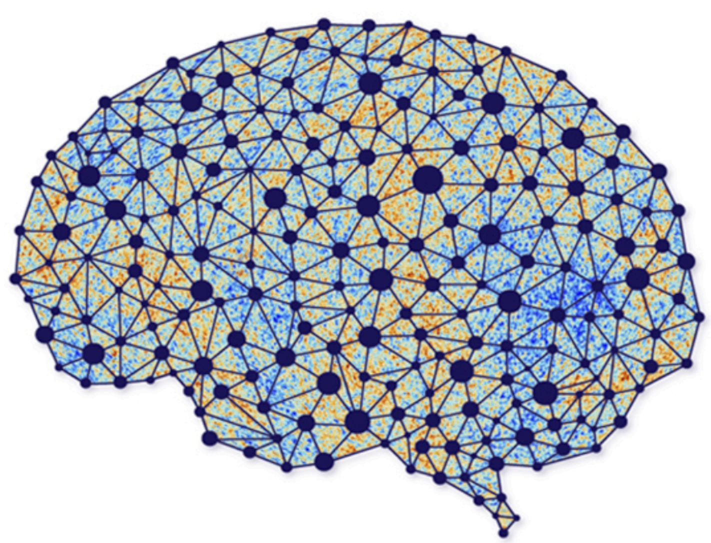
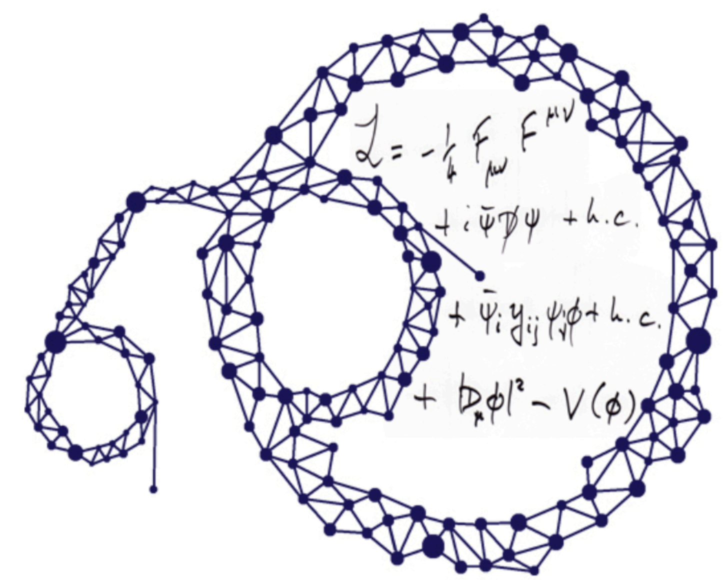
Physics 361 - Machine Learning in Physics

Lecture 16 – Large Language Models

March 13th 2025



AI
∩
Universe



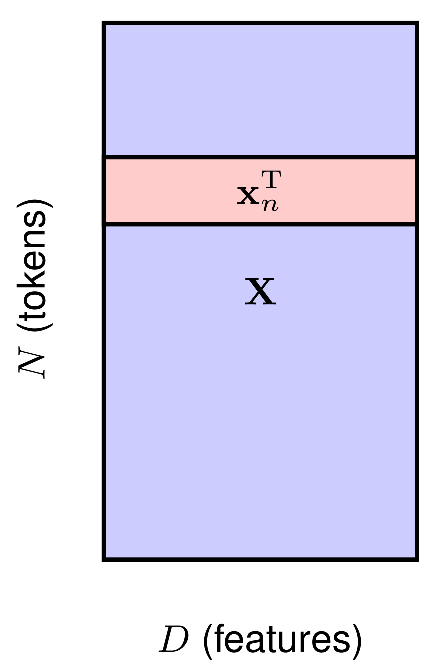
Moritz Münchmeyer

Transformers

Recall: Transformers

Transformer Processing

- Input data is a set of vectors $\{\mathbf{x}_n\}$ of dimensionality D, $n = 1,\ldots,N$.
- These data vectors are known as **tokens** (e.g., a word within a sentence, a patch within an image, or an amino acid within a protein).
- The elements x_{ni} of the tokens are called **features**.
- Transformers can handle a mix of different data types by combining the data variables into a joint set of tokens.
- Combining the data vectors into a matrix X of dimensions $N \times D$.



$$\widetilde{\mathbf{X}} = \mathrm{TransformerLayer}\left[\mathbf{X}\right]$$

same dimensionality as X

Apply multiple transformer layer
to learn rich internal representations.

Network Parameters

- Define query, key, & value matrices each w/ different transformations:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathrm{q})}$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathrm{k})}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathrm{v})}$$

  the weight matrices $\mathbf{W}^{(q)}, \mathbf{W}^{(k)}, \mathbf{W}^{(v)}$ represent parameters that will be **learned** during the training of the transformer architecture.

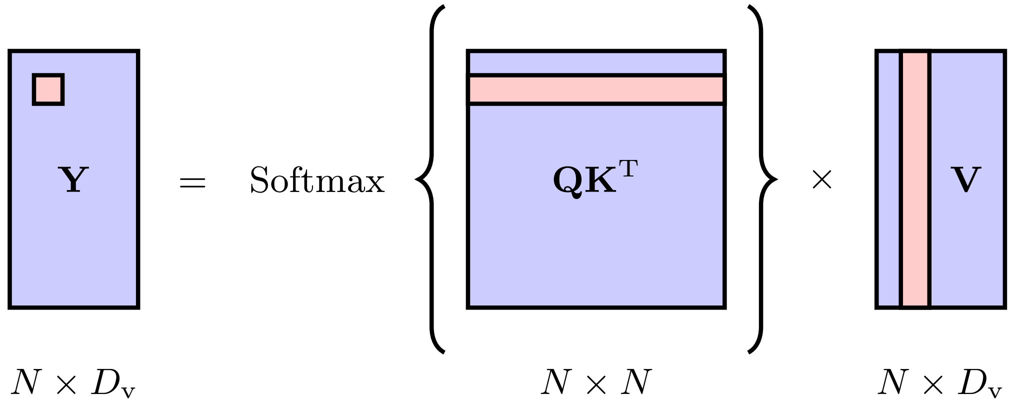
- $\mathbf{W}^{(q)}, \mathbf{W}^{(k)}, \mathbf{W}^{(v)}$ are matrices of dim. $D \times D_k, D \times D_q, D \times D_v$. Setting $D_k = D_q$ allows for dot-products between query and key while $D_v = D$ allows multiple transformer layers to be stacked. We set $D_k = D_q = D_v = D$.

- Note: E.g. in GPT-3, the embedding dimension is divided among multiple attention heads but the overall dimensional consistency is maintained.

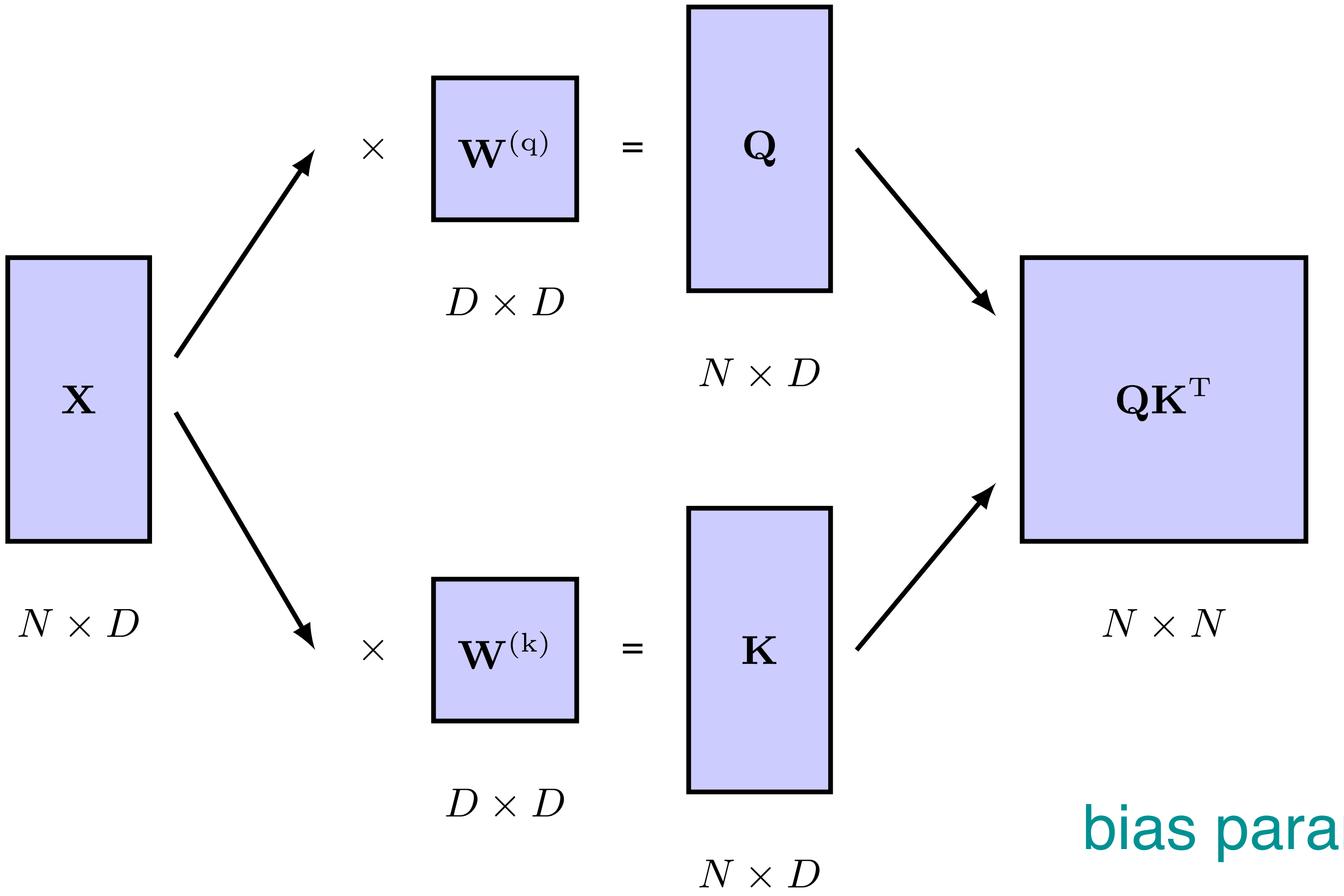
Network Parameters

- The transformation is now generalized to:

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{Q}\mathbf{K}^{\mathrm{T}}\right]\mathbf{V}$$



whereas the dot-product can be computed by:



bias parameters are implicit

Algorithm 12.1: Scaled dot-product self-attention

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Weight matrices $\{\mathbf{W}^{(\mathrm{q})}, \mathbf{W}^{(\mathrm{k})}\} \in \mathbb{R}^{D \times D_\mathrm{k}}$ and $\mathbf{W}^{(\mathrm{v})} \in \mathbb{R}^{D \times D_\mathrm{v}}$

Output: $\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \in \mathbb{R}^{N \times D_\mathrm{v}} : \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$

$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathrm{q})}$ // compute queries $\mathbf{Q} \in \mathbb{R}^{N \times D_\mathrm{k}}$

$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathrm{k})}$ // compute keys $\mathbf{K} \in \mathbb{R}^{N \times D_\mathrm{k}}$

$\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathrm{v})}$ // compute values $\mathbf{V} \in \mathbb{R}^{N \times D}$

return $\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^\mathrm{T}}{\sqrt{D_\mathrm{k}}}\right]\mathbf{V}$

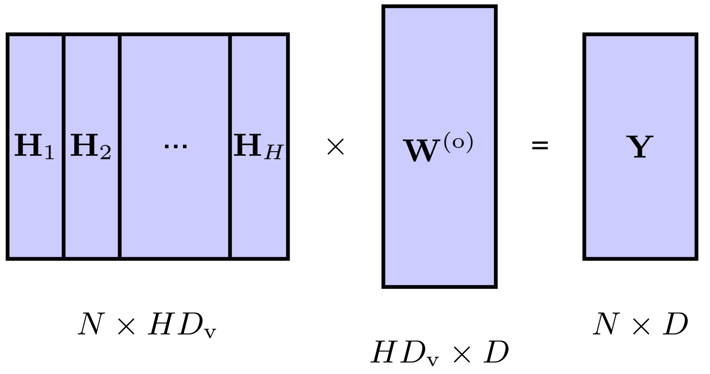
Multi-head attention

- There might be multiple patterns of attention relevant at the same time, e.g., some associated with tenses, some with vocabulary.
- Single “attention head” averages out these effects. Instead use multiple attention heads in parallel; analogous to channels in CNN.
- Suppose we have H heads indexed by $h = 1,\ldots,H$:

$$\mathbf{H}_h = \mathrm{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$$

- The heads are concatenated into a single matrix, and the result is then linearly transformed to give a combined output:

$$\mathbf{Y}(\mathbf{X}) = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right] \mathbf{W}^{(\mathrm{o})}$$



- The matrix $\mathbf{W}^{(\mathrm{o})}$ is learned along with the weight matrices $\mathbf{W}^{(q)}, \mathbf{W}^{(k)}, \mathbf{W}^{(v)}$.

Algorithm 12.2: Multi-head attention

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$

Query weight matrices $\{\mathbf{W}_1^{(\mathrm{q})}, \ldots, \mathbf{W}_H^{(\mathrm{q})}\} \in \mathbb{R}^{D \times D}$

Key weight matrices $\{\mathbf{W}_1^{(\mathrm{k})}, \ldots, \mathbf{W}_H^{(\mathrm{k})}\} \in \mathbb{R}^{D \times D}$

Value weight matrices $\{\mathbf{W}_1^{(\mathrm{v})}, \ldots, \mathbf{W}_H^{(\mathrm{v})}\} \in \mathbb{R}^{D \times D_\mathrm{v}}$

Output weight matrix $\mathbf{W}^{(\mathrm{o})} \in \mathbb{R}^{HD_\mathrm{v} \times D}$

Output: $\mathbf{Y} \in \mathbb{R}^{N \times D} : \{\mathbf{y}_1, \ldots, \mathbf{x}_N\}$

`// compute self-attention for each head (Algorithm 12.1)`

for $h = 1, \ldots, H$ **do**

$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})}, \quad \mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})}, \quad \mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}$

$\mathbf{H}_h = \mathrm{Attention}\left(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h\right)$ `//` $\mathbf{H}_h \in \mathbb{R}^{N \times D_\mathrm{v}}$

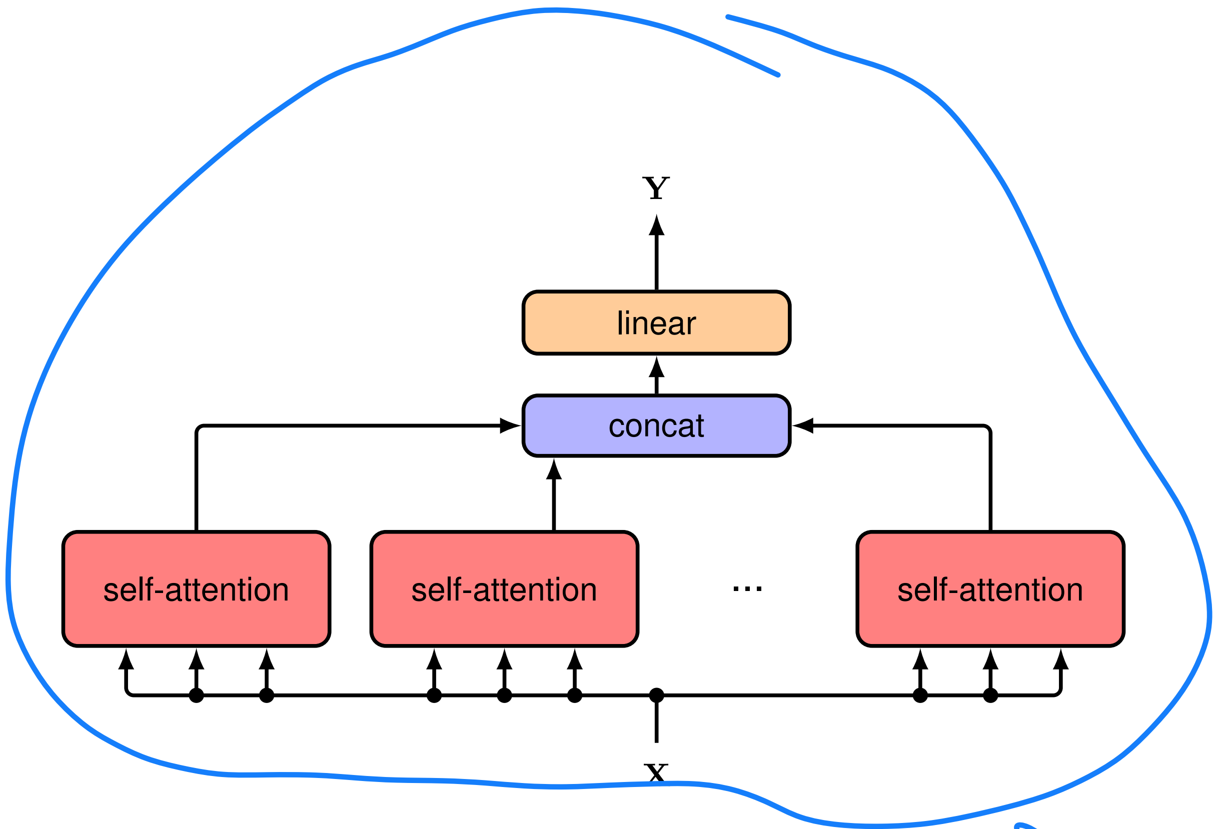
end for

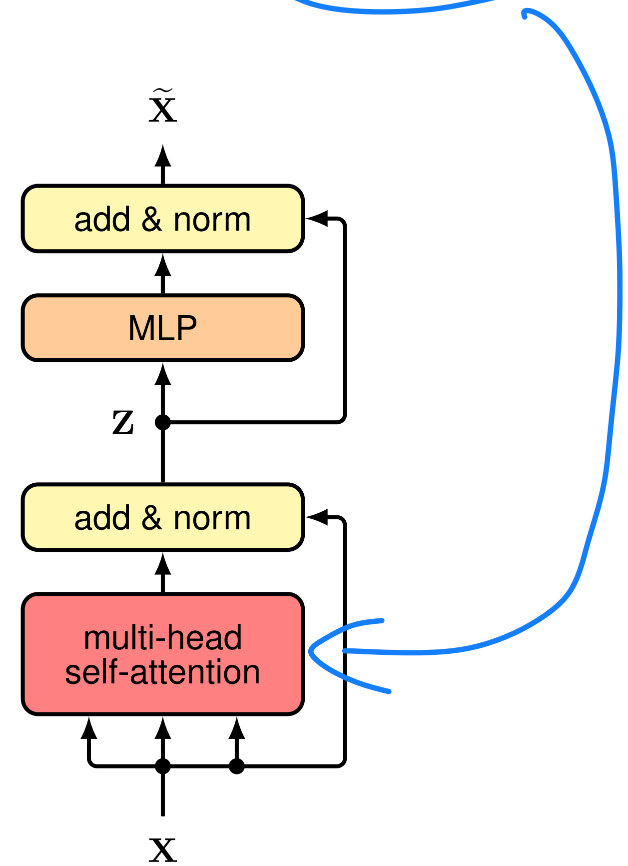
$\mathbf{H} = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_\mathbf{N}\right]$ `// concatenate heads`

return $\mathbf{Y}(\mathbf{X}) = \mathbf{H}\mathbf{W}^{(\mathrm{o})}$

Transformer Layers

- NNs benefit greatly from depth, so we can stack self-attention layers (like the right) on top of each other.
- To improve efficiency, transformer layers are followed by **layer normalization**: https://arxiv.org/abs/1607.06450
- Output of an attention layer are constrained to be linear combinations of the inputs, though non-linearities enter through the attention weights.
- Enhance flexibility by post-processing the output of each layer using non-linear network denoted by MLP (same for each vector).





