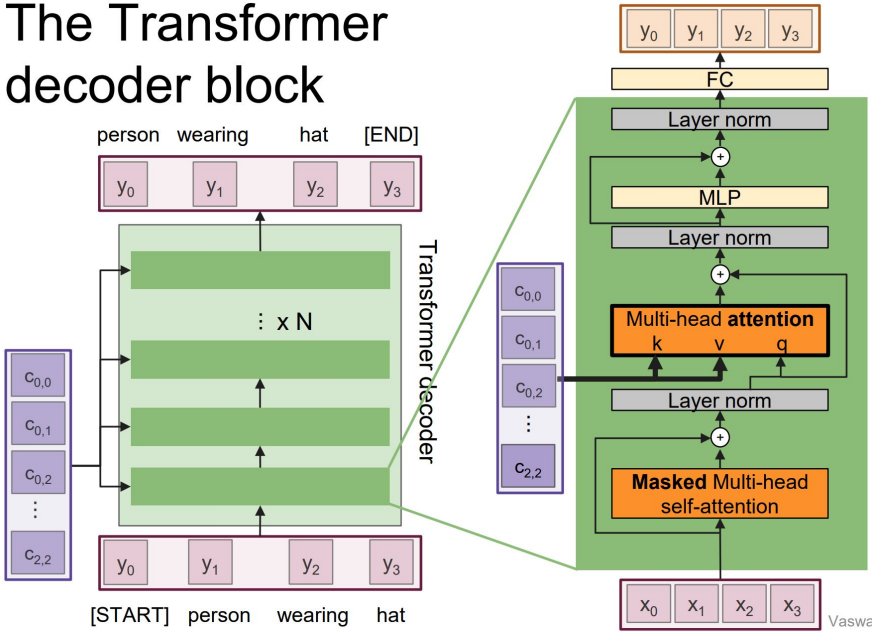
Also called:

- Causal LLMs
- Autoregressive LLMs
- Left-to-right LLMs
- Predict words left to right



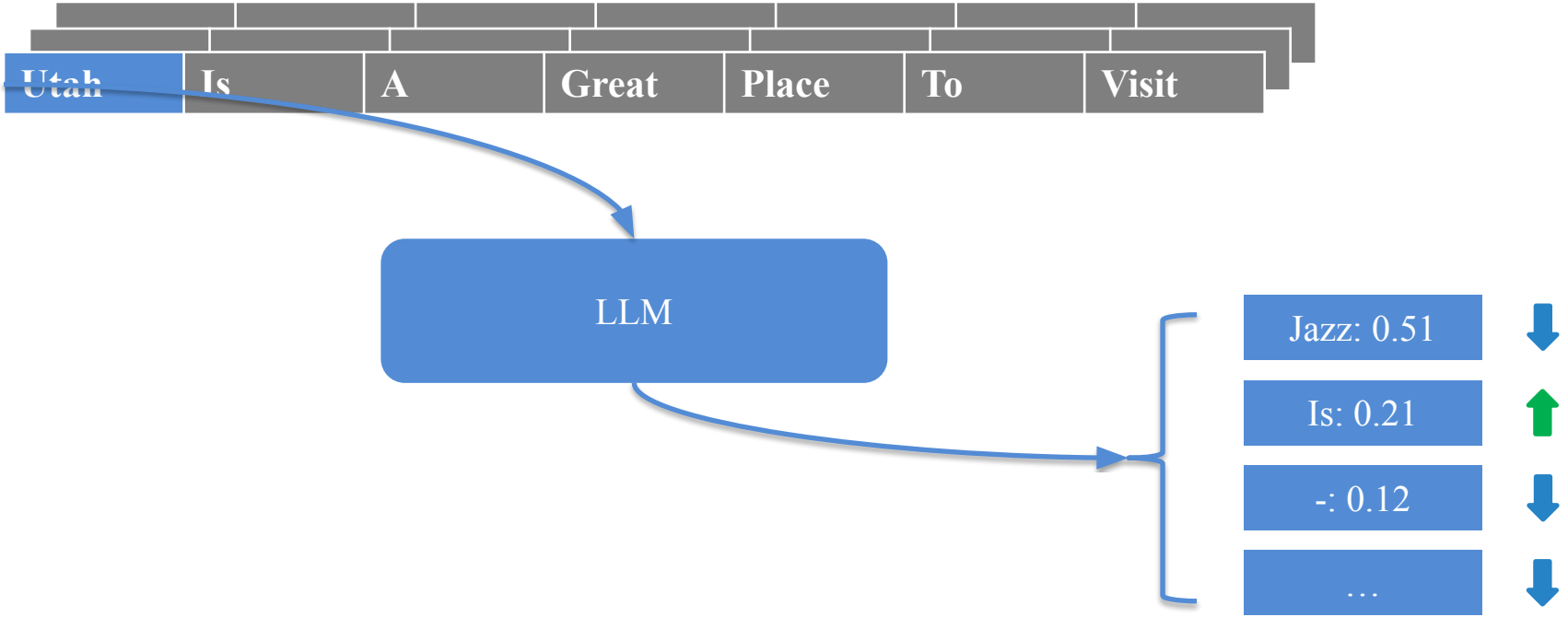
We can completely skip the step where we encode the input, just feed input and output into the decoder

The Transformer decoder block



Training Autoregressive LLMs

- It's all about predicting the next word!
  - What should be the next word following word "Utah"?

