LLMs from Dummies: Part 3

Running our models on an EC2 instance

1. Select EC2 instance

Example: Run a “Lit-Llama” model (7B parameters)

Single GPU instances

Instance type	GPU	GPU Ram (GB)	Cost ($/hour)
p3.2xlarge	V100	16	$3
g5.2xlarge	A10G	24	$1.2
g4dn.2xlarge	T4	16	$0.72
g5dn.2xlarge	T4G	16	$1

1. Select EC2 instance

2. Install CUDA drivers

```
$ sudo apt-get install linux-headers-$(uname -r)

# Install the CUDA repository public GPG key.
distribution=$(. /etc/os-release;echo $ID$VERSION_ID | sed -e 's/\.//g')
wget https://developer.download.nvidia.com/compute/cuda/repos/$distribution/x86_64/cuda-
keyring_1.0-1_all.deb

dpkg -i cuda-keyring_1.0-1_all.deb

# Update the APT repository cache and install the driver using the cuda-drivers meta-
package.

apt-get update
apt-get -y install cuda-drivers




# Check CUDA drivers installation

# Install pip
apt install python3-pip

# Torch
pip3 install torch torchvision torchaudio

# Check if cuda enabled. Should output "True"
python3 -c "import torch; print(torch.cuda.is_available())"
```

1. Select EC2 instance
2. Install CUDA drivers
3. Install Llama model

```
# Install Lit-LLama repo
git clone https://github.com/Lightning-AI/lit-llama

# Install required packages
cd lit-llama/
pip install -r requirements.txt




# Install Git-lfs
apt install git-lfs
git-lfs install


# Get “final release” weights
git clone https://huggingface.co/openlm-research/open_llama_7b data/checkpoints/open-llama/7B

# Convert weights (~5 min)
time python3 scripts/convert_hf_checkpoint.py \
     --checkpoint_dir checkpoints/open-llama/7B/ \
     --model_size 7B
```

1. Select EC2 instance

2. Install CUDA drivers

3. Install Llama model

4. Test that the model works

```
# Generate. This requires ~14GB of GPU memory.
# Weights are converted to bpfloat16
# Time ~3 minutes

time python generate.py --prompt "Hello, my name is"
```

Transformer: Encoder & Decoder

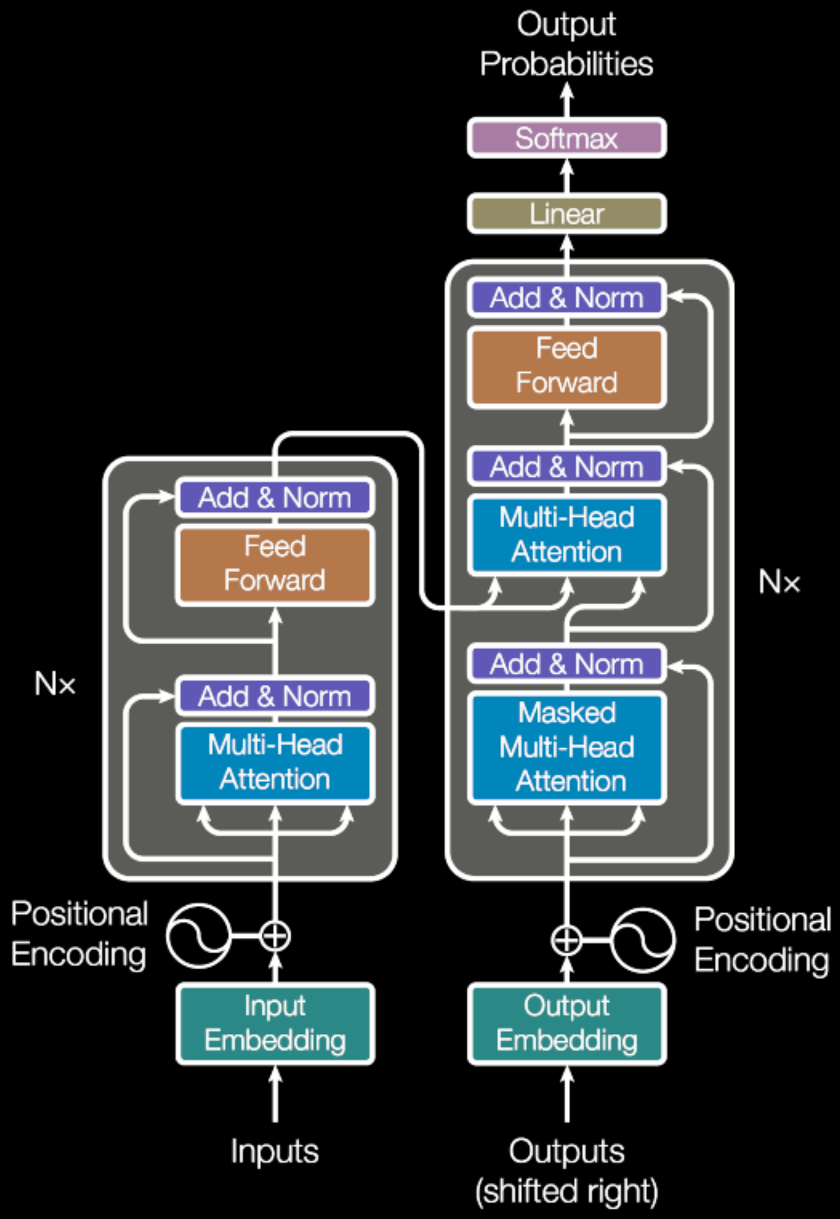


Figure 1: The Transformer - model architecture.

Encoder

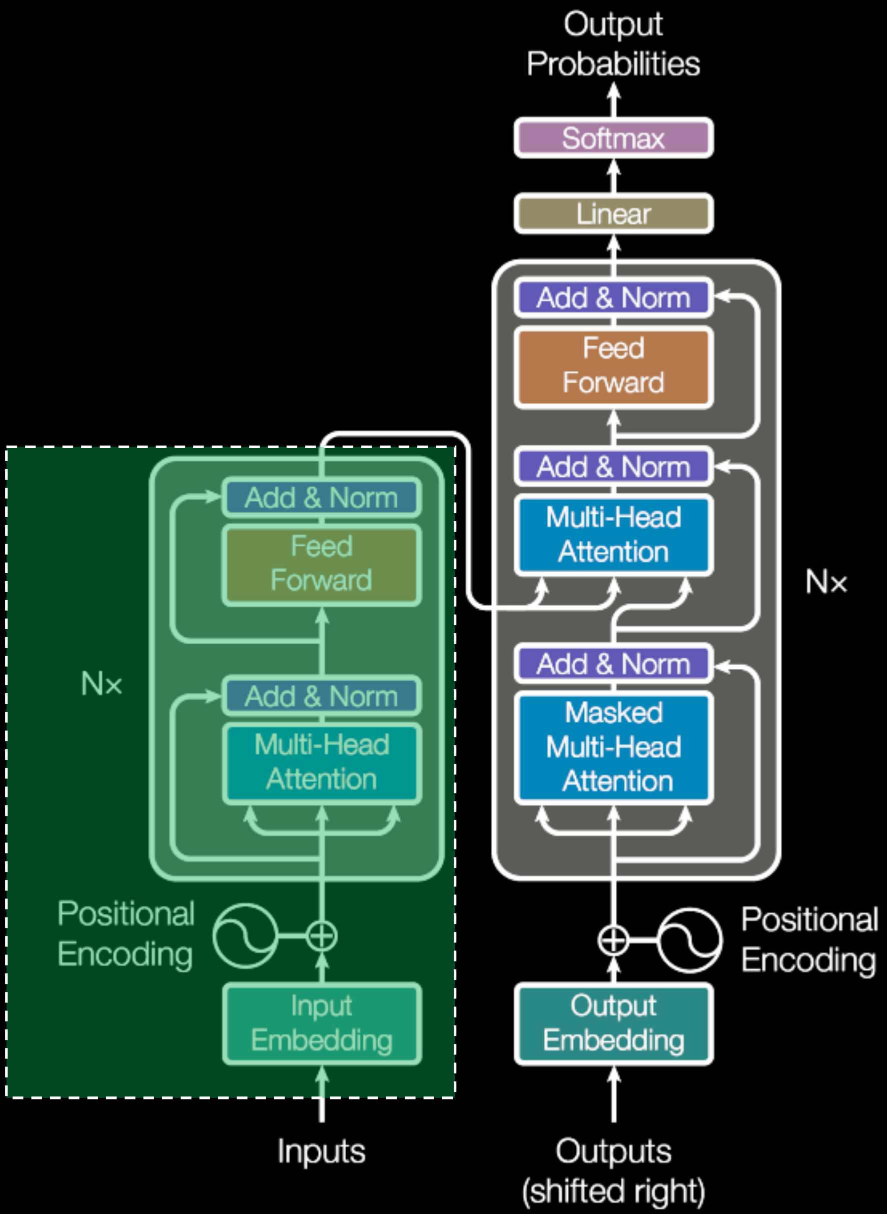


Figure 1: The Transformer - model architecture.