Algorithm 12.3: Transformer layer

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Multi-head self-attention layer parameters
Feed-forward network parameters
Output: $\widetilde{\mathbf{X}} \in \mathbb{R}^{N \times D} : \{\widetilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_N\}$

$\mathbf{Z} = \text{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right]$ `// Y(X) from Algorithm 12.2`
$\widetilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}\left[\mathbf{Z}\right] + \mathbf{Z}\right]$ `// shared neural network`
return $\widetilde{\mathbf{X}}$

Transformers

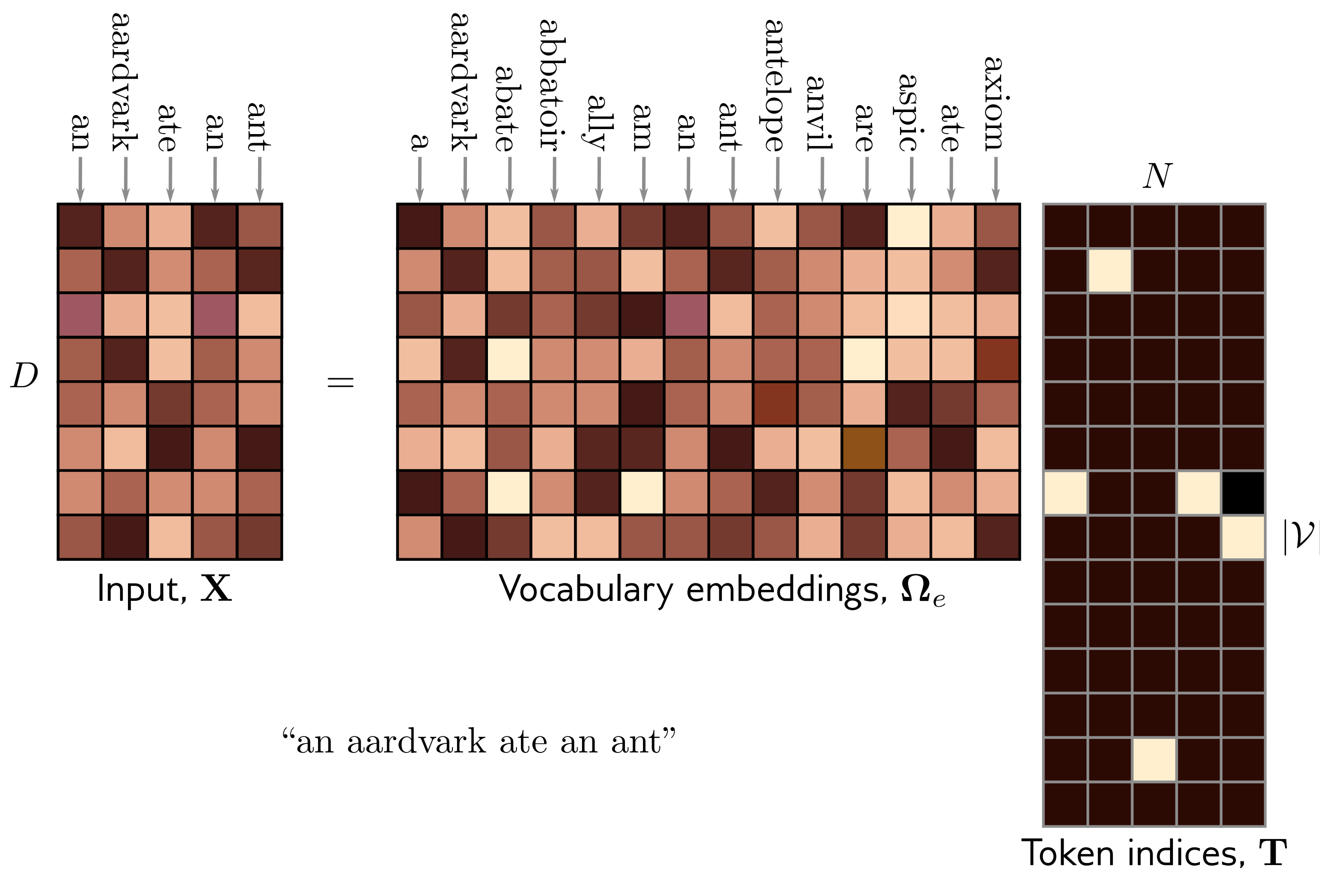
Large Language Models

Transformer for NLP

- A typical NLP pipeline starts with a **tokenizer** that splits the text into words or word fragments.
- Then each of the tokens is mapped to a learned **embedding**.
    - The whole vocabulary is stored in a matrix $\Omega_e \in \mathbb{R}^{D \times |\mathcal{V}|}$ where $|\mathcal{V}|$ is the vocabulary size; this vocabulary matrix is learned.
- These embeddings are passed through a series of **transformer layers**.

Learned Embeddings

- Each token is mapped to a unique word embedding; the embeddings for the whole vocabulary are storied in a matrix $\Omega_e \in \mathbb{R}^{D \times |\mathscr{V}|}$



$$\mathbf{X} = \boldsymbol{\Omega}_e \mathbf{T}$$

- The matrix Ω_e can be learned like any other network parameter.

- A typical embedding size D is 1024 and a typical total vocabulary size $|\mathscr{V}|$ is $30{,}000$. Many parameters in Ω_e to learn.

Transformer Encoders and Decoders

- The embedding matrix $\mathbf{X}$ representing the text is passed through a series of K transformer layers, called a **transformer model**.
- Three types of transformer models:
  - An **encoder** transforms the text embeddings into a representation that can support a variety of tasks (e.g., sentiment analysis).
  - A **decoder** predicts the next token to continue the input text.
  - **Encoder-decoder** used in sequence-to-sequence tasks, where one text string is converted into another, e.g., machine translation.
- A hands-on tutorial on transformers in pytorch can be found here: https://peterbloem.nl/blog/transformers

Transformers

Large Language Models - Encoders

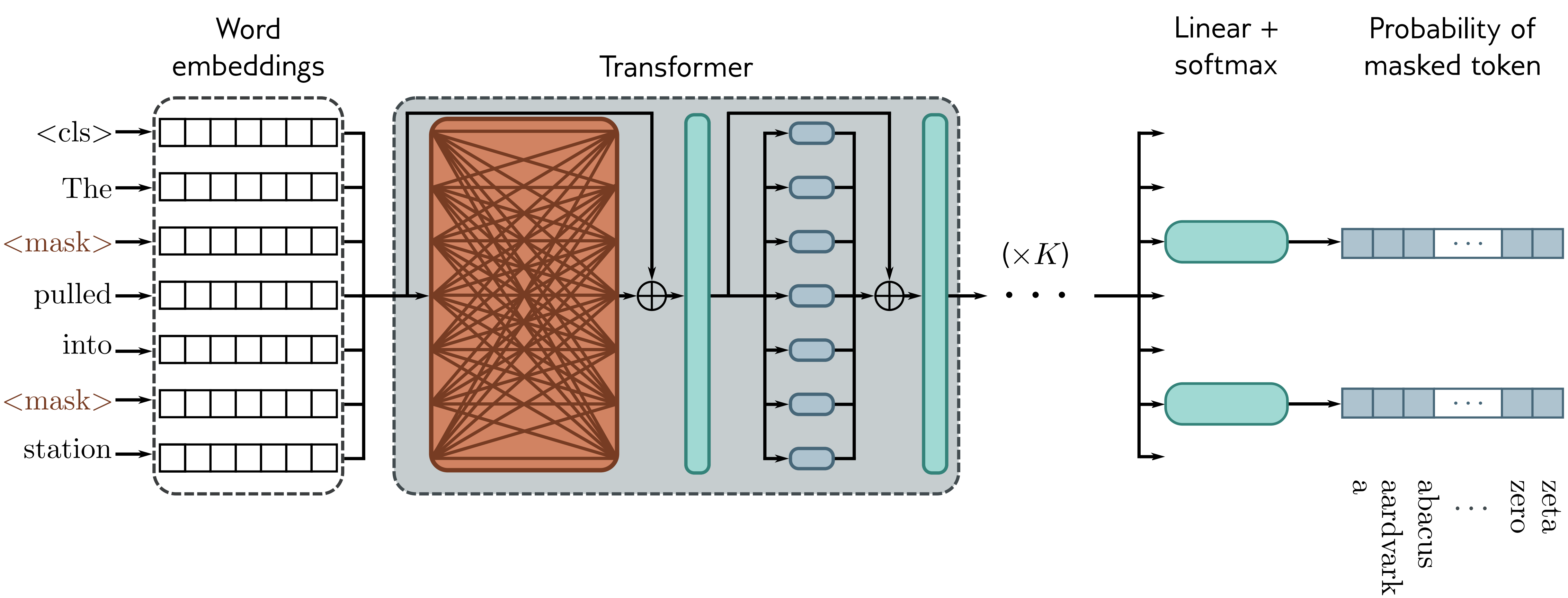
Encoder model example: BERT

https://arxiv.org/abs/1810.04805v2

- BERT is an encoder model that uses a vocabulary of 30,000 tokens.
- Input tokens are converted to 1024 dimensional word embeddings and passed through 24 transformer layers.
- Each contains a self-attention mechanism with 16 heads.
- The weight matrices Q_h, K_h, V_h for each head are 1024×64.
- The total number of parameters is ~ 340 million, but it is now much smaller than state-of-the-art models.
- Encoder models like BERT exploit **transfer learning**: parameters of the ML model are learned during *pre-training* using *self-supervision* from a large corpus of data, followed by a *fine-tuning* stage to adapt for specific task using a smaller body of *supervised training data*.