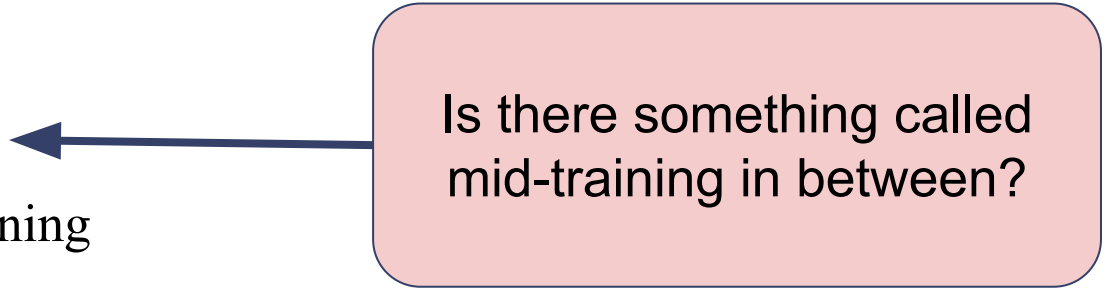


LLM Training Stages

- Pre-training
- Supervised fine-tuning
- Alignment (RLHF: Reinforcement Learning from Human Feedback)

LLM Training Stages

- Pre-training
- Supervised fine-tuning
- Alignment (RLHF: Reinforcement Learning from Human Feedback)



See: https://vintagedata.org/blog/posts/what-is-mid-training

LLM Training Stages: Pre-training

- Goal
  - Enable comprehensive language processing capability
- Data
  - Plain text including billion to trillion of tokens scrapped from Internet (super large)

How much Pre-training Data?

[Training Compute-Optimal Large Language Models]

➔ "...given a 10× increase computational budget, they [Kaplan et al., 2020] suggests that the size of the model should increase 5.5× while the number of training tokens should only increase 1.8×. Instead, we find that model size and the number of training tokens should be scaled in equal proportions."

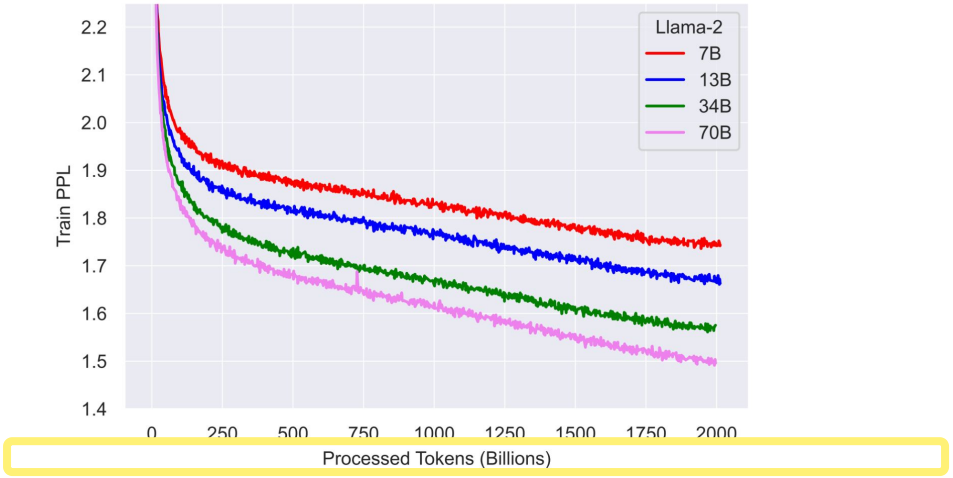


Figure 5: Training Loss for LLAMA 2 models. We compare the training loss of the LLAMA 2 family of models. We observe that after pretraining on 2T Tokens, the models still did not show any sign of saturation.

LLM Training Stages: Pre-training

- Goal
  - Enable comprehensive language processing capability
- Data
  - Plain text including billion to trillion of tokens scrapped from Internet (super large)
- Training task
  - Next word prediction

LLM Training Stages: Pre-training

Self-supervised training algorithm

- We just train them to predict the next word!
  - Take a corpus of text
  - At each time step t
    - ask the model to predict the next word
    - train the model using gradient descent to minimize the error in this prediction

“Self-supervised” because it just uses the next word as the label!

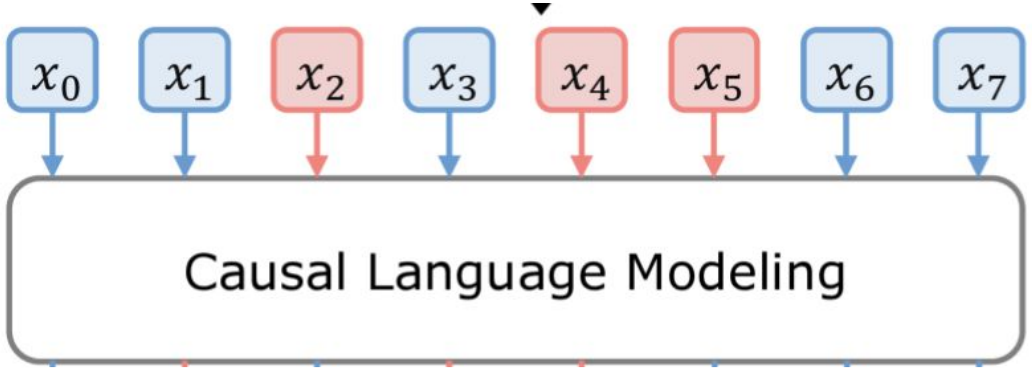
Cross-entropy loss: the model assigns a high probability to true word w

Pre-training Loss (Simplified/Wrong)

Data:

The quick brown fox jumped over the lazy dog

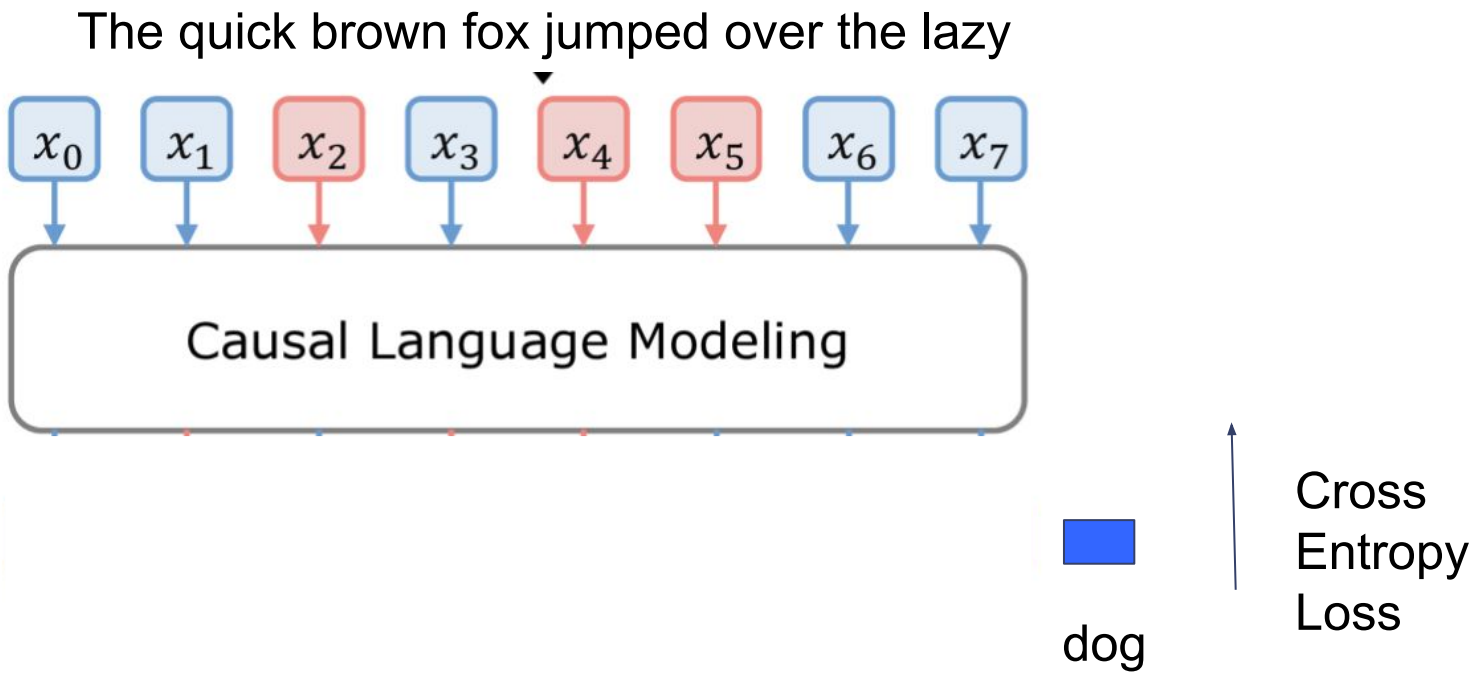
The quick brown fox jumped over the lazy



dog

Pre-training Loss (Simplified/Wrong)

Data:
The quick brown fox jumped over the lazy dog

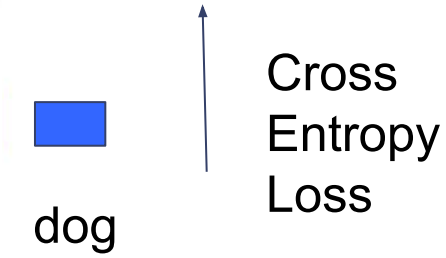


Pre-training Loss (Simplified/Wrong)

Data:

The quick brown fox jumped over the lazy dog

This is kind of one of those lies they teach in school

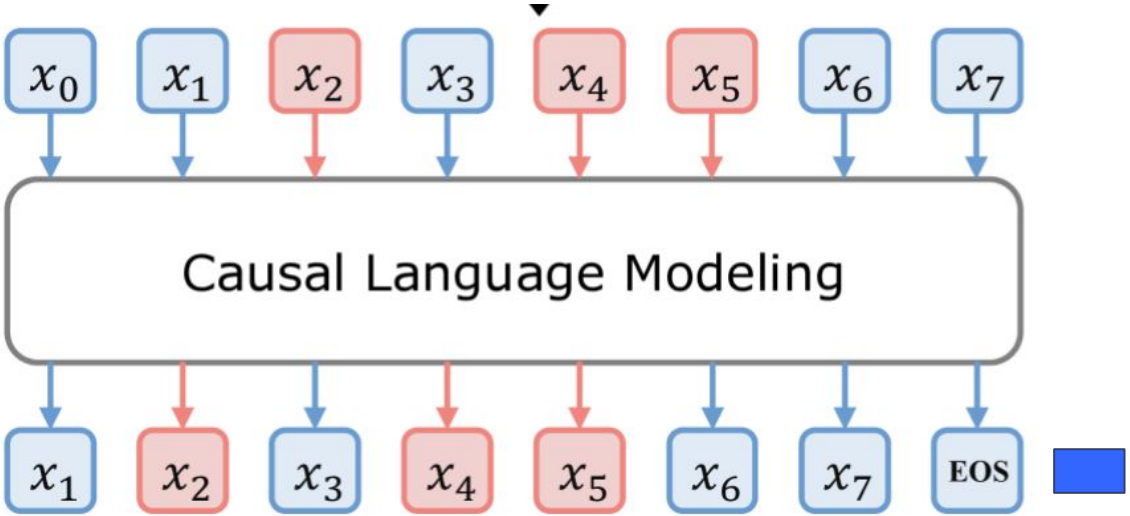


Pre-training Loss (Real)