Decoder

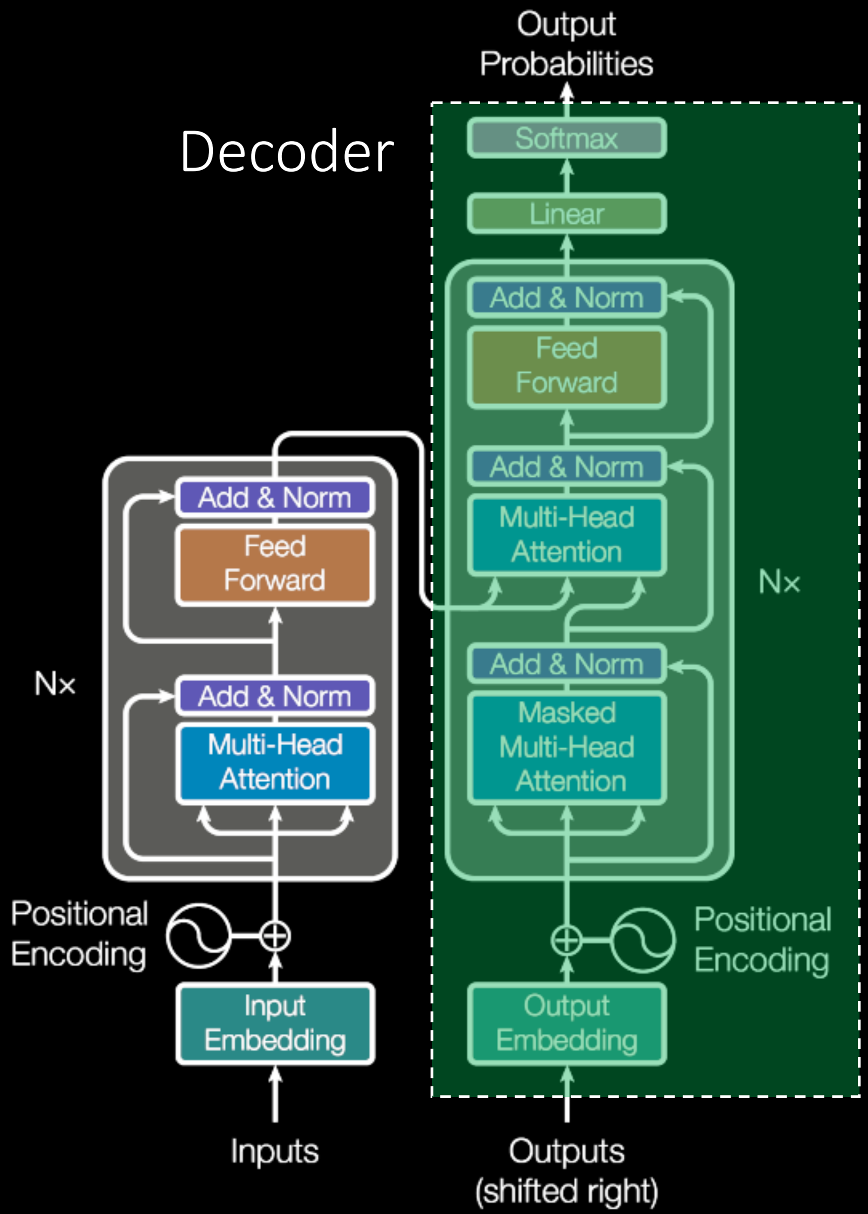


Figure 1: The Transformer - model architecture.

Transformer: Translation

Input: *“le chat est sous la table”*

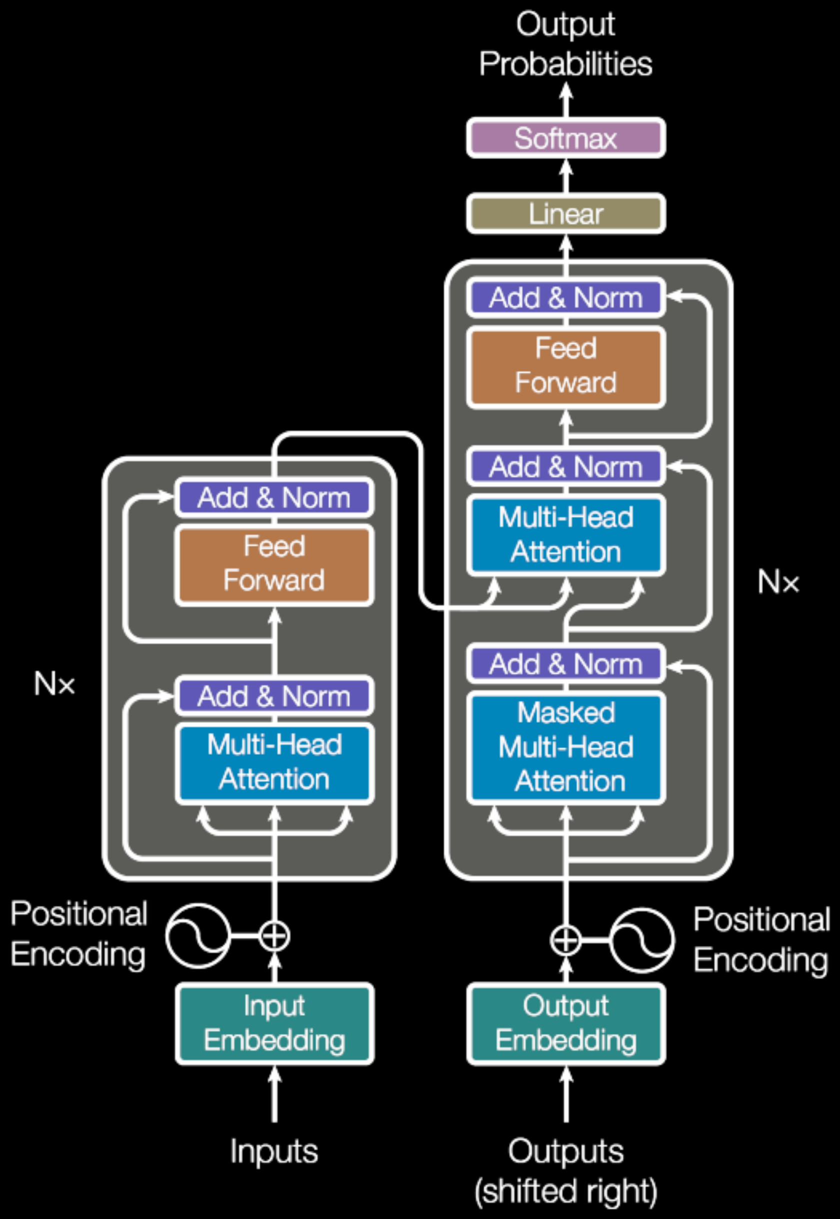


Figure 1: The Transformer - model architecture.