Pre-training

- For BERT, the self-supervision task consists of predicting missing words from sentences from a large internet corpus.

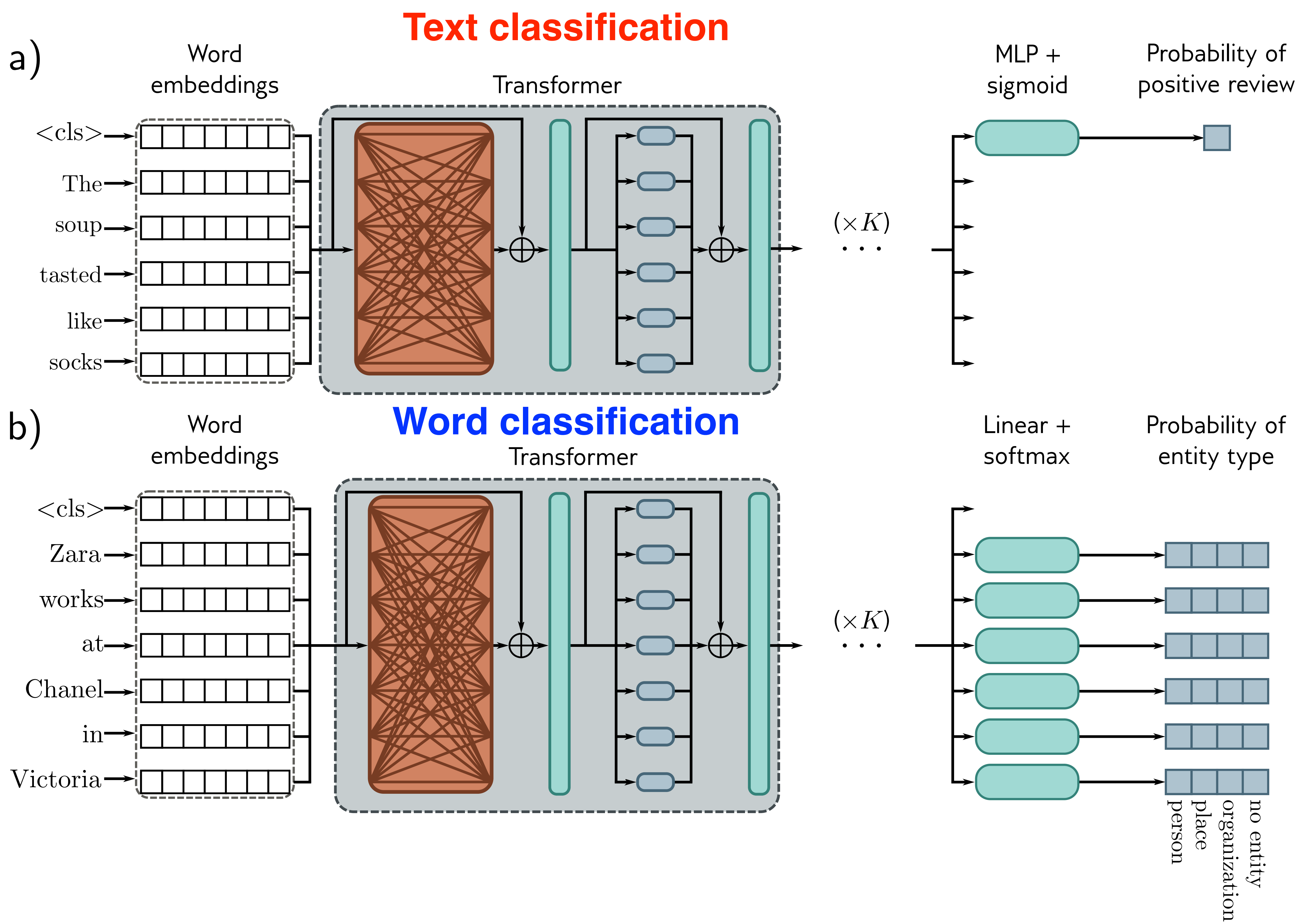


- Predicting missing words forces the transformer model to understand some syntax. For example, red is often found before car or dress than swim. In the above example, train is more likely than lasagna.

Fine-tuning

- In the fine-tuning stage, the model parameters are adjusted to specialize the network to a particular task.
- An extra layer is appended onto the transformer network to convert the output vectors to the desired output format.
- Specific tasks include:
  - **Text classification**: <cls> token is added to the start of each string during pre-training. sentiment analysis, the vector associated with <cls> is mapped to a number & passed through a logistic sigmoid.
  - **Word classification**: e.g., to classify a word into entity types (person, place, organization, or no-entry). Input is mapped to a $E \times 1$ vector where $E =$ entry types, then Softmax for probabilities.

Fine-tuning



Text classification
a)
Word embeddings
Transformer
MLP + sigmoid
Probability of positive review
<cls>
The
soup
tasted
like
socks
$(\times K)$
Word classification
b)
Word embeddings
Transformer
Linear + softmax
Probability of entity type
<cls>
Zara
works
at
Chanel
in
Victoria
$(\times K)$
person
place
organization
no entity

Transformers

Large Language Models - Decoders

Autoregressive text generation

- The basic architecture is similar to the encoder model & comprises a series of transformer layers that operate on learned word embeddings.
- Different goal: to generate the next token in a sequence (and generate a coherent text passage by feeding the sequence back into the model).
- **Autoregressive language model**: factors the joint probability of a sequence of observed tokens into an autoregressive sequence.
- Consider e.g.: "It takes great courage to let yourself appear weak."

$$
\begin{aligned}
Pr(\text{It takes great courage to let yourself appear weak}) \quad = \\
& Pr(\text{It}) \times Pr(\text{takes}|\text{It}) \times Pr(\text{great}|\text{It takes}) \times Pr(\text{courage}|\text{It takes great}) \times \\
& Pr(\text{to}|\text{It takes great courage}) \times Pr(\text{let}|\text{It takes great courage to}) \times \\
& Pr(\text{yourself}|\text{It takes great courage to let}) \times \\
& Pr(\text{appear}|\text{It takes great courage to let yourself}) \times \\
& Pr(\text{weak}|\text{It takes great courage to let yourself appear}).
\end{aligned}
$$

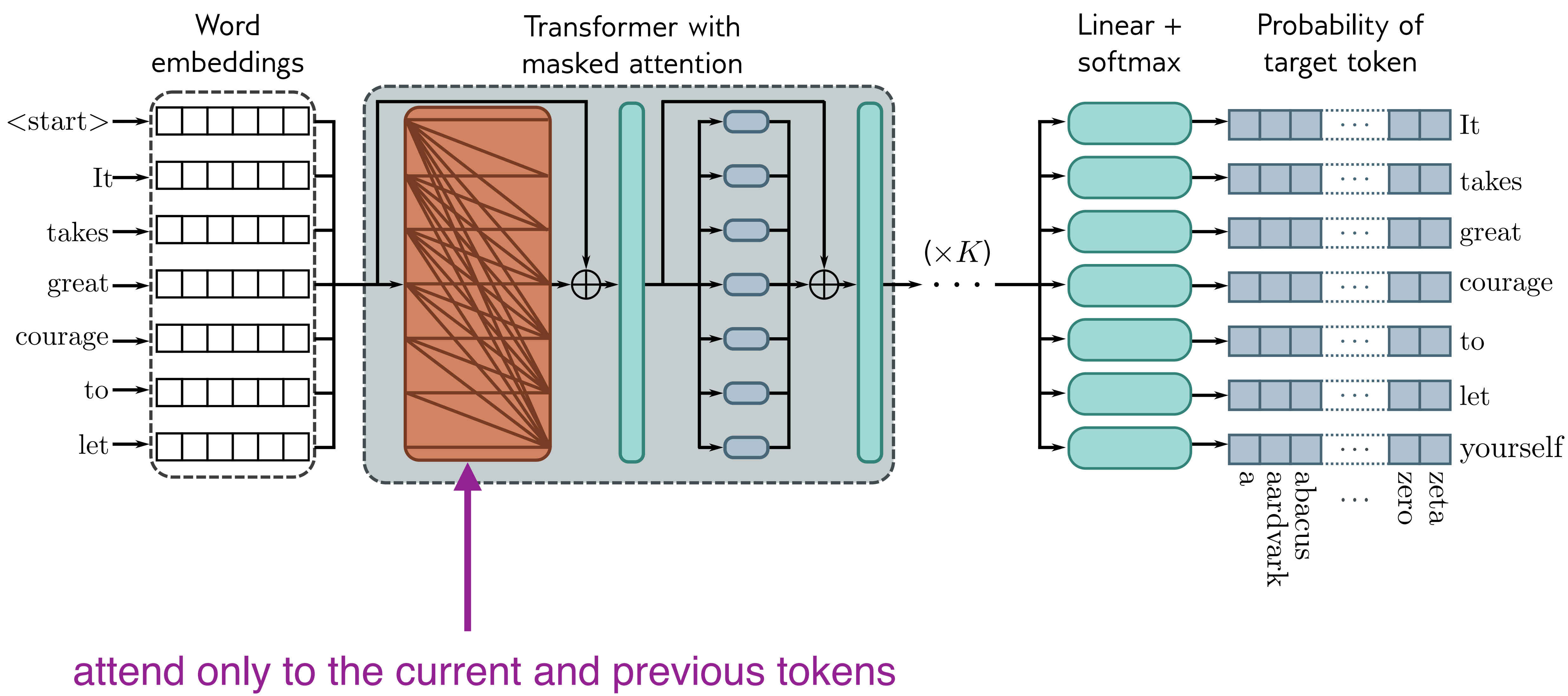
Generally:

$$
Pr(t_1, t_2, \ldots, t_N) = Pr(t_1) \prod_{n=2}^{N} Pr(t_n | t_1, \ldots, t_{n-1}).
$$

Decoder model example: GPT3

- To train a decoder, we maximize the log probability of the input text under the autoregressive model defined above.
- This poses a problem: if we pass the full sentence, the term computing $\log | Pr(\text{great} | \text{It takes})$ has access to the rest of the sentence.
- The system can cheat rather than learn to predict, and thus will not train properly.
- **Masked self-attention**: setting the dot products with future tokens in the self-attention computation to $-\infty$ before passing through softmax.
- The transformer layers use masked self-attention so that only attention to the current and previous tokens are allowed.
- During training, we aim to maximize the sum of the log probabilities of the next token using a standard multiclass cross-entropy loss.