Data:

The quick brown fox jumped over the lazy dog

The quick brown fox jumped over the lazy

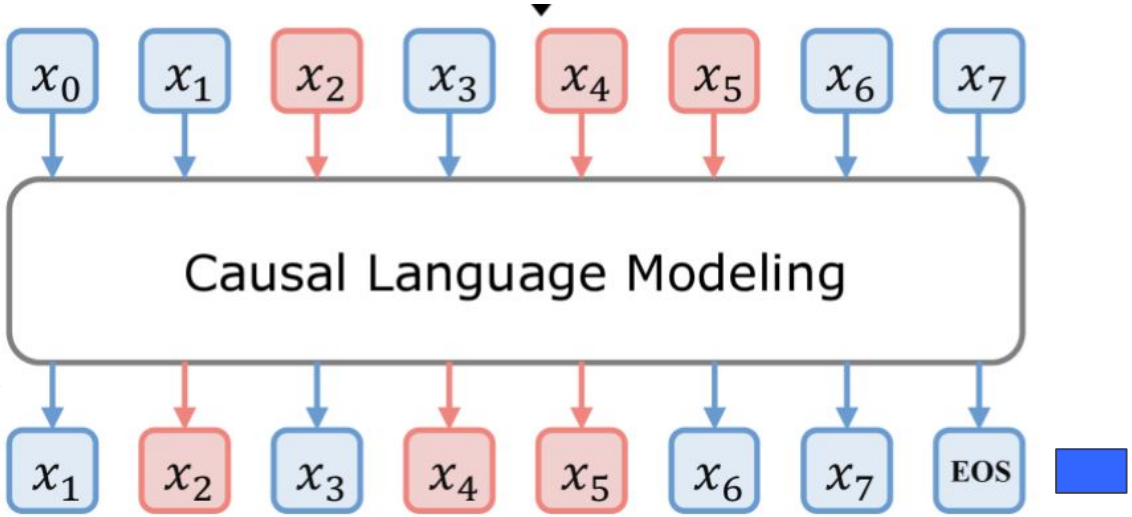


quick brown fox jumped over the lazy dog

Pre-training Loss (Real)

Data:
The quick brown fox jumped over the lazy dog

The quick brown fox jumped over the lazy



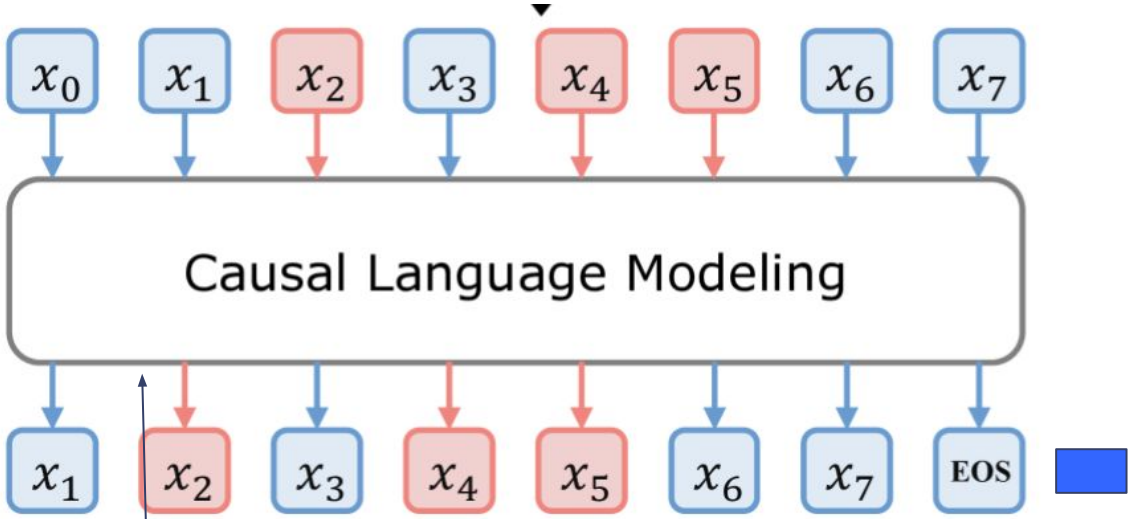
CEL on
P(quick|The)

quick brown fox jumped over the lazy dog

Pre-training Loss (Real)

Data:
The quick brown fox jumped over the lazy dog

The quick brown fox jumped over the lazy



CEL on P(brown|The quick)