Input: *“le chat est sous la table”*

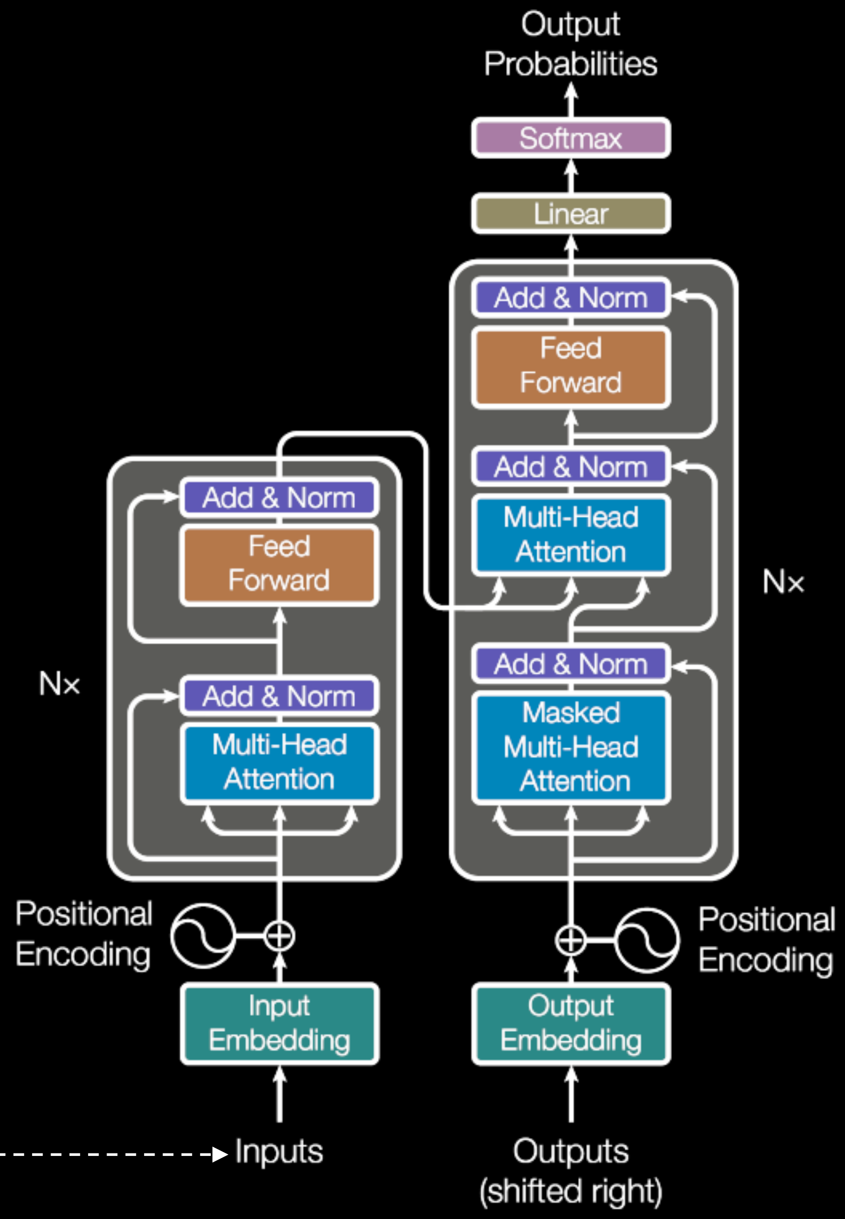


Figure 1: The Transformer - model architecture.

Input: *"le chat est sous la table"*

Output: <BOS>

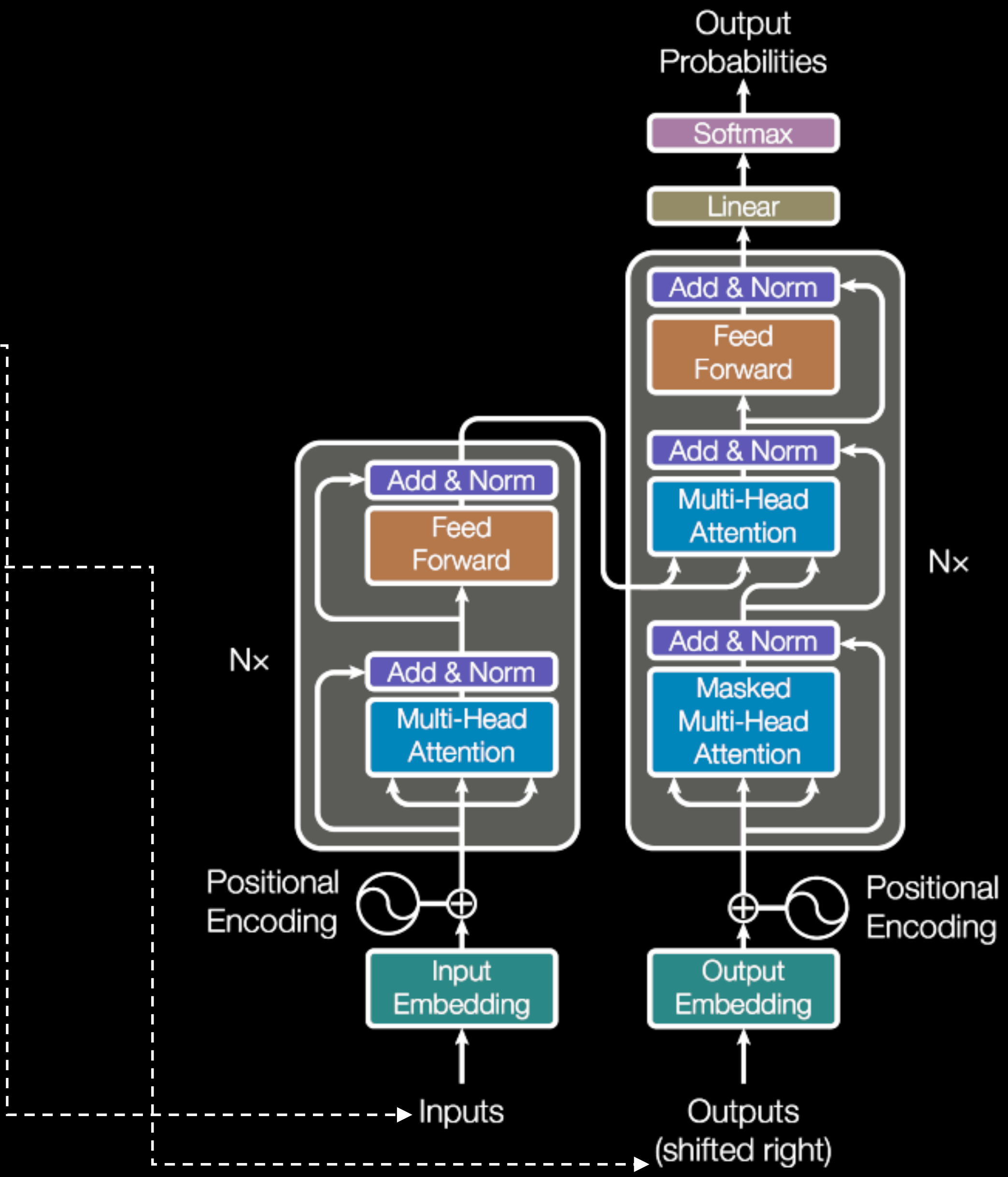


Figure 1: The Transformer - model architecture.

Input: *“le chat est sous la table”*

Output: <BOS>

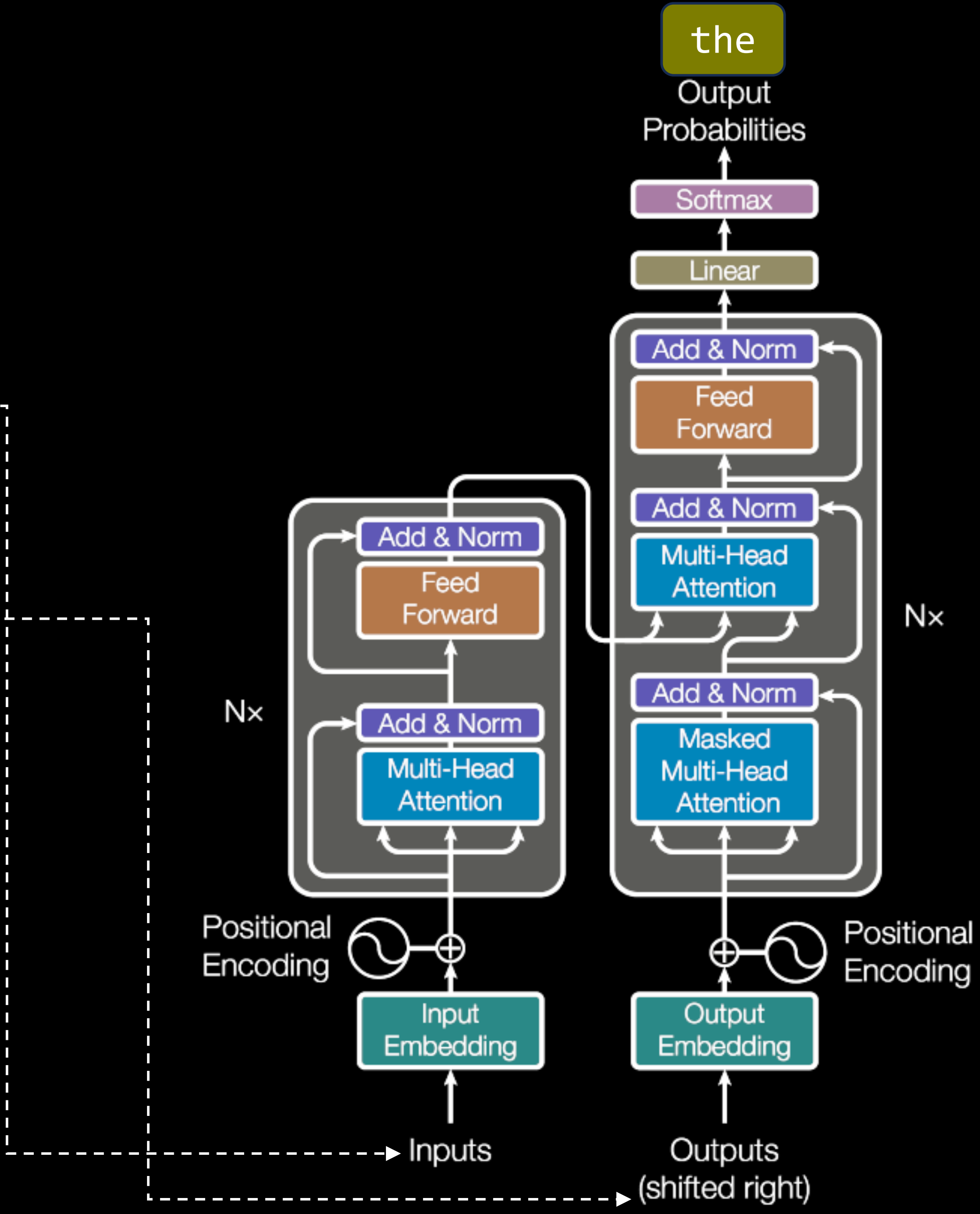


Figure 1: The Transformer - model architecture.