Masked self-attention



Sampling: Generating text from a decoder

- The autoregressive language model is a **generative model**.
- Start with an input sequence of text, beginning with a <start> token.
- The outputs are the probabilities over possible subsequent tokens. **We can either pick the most likely token or sample from this probability distribution**.
- The new extended sequence can be fed back into the decoder network that outputs the probability distribution over the next token.
- At each step, **the decoder takes the entire sequence generated so far (including all previous tokens) as input and produces a probability distribution for the next token**. Then, you typically select one token from that distribution and append it to the sequence. This updated sequence, now containing the newly generated token, is then fed back into the decoder for the next prediction.
- Other strategies (instead of greedy search): **beam search and top-k sampling**, etc.

Transformers

Large Language Models -
Encoder-Decoder
Transformer (briefly)

Encoder-decoder model example: machine translation

- Translation between languages is a **sequence-to-sequence** task.
- **An encoder computes a good representation of the source sentence. A decoder generates the sentence in the target language**.
- Consider a **encoder-decoder model for English-French** translation.
  - The encoder receives the sentence in English and process it through a series of transformer layers to create an output rep. for each token.
  - During training, the decoder receives the ground truth translation in French and passes it through a series of transformer layers that use masked self-attention and predict the following word at each position.
  - However, the decoder layers also attend to the output of the encoder. Each French output word is conditioned on the previous output words and the source English sentence.
- This is the **original setup which invented the transformer.**

https://arxiv.org/abs/1706.03762

NeurIPS
https://papers.neurips.cc › paper › 7181-attention... PDF

Attention is All you Need

by A Vaswani · Cited by 171064 — We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely.
11 pages

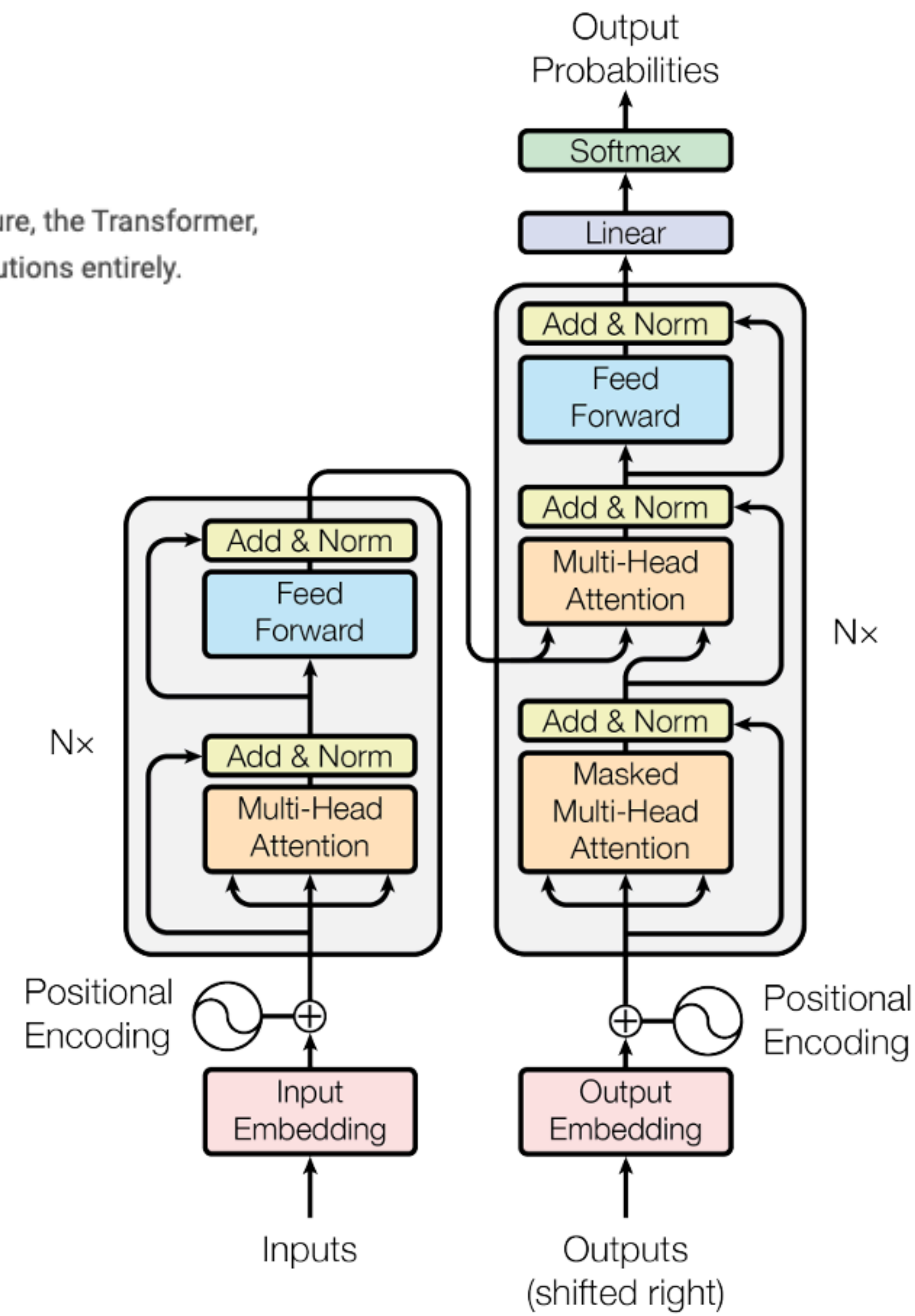


Figure 1: The Transformer - model architecture.

Transformers

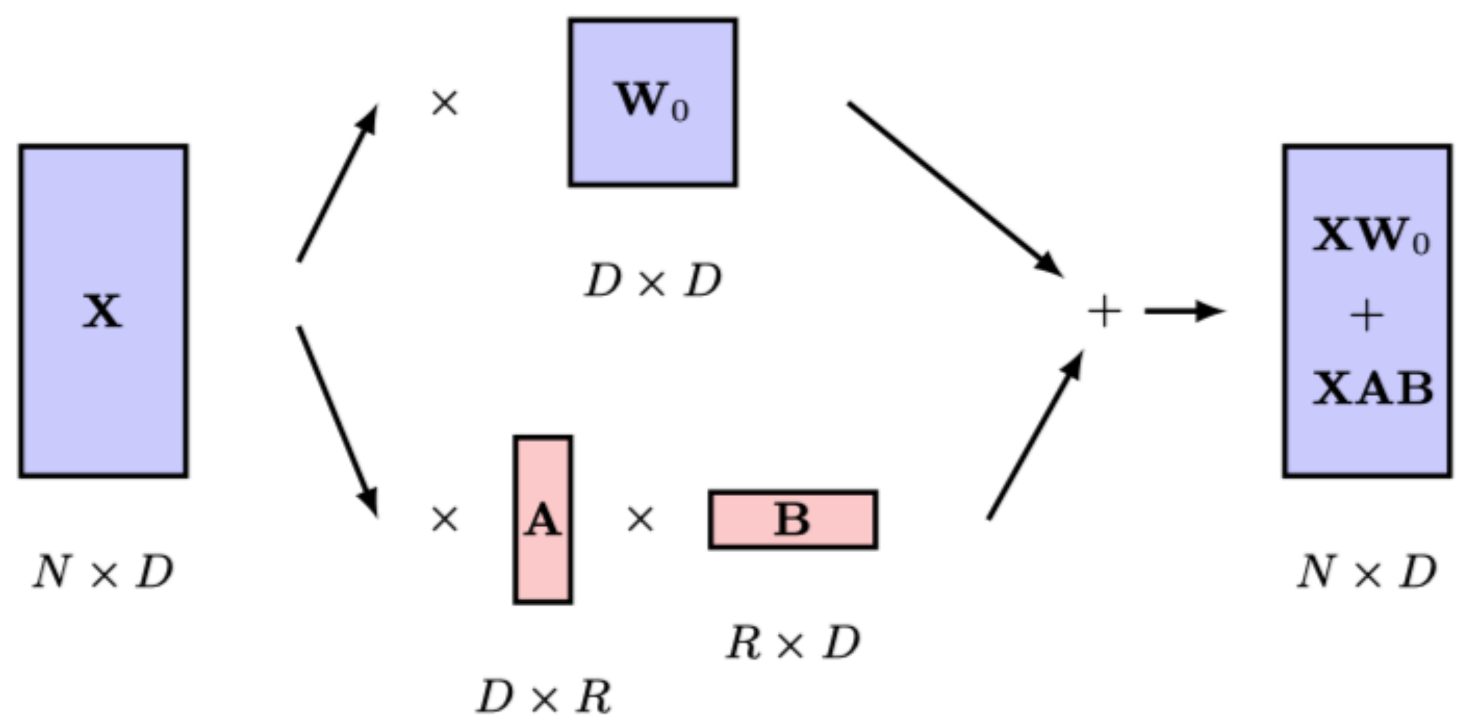
Large Language Models - Fine Tuning

Fine Tuning Foundation Models

- A pre-trained LLM (or large transformer) is also called a **“Foundation Model”**.
- We can then use supervised fine tuning for specific applications, often called **“downstream tasks”**.
- The fine-tuning **can be done by adding extra layers to the outputs of the network or by replacing the last few layers** with fresh parameters and then using the labelled data to train these final layers.
- During the fine-tuning stage, **the weights and biases in the main model can either be left unchanged or be allowed to undergo small levels of adaptation.** Typically the cost of the fine-tuning is small compared to that of pretraining.