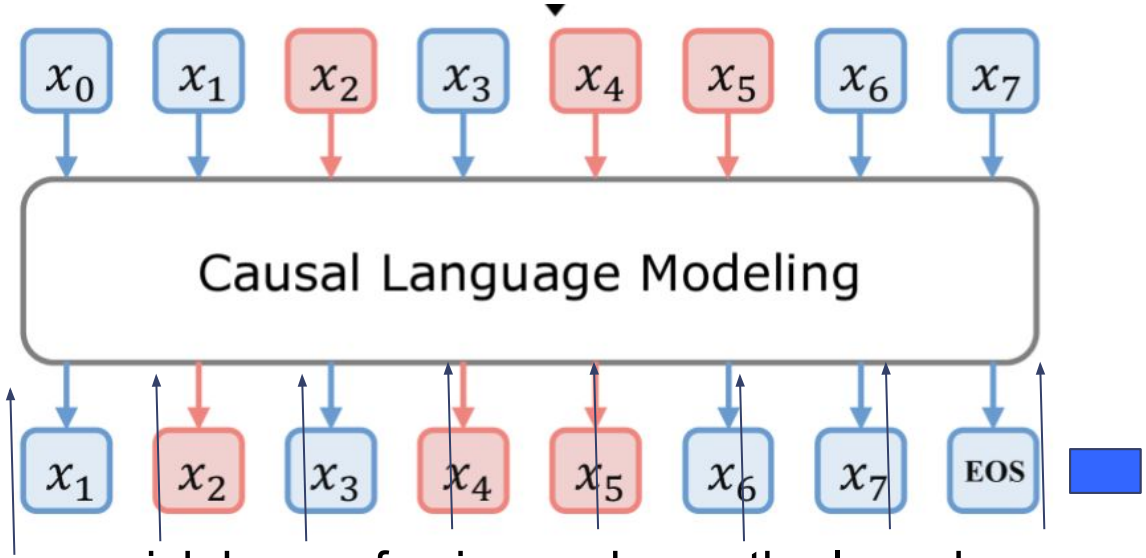
quick brown fox jumped over the lazy dog

Pre-training Loss (Real)

Data:

The quick brown fox jumped over the lazy dog

The quick brown fox jumped over the lazy



Etc

quick brown fox jumped over the lazy dog

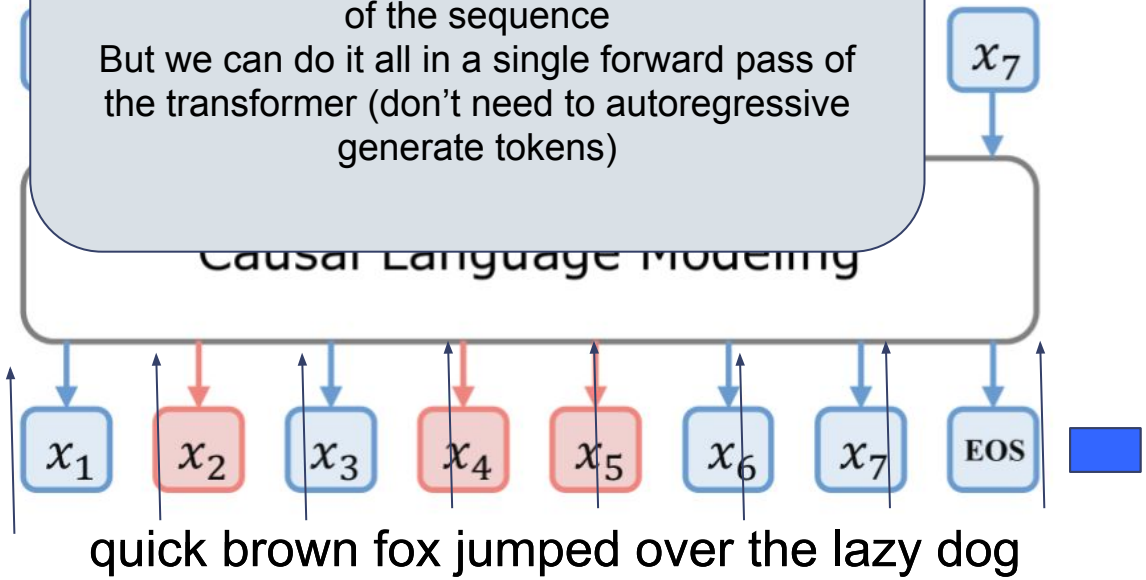
Pre-training Loss (Real)

Data:

Th … do … laz …

This is important because it means that during training - we can train on completing each token of the sequence
But we can do it all in a single forward pass of the transformer (don't need to autoregressive generate tokens)

This is important because this is WHY Transformers can be massively scaled to train on lots of data!



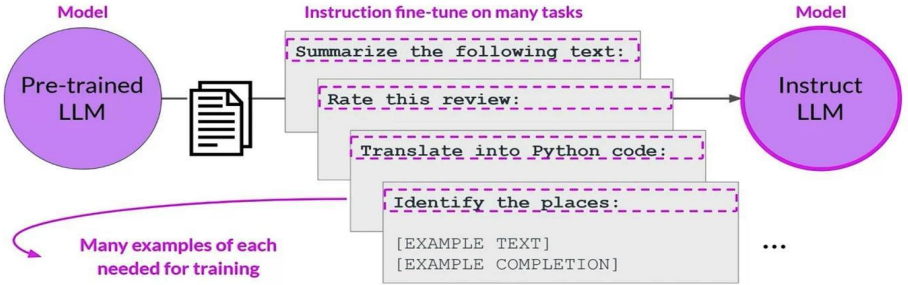
Etc

LLM Training Stages: Pre-training

- Goal
  - Enable comprehensive language processing capability
- Data
  - Plain text including billion to trillion of tokens scrapped from Internet (super large)
- Training task
  - Next word prediction
- Production
  - Usually called the **base** model
  - Pre-train base LLM (Llama-2-7b-base, etc.)