Input: *“le chat est sous la table”*

Output: `<BOS> the`

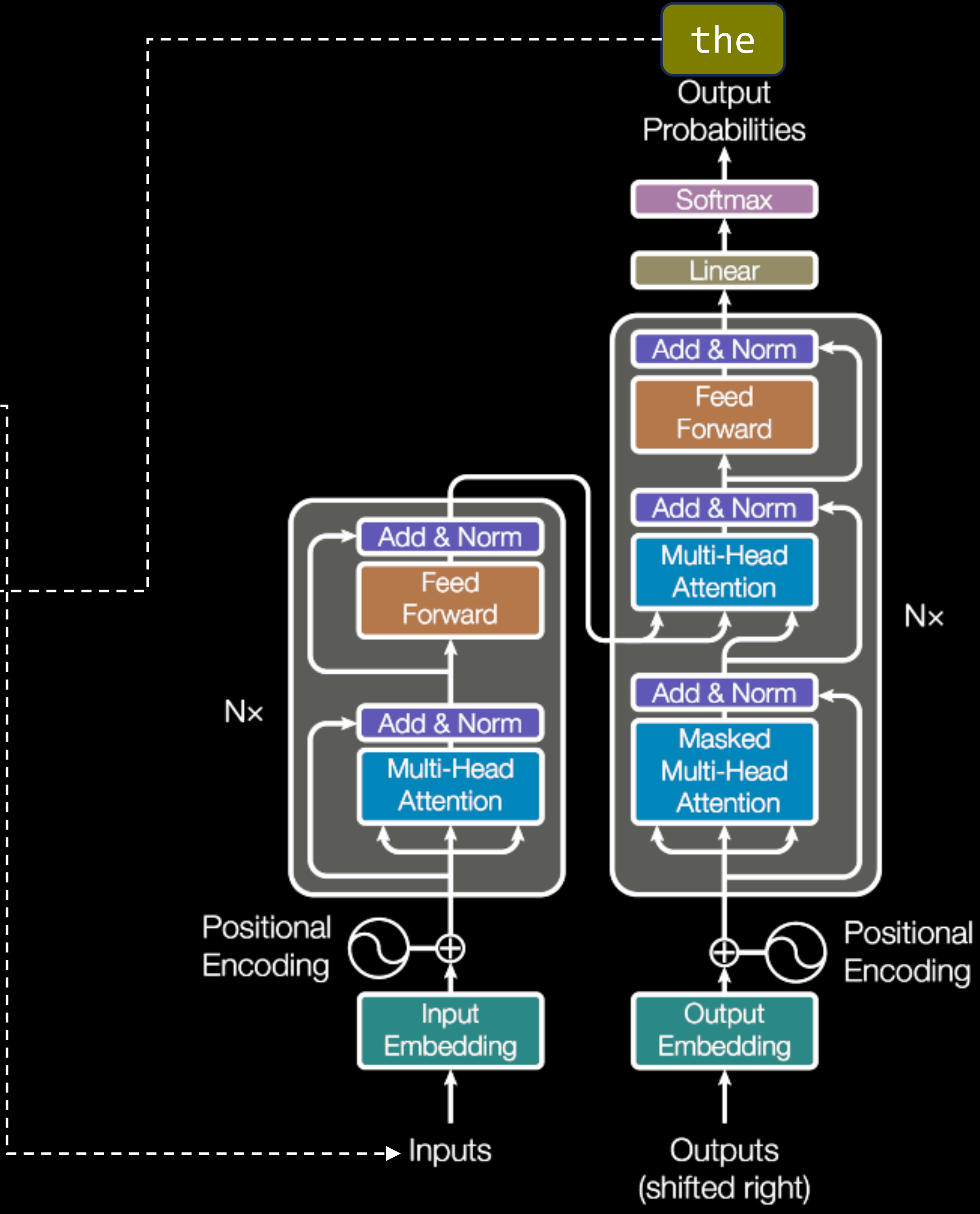


Figure 1: The Transformer - model architecture.

Input: *“le chat est sous la table”*

Output: `<BOS> the`

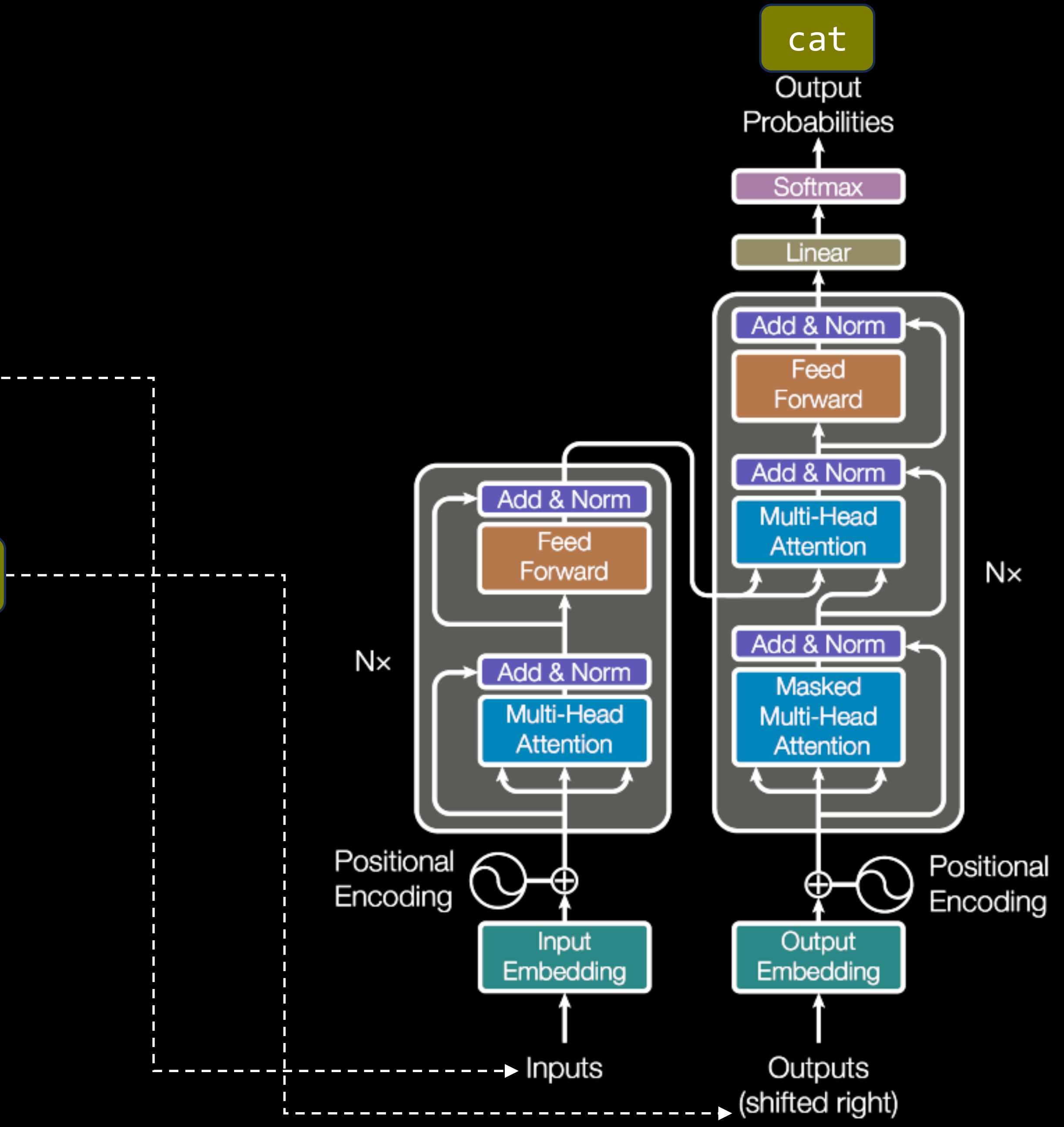


Figure 1: The Transformer - model architecture.