Fine Tuning with LoRA

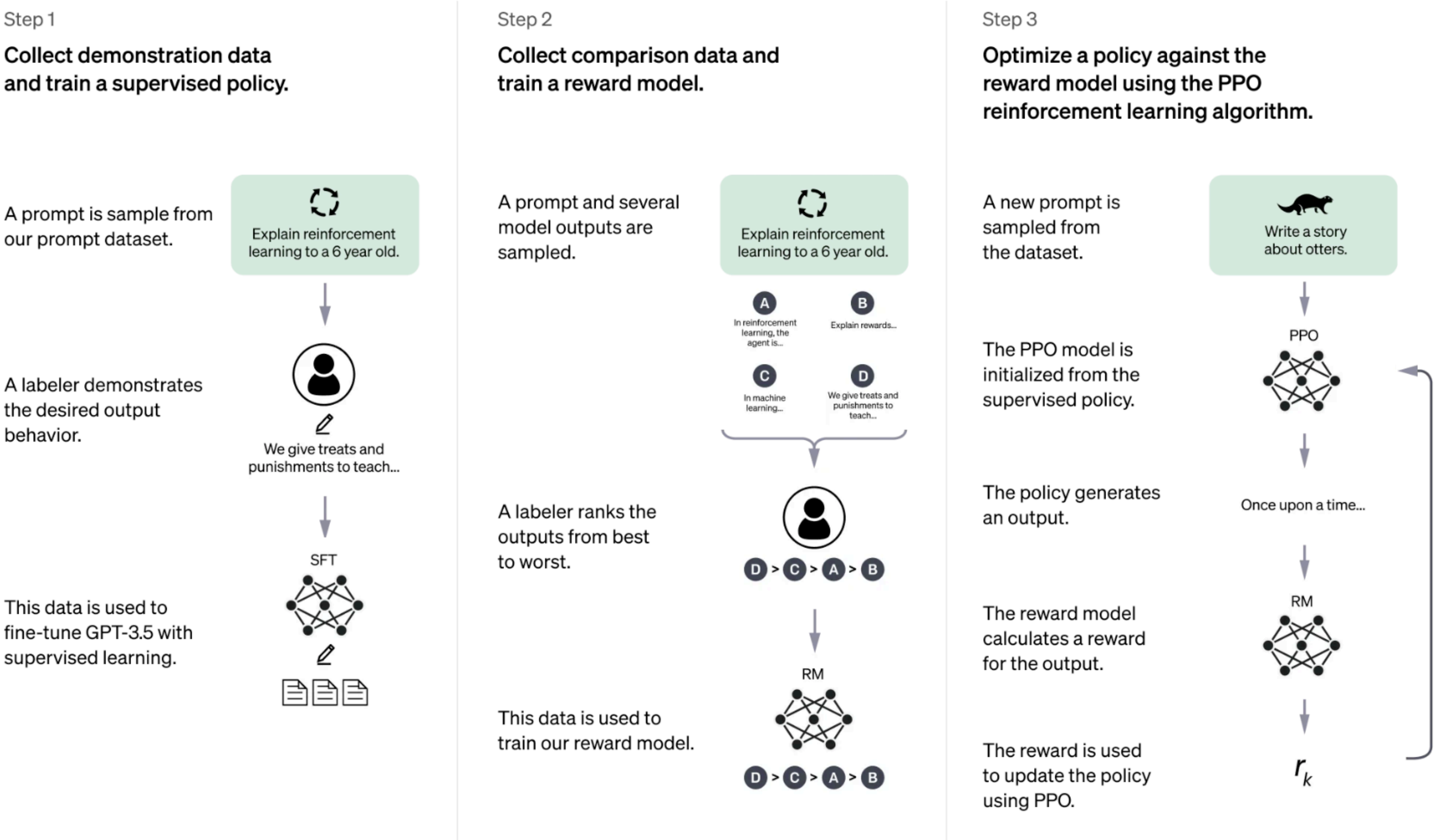
- One very efficient approach to fine-tuning is called **low-rank adaptation or LoRA** (Hu et al., 2021). This approach is inspired by results which show that a trained overparameterized model has a low intrinsic dimensionality with respect to fine-tuning.
- LoRa exploits this by freezing the weights of the original model and adding additional learnable weight matrices into each layer of the transformer in the form of low-rank products.



Schematic illustration low-rank adaptation showing a weight matrix $\mathbf{W}_0$ from one of the attention layers in a pre-trained transformer. Additional weights given by matrices $\mathbf{A}$ and $\mathbf{B}$ are adapted during fine-tuning and their product $\mathbf{AB}$ is then added to the original matrix for subsequent inference.

Fine Tuning and RLHF

- After training a decoder model model will “babble” text, trying to complete sequences. e.g. if you give it a question it might follow up with more questions.
- Chat Bots are then fine tuned in several steps to make them more useful:
- https://openai.com/index/chatgpt/
  - Step 1: Fine tuning
    - They have thousands of question and answer pairs in a curated data set
  - Step 2: Humans rank different answers (Reinforcement Learning with Human Feedback RLHF). Train a reward model.
  - Step 3: Use Reinforcement Learning using this reward model.
- Roughly, this brings the model **from a “document completer” to a “question answerer”**



We will talk more about Reinforcement Learning later.

Transformers

Using LLMs

LLM Model Zoo

- https://lmarena.ai/

🏆 Chatbot Arena LLM Leaderboard: Community-driven Evaluation for Best LLM and AI chatbots

Discord | Twitter | 小红书 | Blog | GitHub | Paper | Dataset | Kaggle Competition

Chatbot Arena is an open platform for crowdsourced AI benchmarking, developed by researchers at UC Berkeley SkyLab and LMArena. With over 1,000,000 user votes, the platform ranks best LLM and AI chatbots using the Bradley-Terry model to generate live leaderboards. For technical details, check out our paper.

Chatbot Arena thrives on community engagement — cast your vote to help improve AI evaluation!

New Launch! WebDev Arena: web.lmarena.ai - AI Battle to build the best website!

Language | Overview | Price Analysis | WebDev Arena | Vision | Text-to-Image | Copilot Arena Leaderboard | Arena-Hard-Auto

Total #models: **212**. Total #votes: **2,768,389**. Last updated: 2025-03-10.

Code to recreate leaderboard tables and plots in this notebook. You can contribute your vote at lmarena.ai!

Category: Overall

Apply filter: Style Control | Show Deprecated

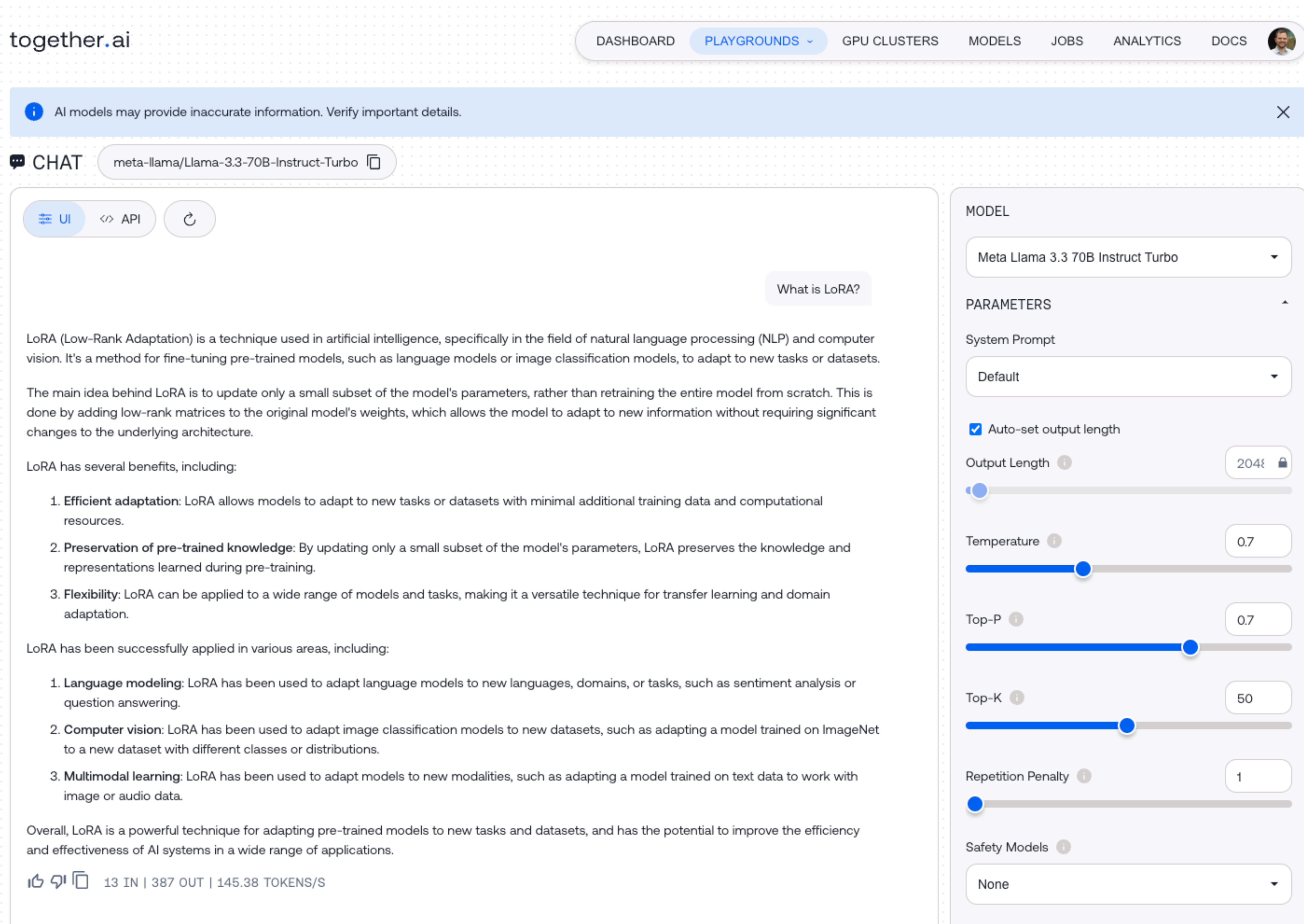
Overall Questions

#models: 212 (100%) #votes: 2,768,389 (100%)