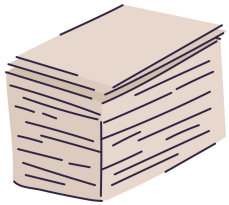
LLM Training Stages: Supervised Fine-Tuning

- Goal
  - Enable instruction following capability
  - Adaptation to new domains
- Data
  - A set of instructions and their corresponding outputs (relatively small)
- Training task
  - Predicting target outputs
- Production
  - Usually called the **instruct** model
  - Chat LLM (Llama-2-7b-chat, etc)



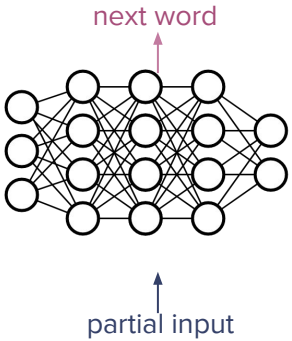
Pre-train/Finetune Paradigm

Stage 1:
Pretrain a model



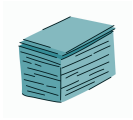
text

+



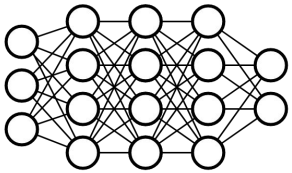
Objective: generate next word
(does not require that people label the next word)

Stage 2:
Finetune the model



text + **labels**

+

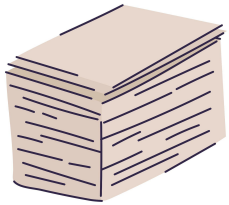


Objective: standard supervised training

Pre-train/Finetune Paradigm

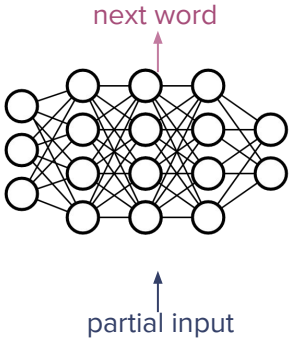
Stage 1:
Pretrain a model

Train on as much data as possible
Get a really general understanding of language



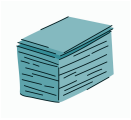
text

\+



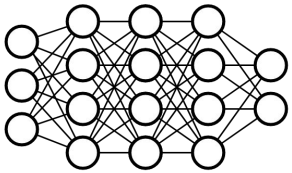
Objective: generate next word
(does not require that people label the next word)

Stage 2:
Finetune the model



text + **labels**

\+

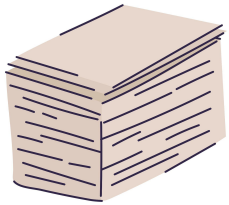


Objective: standard supervised training

Pre-train/Finetune Paradigm

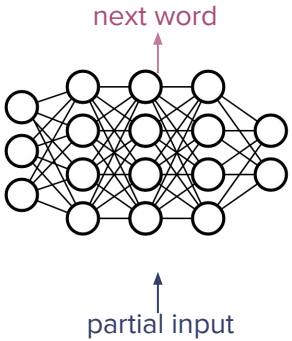
Stage 1:
Pretrain a model

Train on the data you care about
Perform better on that task



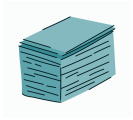
text

+



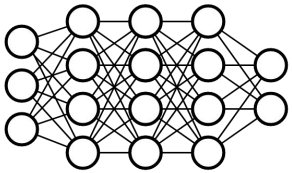
Objective: generate next word
(does not require that people label the next word)

Stage 2:
Finetune the model



text + **labels**

+



Objective: standard supervised training

For Agents

- Finetuning is a very common way to adapt
  - Take a capable base model
  - Get examples of your agent task
  - Finetune
  - (Note: this is the same as Behavioral Cloning)
- Models are now often pre-trained or mid-trained on agent data as well

LLM Training Stages

- Pre-training
- Supervised fine-tuning
- Alignment (RLHF: Reinforcement Learning from Human Feedback)

RLHF at a high level

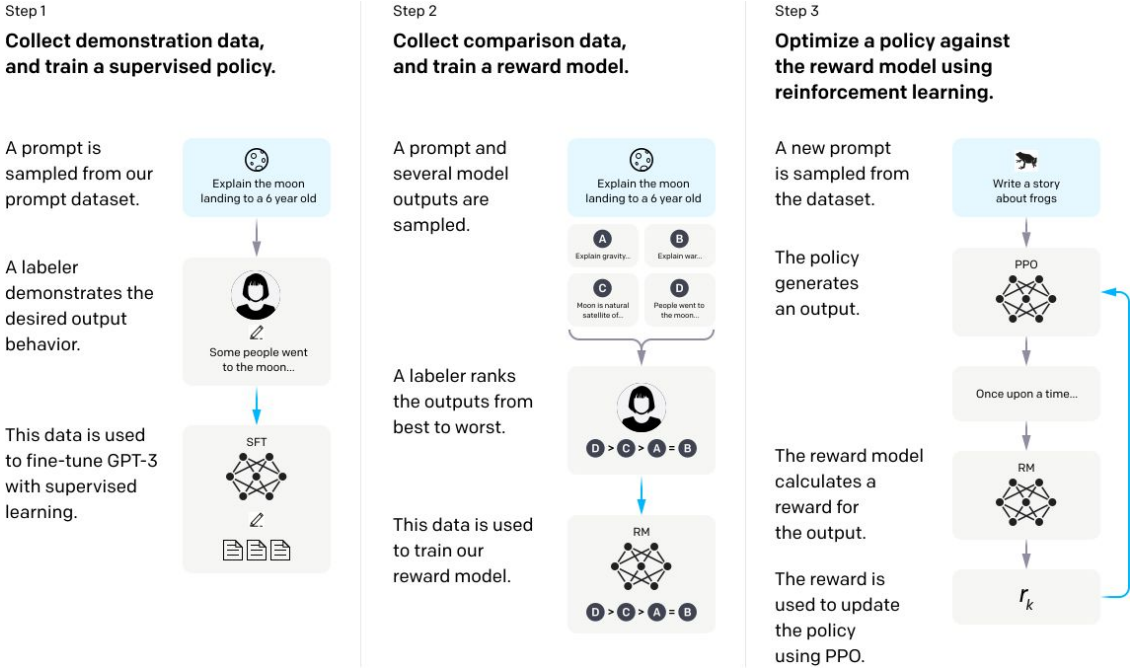


Figure from: https://openai.com/blog/instruction-following/

For Agents

- Pre-training

Usually starting from another pre-trained model

- Supervised fine-tuning

Very common

- Alignment (RLHF: Reinforcement Learning from Human Feedback)