Input: *“le chat est sous la table”*

Output: `<BOS> the cat`

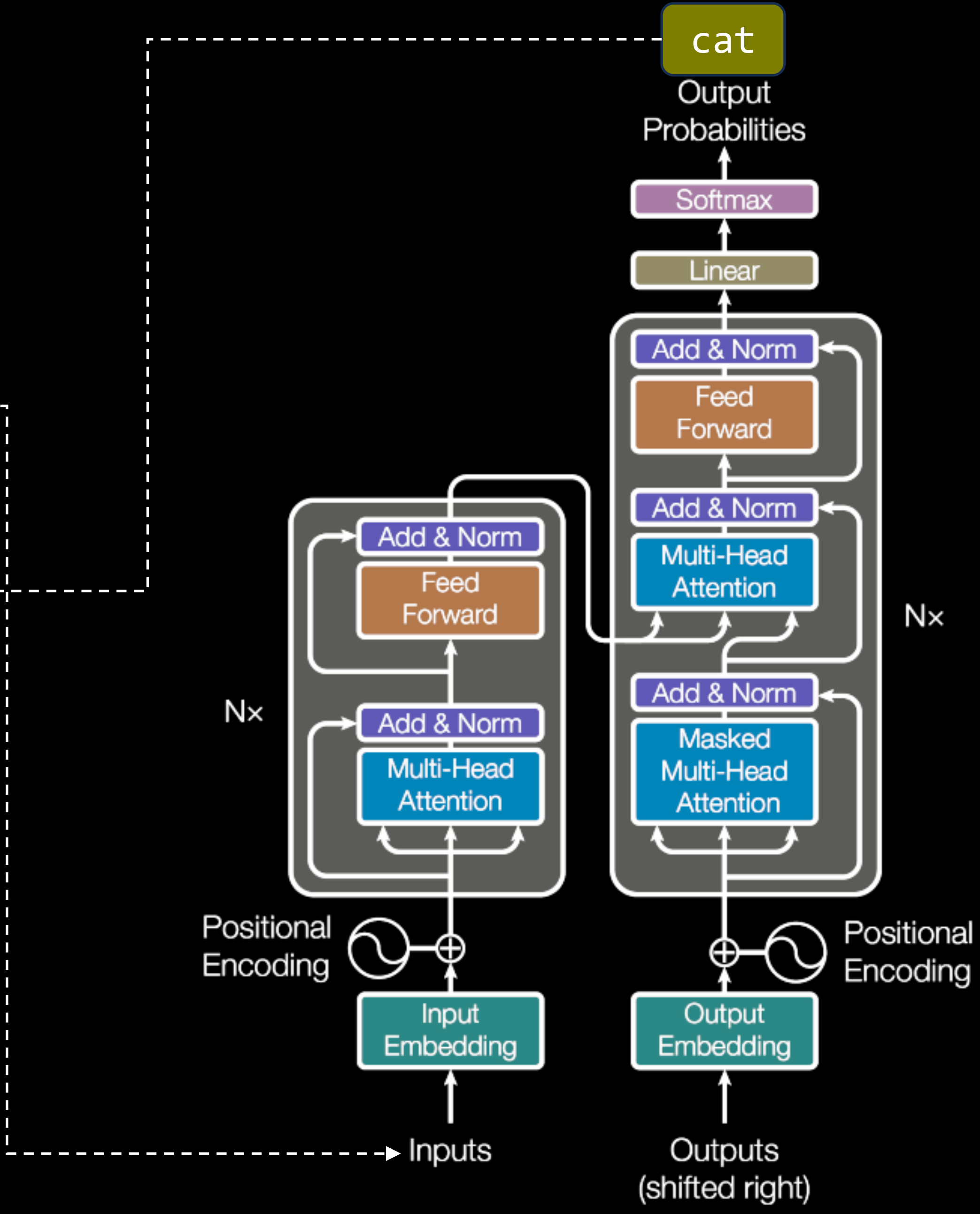


Figure 1: The Transformer - model architecture.