Rank* (UB)	Rank (StyleCtrl)	Model	Arena Score	95% CI	Votes	Organization	License
1	2	Grok-3-Preview-02-24	1407	+7/-7	7580	xAI	Proprietary
1	1	GPT-4.5-Preview	1404	+7/-9	6024	OpenAI	Proprietary
3	6	Gemini-2.0-Flash-Thinking-Exp-01-21	1384	+5/-5	19837	Google	Proprietary
3	3	Gemini-2.0-Pro-Exp-02-05	1380	+4/-4	17695	Google	Proprietary
3	2	ChatGPT-4o-latest (2025-01-29)	1375	+4/-5	19587	OpenAI	Proprietary
6	4	DeepSeek-R1	1361	+5/-6	10474	DeepSeek	MIT
6	10	Gemini-2.0-Flash-001	1355	+4/-5	15416	Google	Proprietary
6	3	o1-2024-12-17	1353	+4/-4	22010	OpenAI	Proprietary
9	10	Gemma-3-27B-it	1339	+9/-11	3870	Google	Gemma
9	10	Qwen2.5-Max	1338	+5/-5	14258	Alibaba	Proprietary
9	7	o1-preview	1335	+4/-4	33195	OpenAI	Proprietary
9	10	o3-mini-high	1328	+6/-5	11409	OpenAI	Proprietary
13	12	DeepSeek-V3	1319	+4/-4	23079	DeepSeek	DeepSeek