RLHF is less common, but has been used

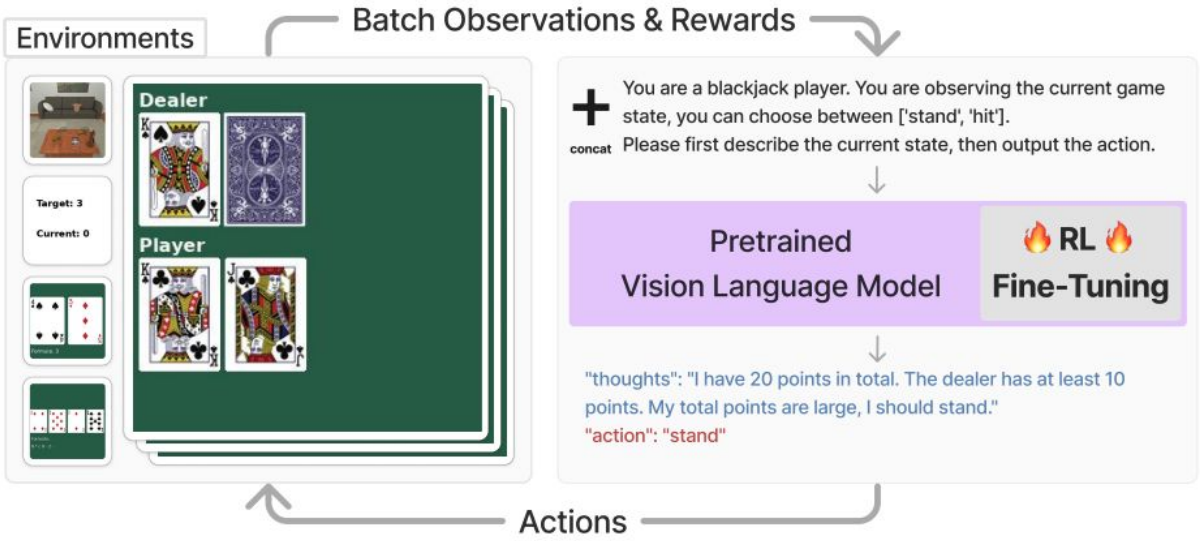
For Age

- Pre-trainin
- Supervised
- Alterna

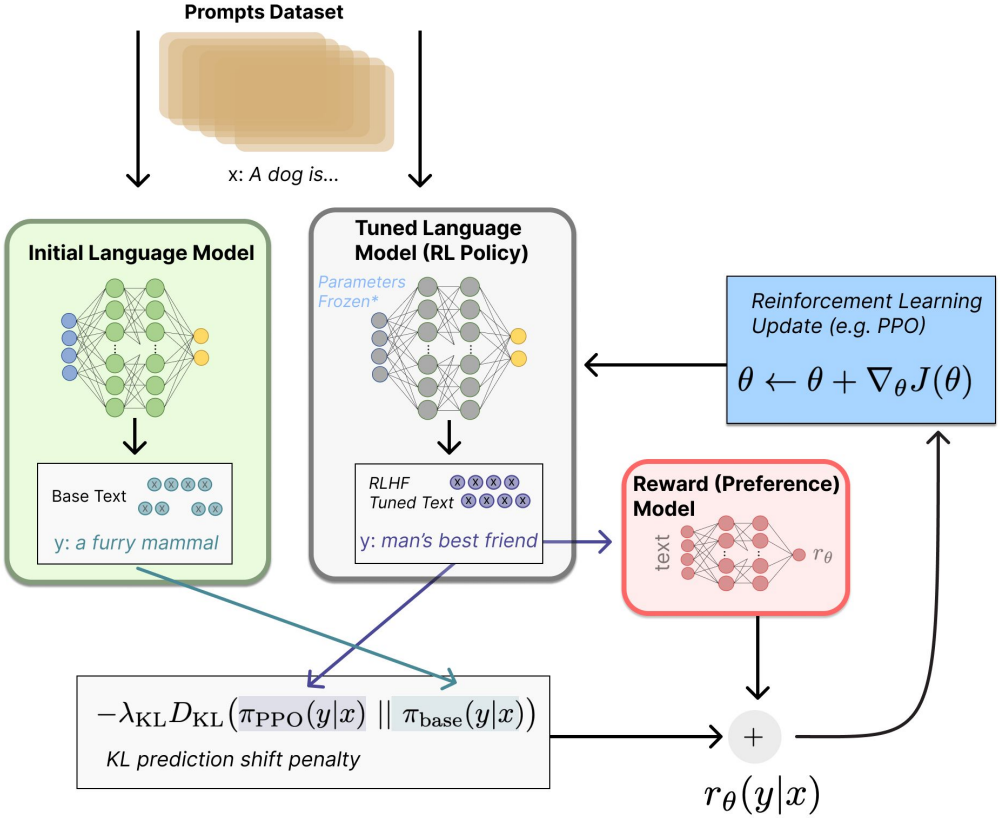
Fine-Tuning Large Vision-Language Models as Decision-Making Agents via Reinforcement Learning