Input: *"le chat est sous la table"*

Output: `<BOS> the cat is under the table <EOS>`

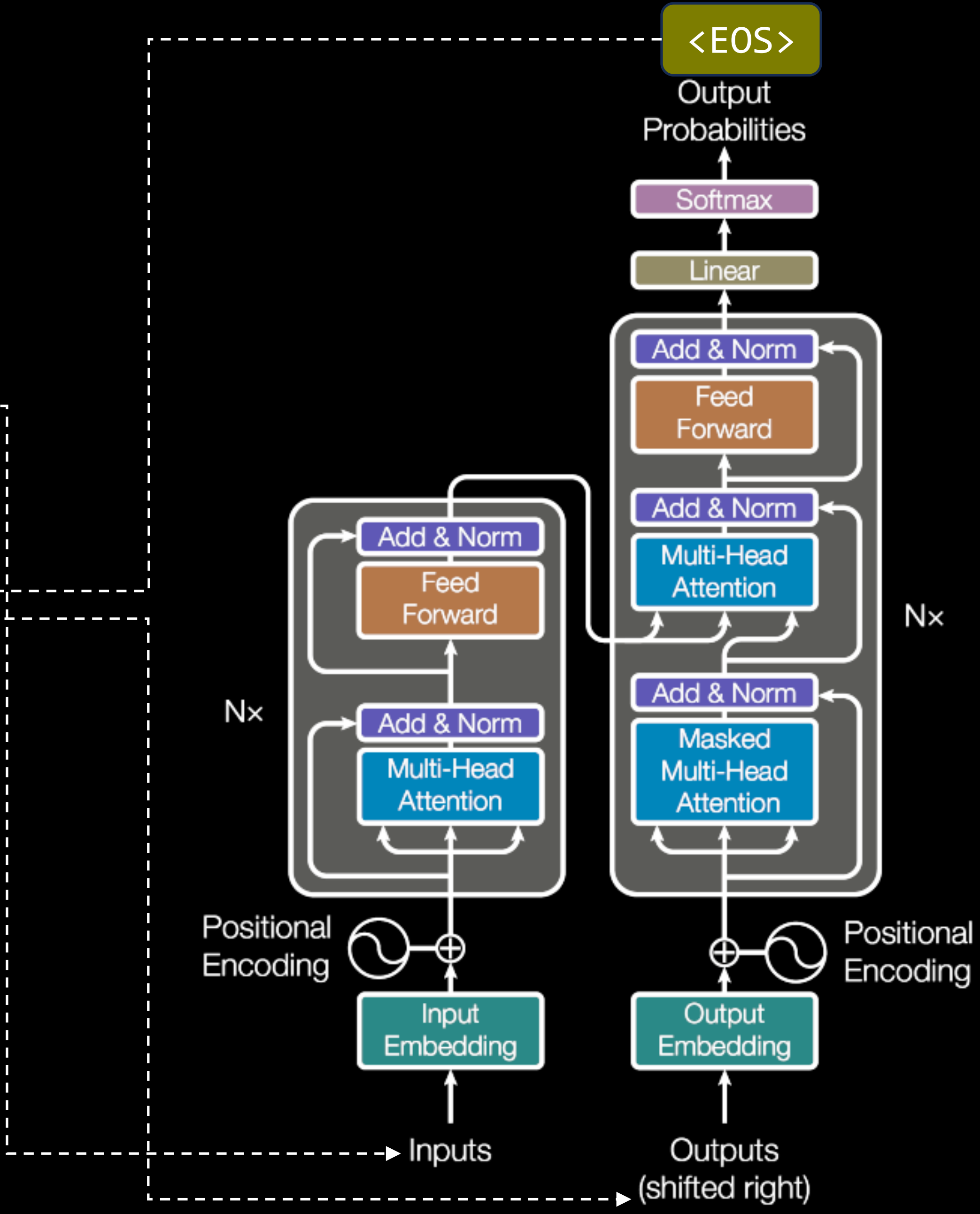
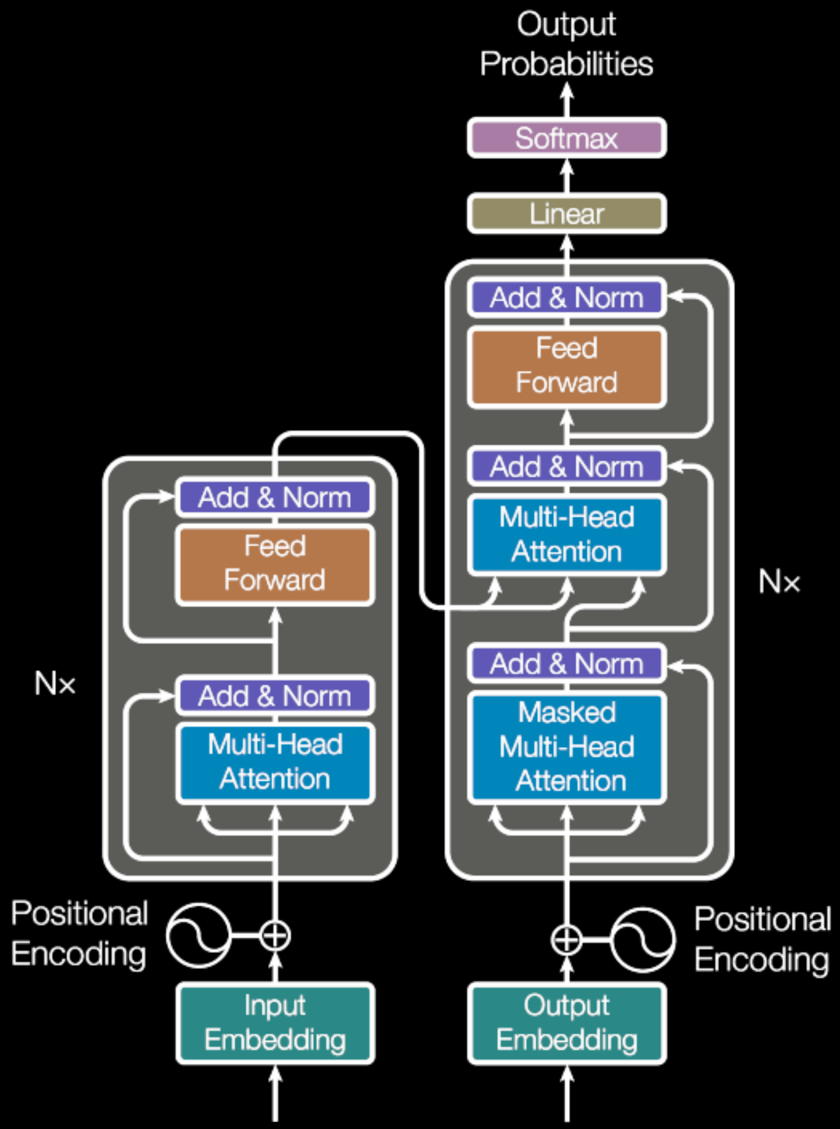
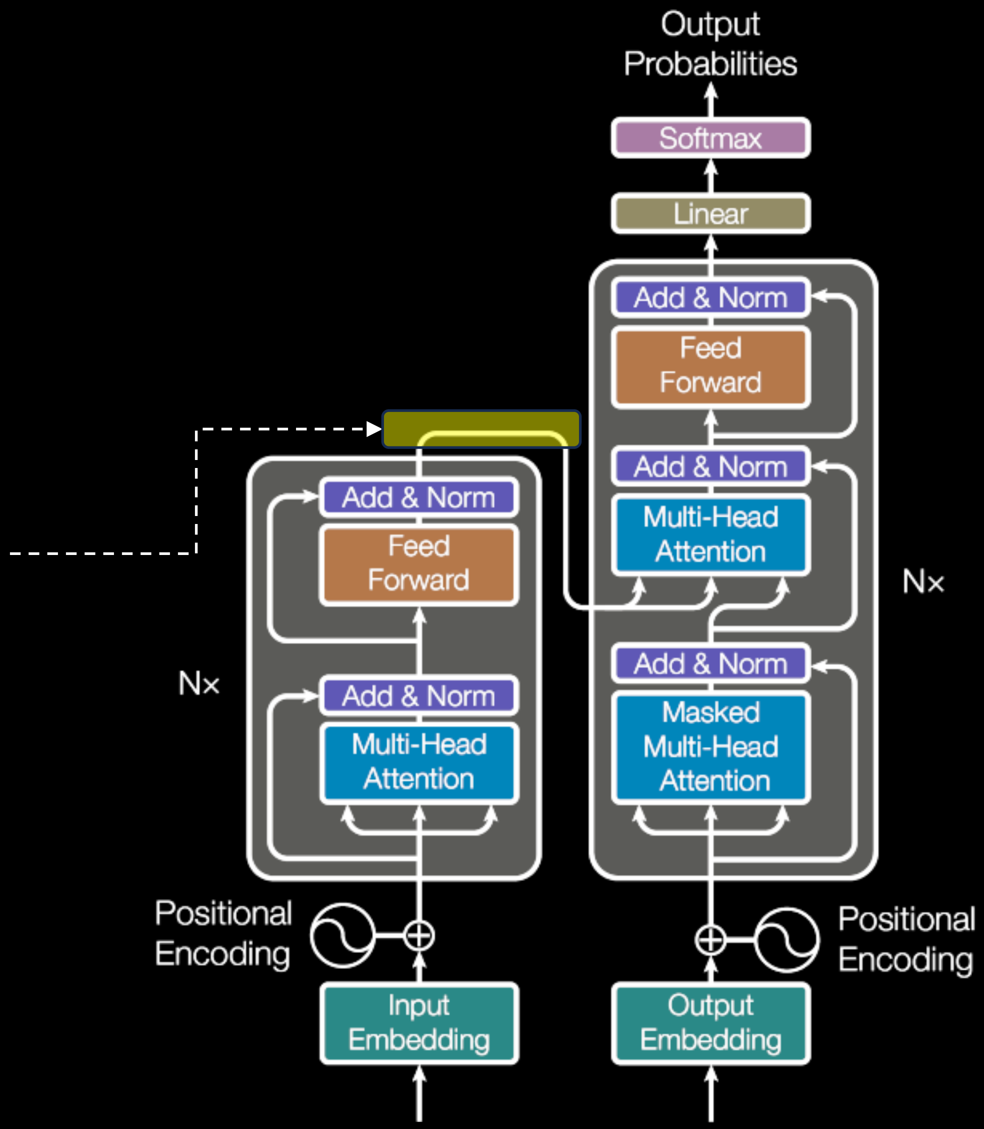


Figure 1: The Transformer - model architecture.

Contextual Embeddings



“le chat est sous la table”

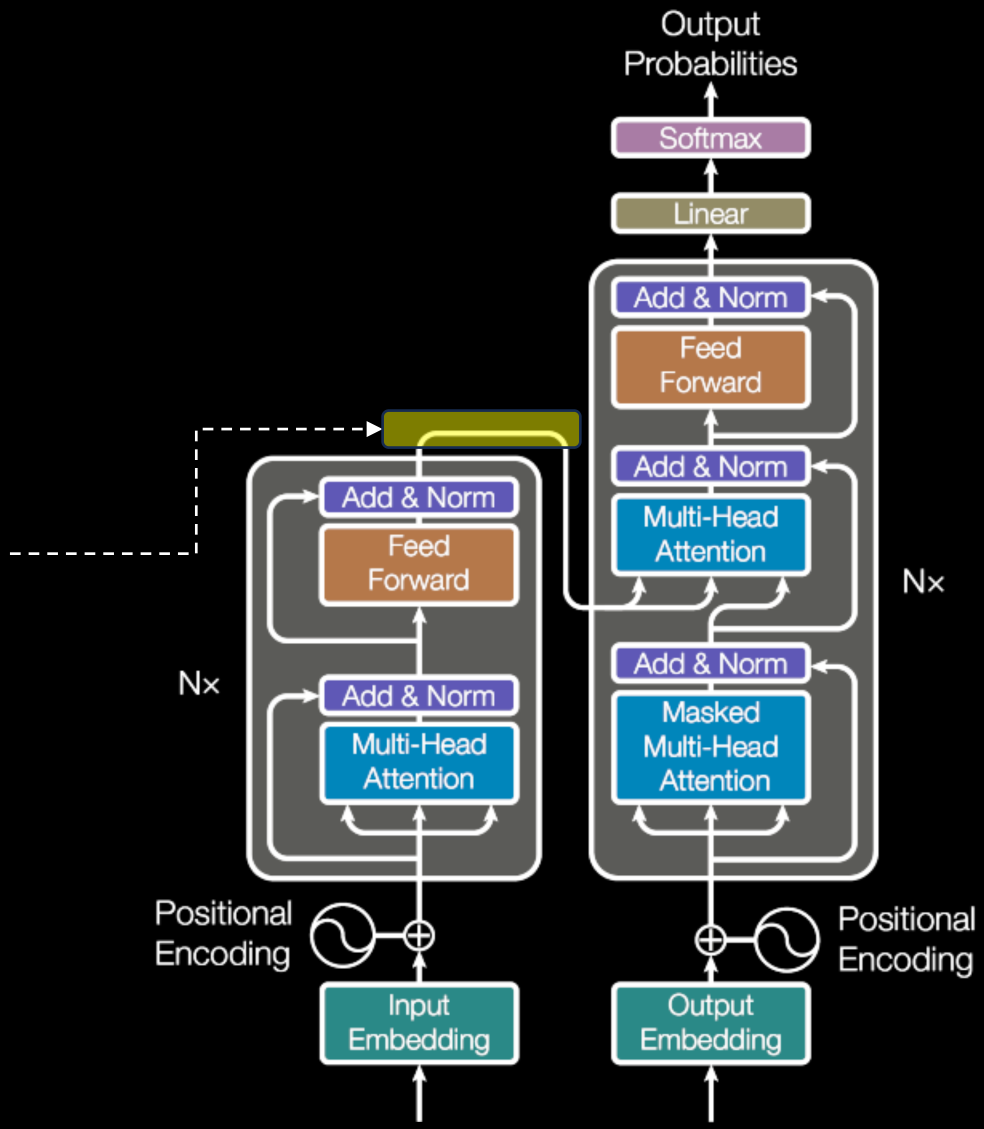


"le chat est sous la table"

This vector has ALL the information to translate the input “sentence”

So it must have ALL the information about the “meaning” of the input sentence.