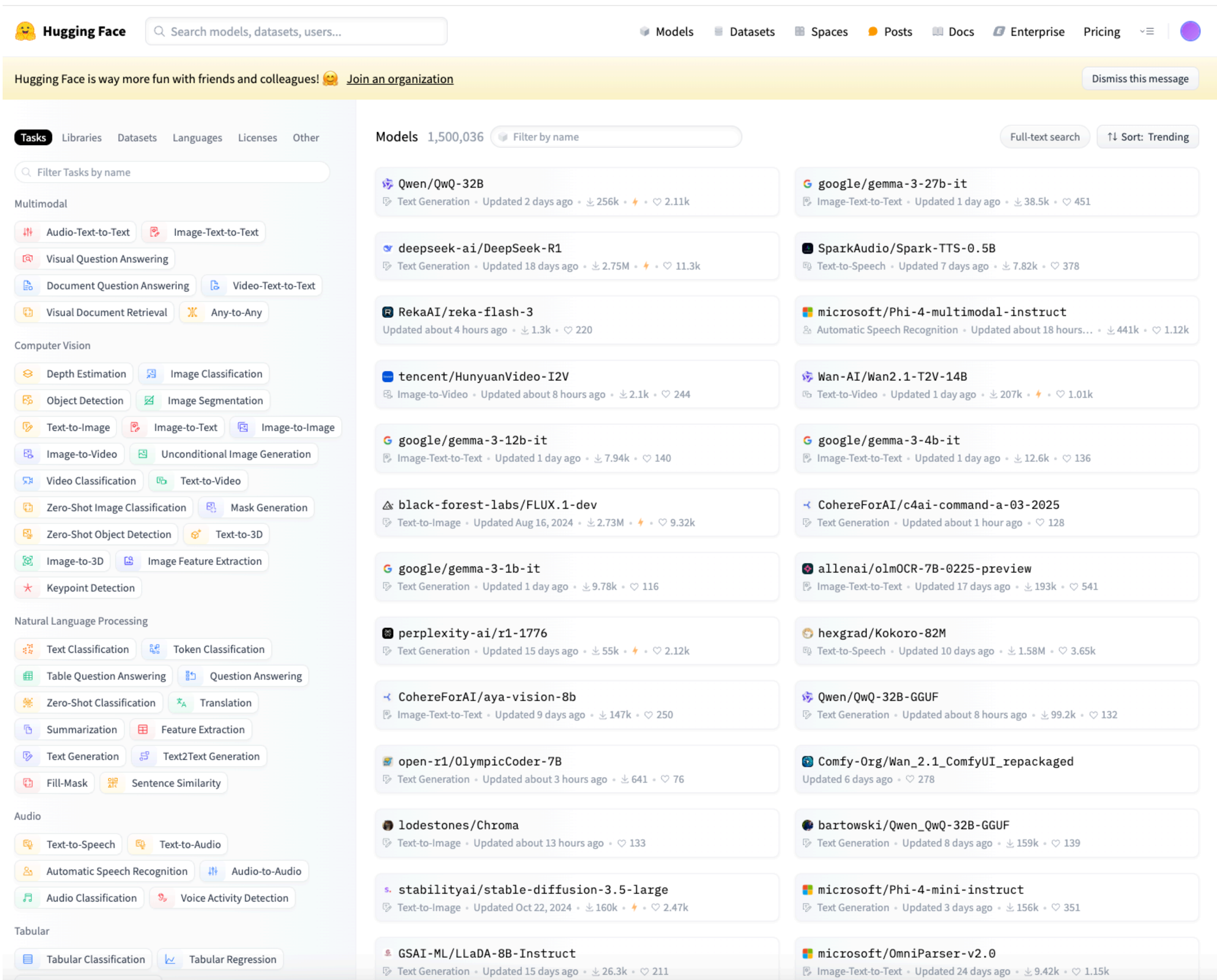
Running them online

- Example: https://www.together.ai/



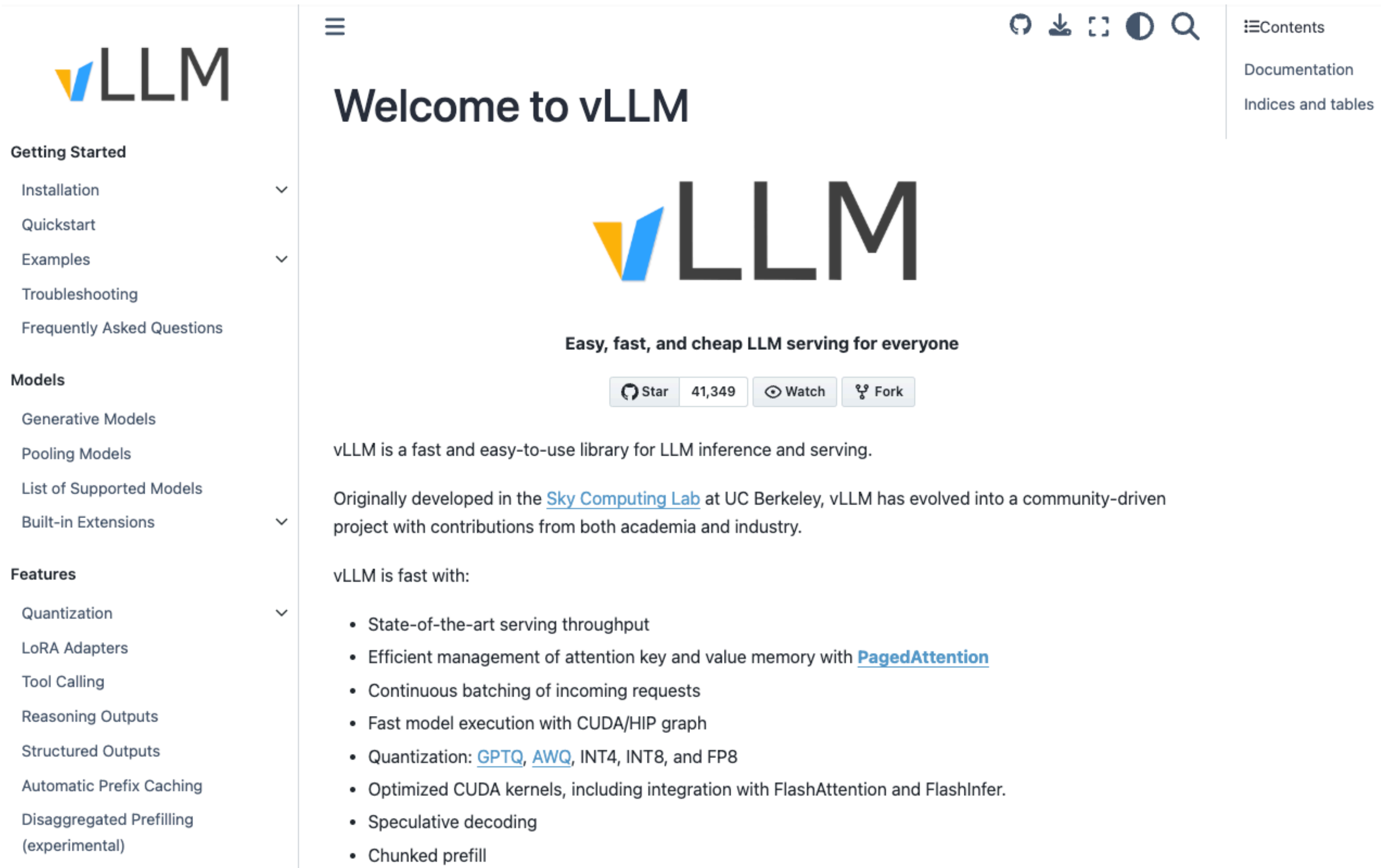
Running them locally

- Large Library of models and data sets: https://huggingface.co/



Running them locally

- My students use VLLM https://docs.vllm.ai/en/latest/



Quantization

- Models have large GPU memory requirements.
- For example, on the 24GB memory GPUs in my group we can run (inference, not training) 7B Llama models.
  - That's because with 2 byte (16bit) precision the required memory for 7B parameters is 14 GB (roughly speaking).
  - We also need some memory for the "**KV Cache**", which depends on the sequence length.
- One can compress these models using a technique called "quantization". This decreases their performance somewhat.
- **Quantization in large language models (LLMs) is a technique used to reduce the memory footprint and computational cost** of inference by representing model parameters with lower-precision data types, such as 8-bit integers (INT8) or even lower, instead of the **standard 16-bit** (FP16) or 32-bit floating-point (FP32) representations.
- In this way we can use models that are two or four times larger.

Prompt Engineering

- Prompt engineering is **the practice of crafting effective prompts to guide AI models**, such as large language models (LLMs).
- It involves **structuring inputs in a way that optimizes the model's performance** for specific tasks, such as text generation, code writing, or problem-solving.
- Example: **Chain-of-Thought (CoT) Prompting** – Encouraging step-by-step reasoning improves logical accuracy.
- https://arxiv.org/abs/2201.11903 Chain-of-Thought Prompting Elicits Reasoning in Large Language Models
- Some people joke that AI research has been reduced to prompt engineering.

Chain-of-Thought Prompting

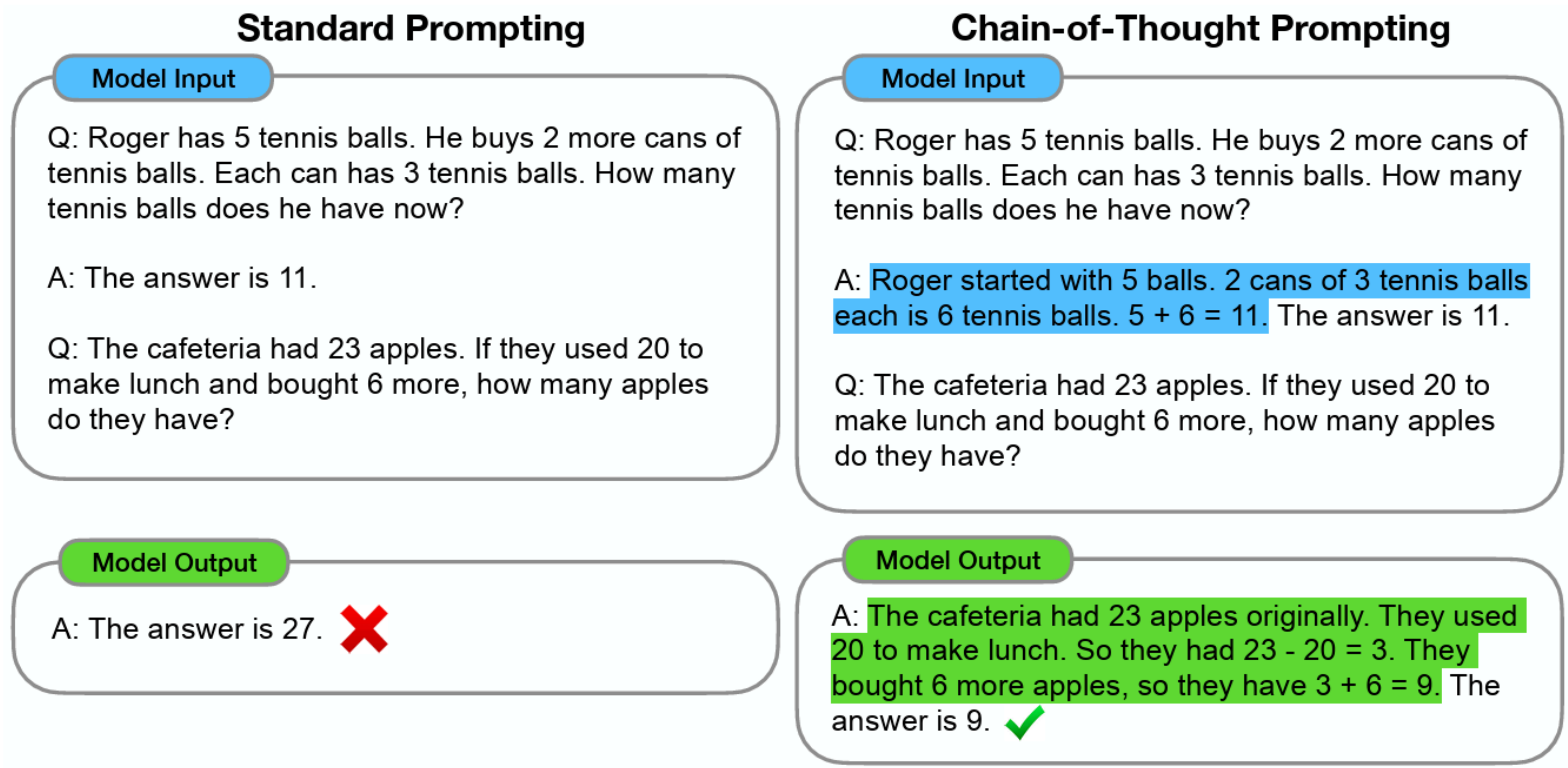


Figure 1: Chain-of-thought prompting enables large language models to tackle complex arithmetic, commonsense, and symbolic reasoning tasks. Chain-of-thought reasoning processes are highlighted.

Few-Shot Learning

- **Few shot learning** - The model is given a few examples of input-output pairs before making a prediction.
  - Few-shot learning enables models to adapt to new tasks without extensive retraining.
- **One-Shot Learning** – A special case of few-shot learning where only one example is provided.
- **Zero-Shot Learning** – The model makes predictions without any examples, relying solely on prior knowledge.

- https://arxiv.org/abs/2005.14165 Language Models are Few-Shot Learners

Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
Translate English to French:        ← task description
cheese => ..................        ← prompt
```

One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
Translate English to French:        ← task description
sea otter => loutre de mer          ← example
cheese => ..................        ← prompt
```

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
Translate English to French:        ← task description
sea otter => loutre de mer          ← examples
peppermint => menthe poivrée
plush girafe => girafe peluche
cheese => ..................        ← prompt
```

Multi-Agent Frameworks

- Many works explore combining LLMs to solve tasks.
- An example of this line of research is
  - https://arxiv.org/abs/2409.15254 Archon: An Architecture Search Framework for Inference-Time Techniques

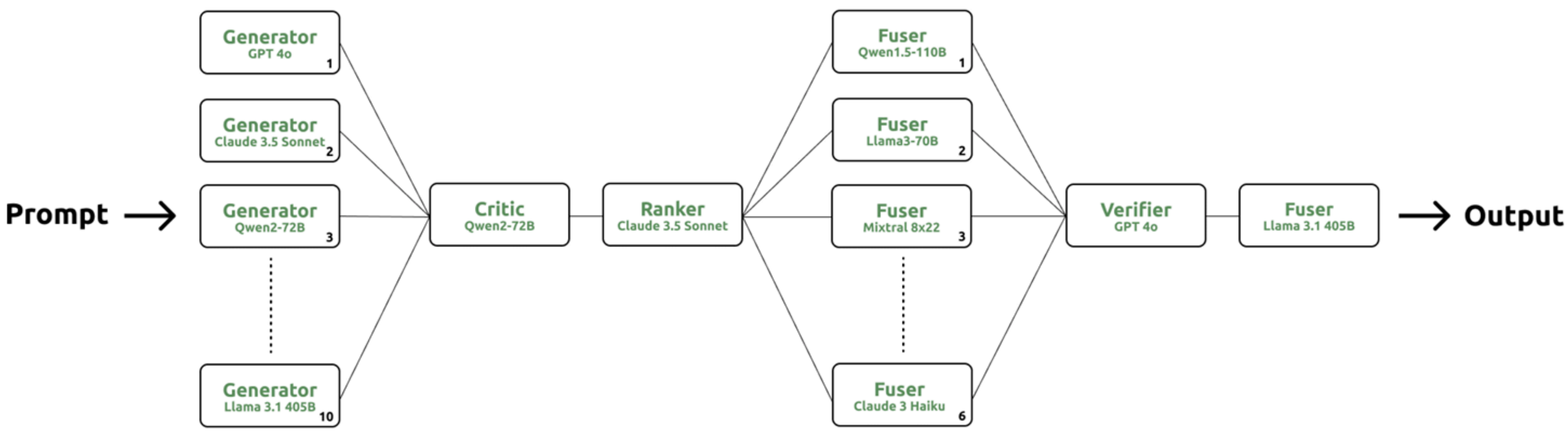


Figure 2: **Example ARCHON Architecture**: The architecture starts with ten generator models, followed by a critic model, a ranker model, one layer of six fuser models, a verifier model, and finishing with a fuser model.

Course logistics

- **Reading for this lecture:**
  - This lecture was based in part on the books by Bishop and Prince, linked on the website. Many figures were taken from these books.