Yuexiang Zhai[1*] Hao Bai[2†] Zipeng Lin[1†] Jiayi Pan[1†] Shengbang Tong[3†] Yifei Zhou[1†]

Alane Suhr[1] Saining Xie[3] Yann LeCun[3] Yi Ma[1] Sergey Levine[1]

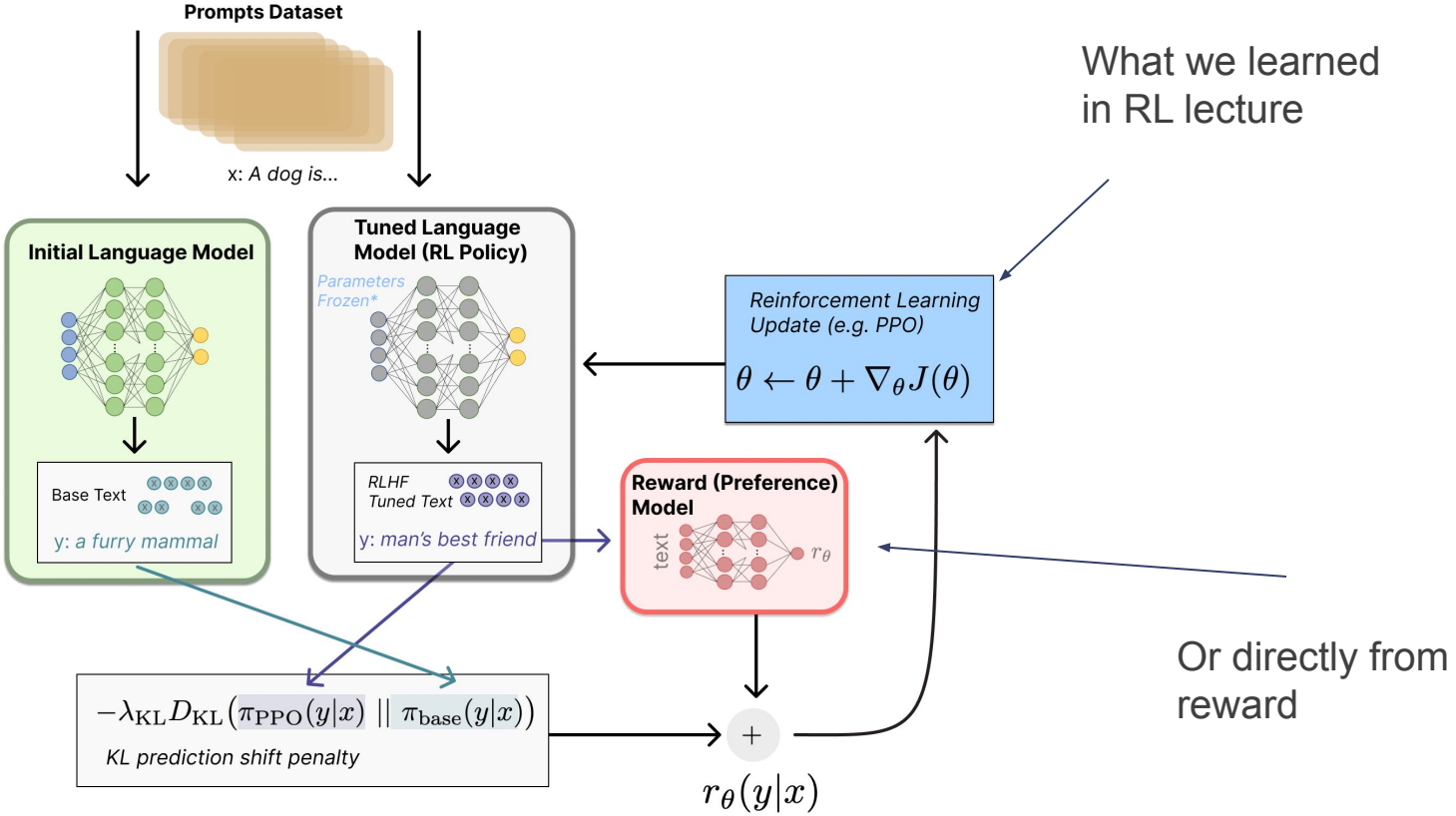


RL Finetuning



What we learned in RL lecture

RL Finetuning



LLM Training Stages: Alignment

- Goal
  - Steering LLM to align with human standard/ethics or some final task
- Data
  - Human preference data (smaller)
  - Or direct reward
- Training task
  - Reinforcement Learning from Human Feedback (RLHF)
  - Maximizing reward (RL)
- Production
  - Usually called the **aligned** model
  - Aligned Chat LLM (Llama-2-7b-chat-hf, etc)

Now you know everything about LLMs

Now you know everything about LLMs…

Sort of.

- There are actually a LOT of other topics here we didn't cover
  - I did not cover test-time compute ("thinking models")
- A ton of technical depth in all of these we didn't cover
- Other topics may come up later in the class / in specific papers

Recommended Readings

- NLP Courses and Tutorials
  - Utah NLP Course: https://utah-cs6340-nlp.notion.site/
  - Stanford NLP Course: https://web.stanford.edu/class/cs224n/
  - HuggingFace LLM Tutorial: https://huggingface.co/learn/llm-course/en/chapter1/1
- More resources
  - Nathan Lambert RLHF Intro: https://rlhfbook.com/c/14-reasoning#ref-wang2025ragen
  - Transformer paper: http://arxiv.org/abs/1706.03762
  - Annotated transformer: https://nlp.seas.harvard.edu/2018/04/03/attention.html

Any Questions

Questions