This is a “Contextual Embedding”



“le chat est sous la table”

Topics

1.Language translation
    1.Simplest program: Literal translation
    2.Dealing with missing tokens

2.Literal translation using "ML”
    1.Encoding: One hot
    2.Using matrices
    3.A 'dictionary' using matrices
    4.Decoding: Cosine similarity

3.Attention
    1.Similar tokens: Softmax
    2.Matrix Query: Q
    3.Scaled dot-product attention

4.Attention Head
    1.Weight matrices
    2.Connecting matrices

5.Revisiting tokens & encoding
    1.Tokenizing: BPE
    2.Encoding: Embeddings

6.Transformer Block
    1.Multi-headed attention
    2.Non-linear (feed forward) layer
    3.Stack blocks
    4.Masked attention

7.Transformer
    1.Stacking deep networks
    2.Normalization layers
    3.Skip connections
    4.Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models

9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search

10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface

11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Pre-training / Training

- So, assume we have a model (like the one we’ve coded from scratch)
- We train it on a large dataset (e.g. all Wikipedia + All GitHub +…)
- Now we run the model with the prompt:

```
Input> Create a python function to generate the Fibonacci sequence
```

Pre-training / Training

- So, assume we have a model (like the one we’ve coded from scratch)
- We train it on a large dataset (e.g. all Wikipedia + All GitHub +…)
- Now we run the model with the prompt:

```
Input> Create a python function to generate the Fibonacci sequence
```

- …and we get as an output

```
Out> Create a python function to add two numbers
```

Pre-training / Training

- So, assume we have a model (like the one we've coded from scratch)
- We train it on a large dataset (e.g. all Wikipedia + All GitHub +…)
- Now we run the model with the prompt:

```
Input> Create a python function to generate the Fibonacci sequence
```

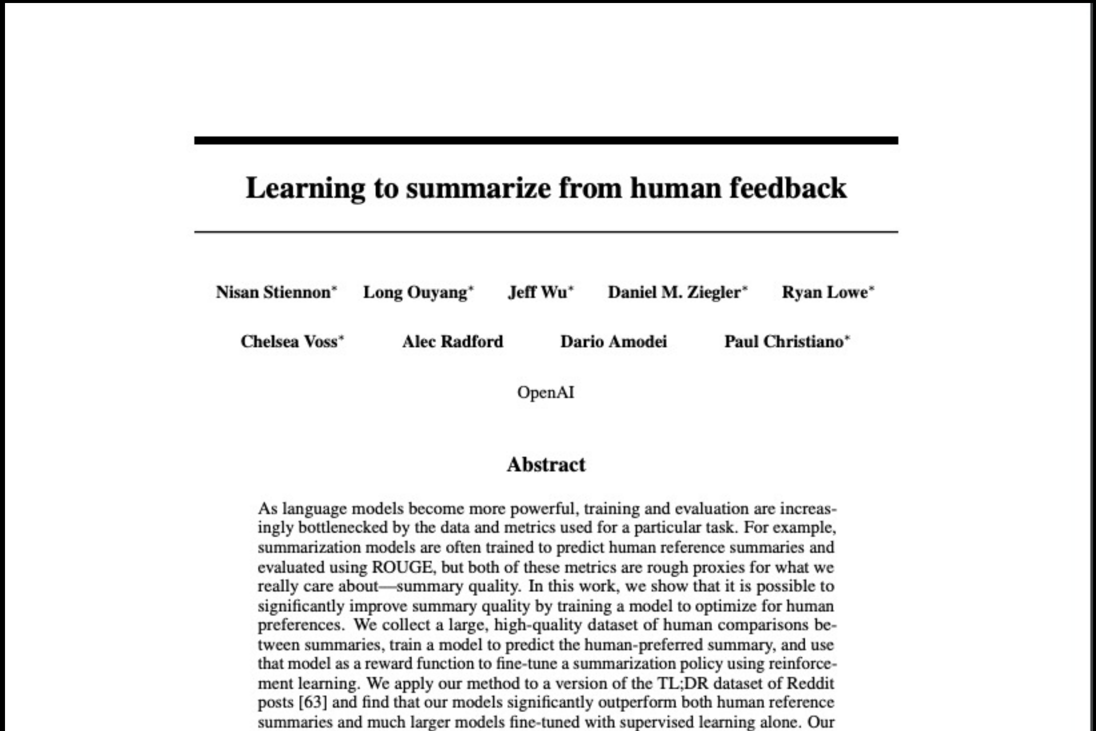
- …and we get as an output

```
Out> Create a python function to add two numbers
```

- The model has been trained to "predict" words, so it predicts the next sentence in a "Python test" questions, which might be another question
- …and not what we were expecting, which is the answer to our request

Pre-training / Training

- How do we solve this issue?
- We train the model to answer questions
- How: Reinforcement Learning (from Human Feedback)



Learning to summarize from human feedback

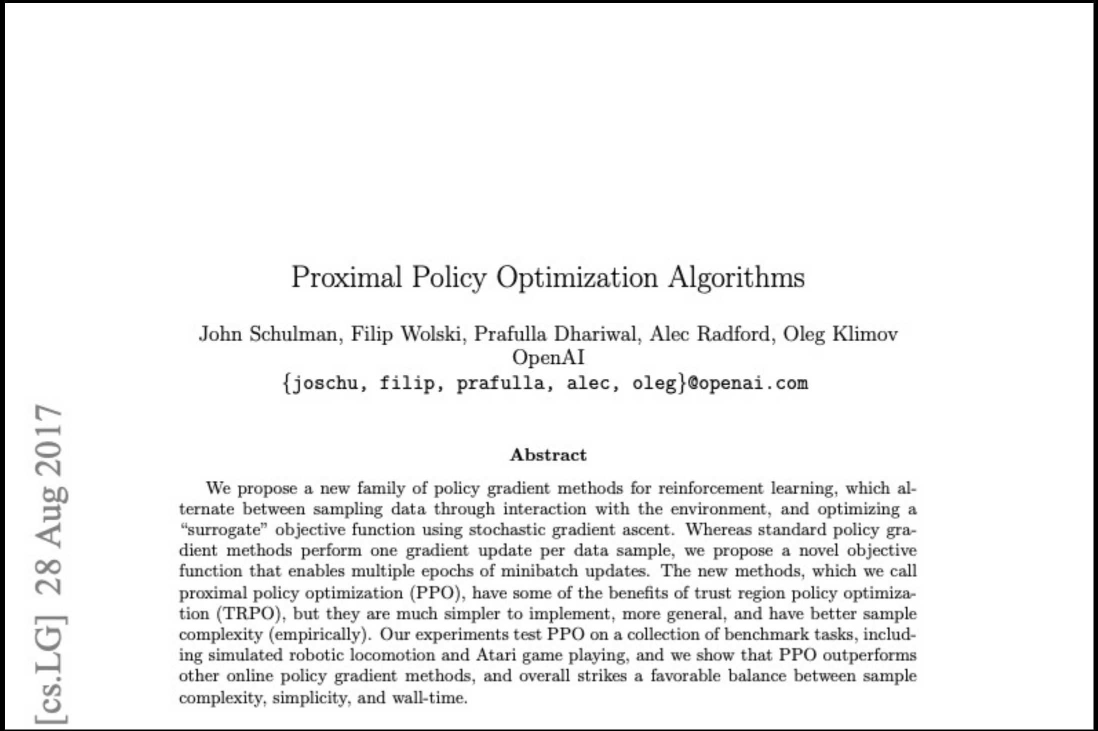
Nisan Stiennon* Long Ouyang* Jeff Wu* Daniel M. Ziegler* Ryan Lowe*

Chelsea Voss* Alec Radford Dario Amodei Paul Christiano*

OpenAI

Abstract

As language models become more powerful, training and evaluation are increasingly bottlenecked by the data and metrics used for a particular task. For example, summarization models are often trained to predict human reference summaries and evaluated using ROUGE, but both of these metrics are rough proxies for what we really care about—summary quality. In this work, we show that it is possible to significantly improve summary quality by training a model to optimize for human preferences. We collect a large, high-quality dataset of human comparisons between summaries, train a model to predict the human-preferred summary, and use that model as a reward function to fine-tune a summarization policy using reinforcement learning. We apply our method to a version of the TL;DR dataset of Reddit posts [63] and find that our models significantly outperform both human reference summaries and much larger models fine-tuned with supervised learning alone. Our



[cs.LG] 28 Aug 2017

Proximal Policy Optimization Algorithms

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov
OpenAI
{joschu, filip, prafulla, alec, oleg}@openai.com

Abstract

We propose a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. Whereas standard policy gradient methods perform one gradient update per data sample, we propose a novel objective function that enables multiple epochs of minibatch updates. The new methods, which we call proximal policy optimization (PPO), have some of the benefits of trust region policy optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). Our experiments test PPO on a collection of benchmark tasks, including simulated robotic locomotion and Atari game playing, and we show that PPO outperforms other online policy gradient methods, and overall strikes a favorable balance between sample complexity, simplicity, and wall-time.

Pre-training / Training

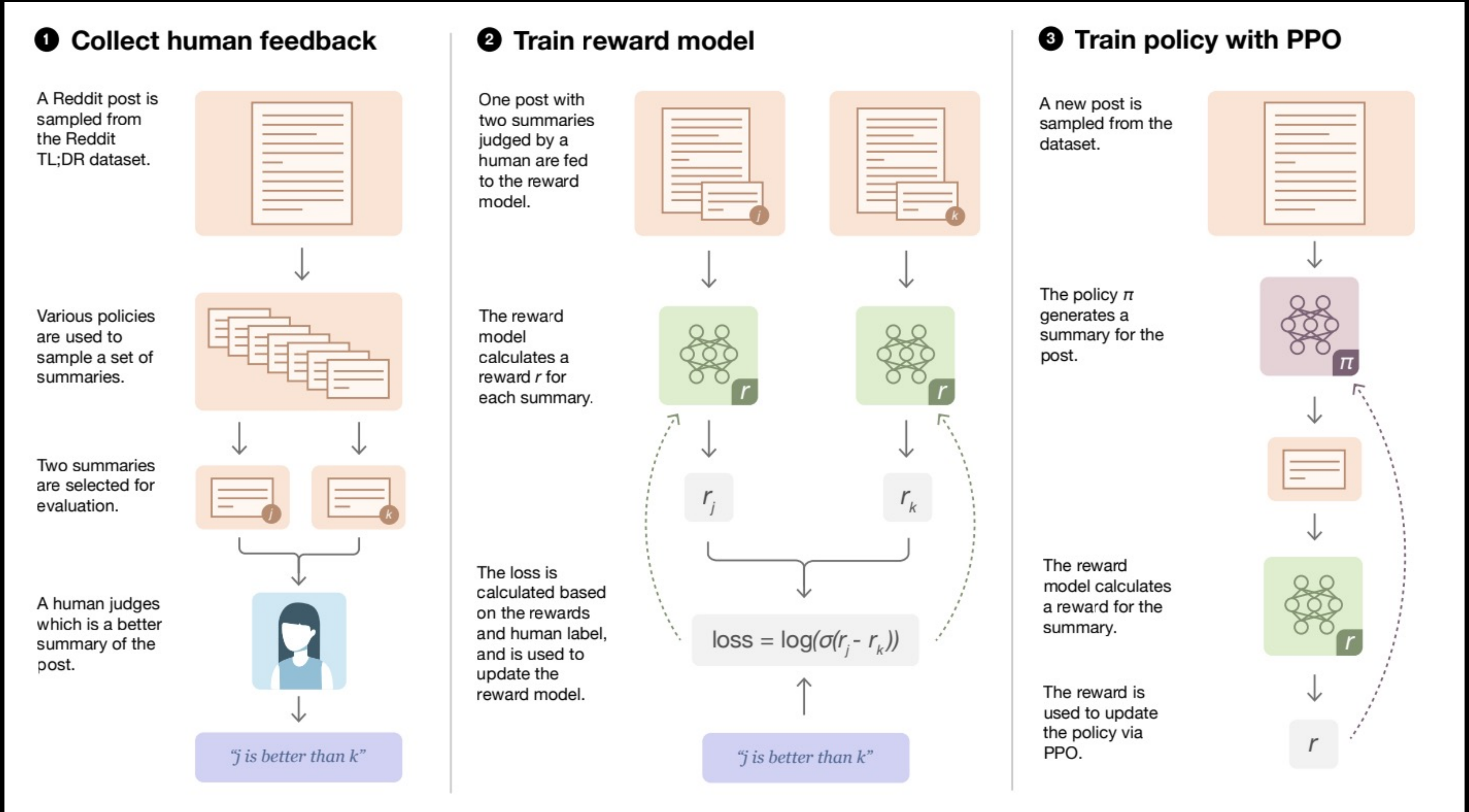
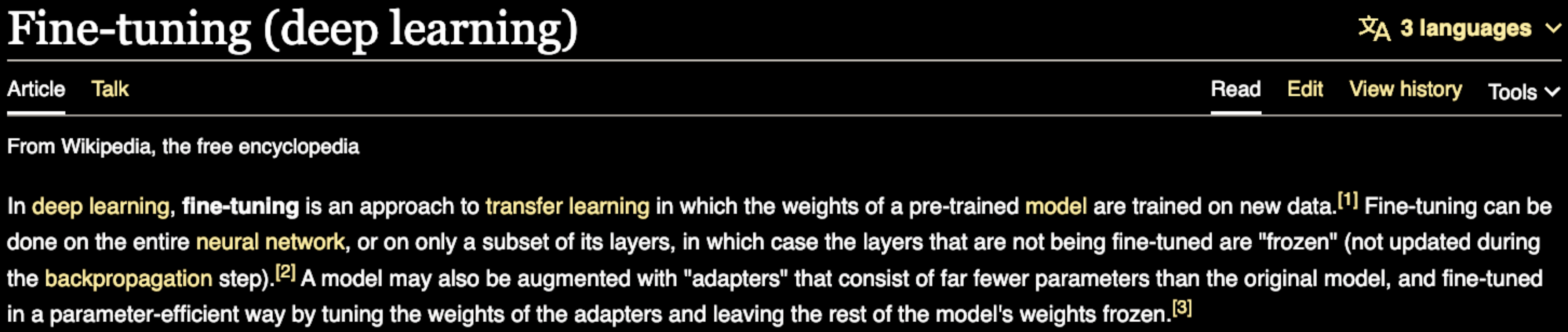

1 Collect human feedback
A Reddit post is sampled from the Reddit TL;DR dataset.
Various policies are used to sample a set of summaries.
Two summaries are selected for evaluation.
j
k
A human judges which is a better summary of the post.
"j is better than k"
2 Train reward model
One post with two summaries judged by a human are fed to the reward model.
The reward model calculates a reward r for each summary.
r
r_j
r_k
The loss is calculated based on the rewards and human label, and is used to update the reward model.
loss = $\log(\sigma(r_j - r_k))$
"j is better than k"
3 Train policy with PPO
A new post is sampled from the dataset.
The policy π generates a summary for the post.
π
The reward model calculates a reward for the summary.
The reward is used to update the policy via PPO.
r

Figure 2: Diagram of our human feedback, reward model training, and policy training procedure.

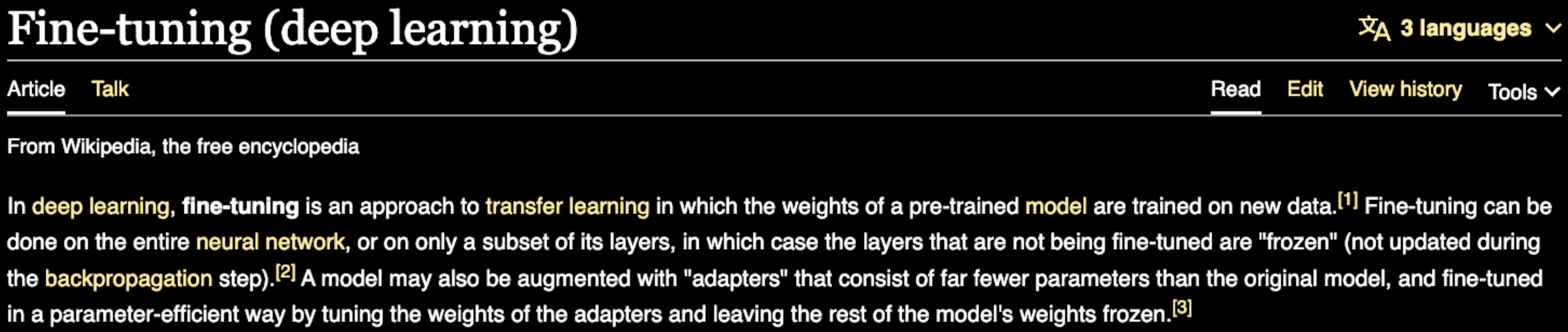
Fine-tunning


Fine-tuning (deep learning)

From Wikipedia, the free encyclopedia

In deep learning, **fine-tuning** is an approach to transfer learning in which the weights of a pre-trained model are trained on new data.[1] Fine-tuning can be done on the entire neural network, or on only a subset of its layers, in which case the layers that are not being fine-tuned are "frozen" (not updated during the backpropagation step).[2] A model may also be augmented with "adapters" that consist of far fewer parameters than the original model, and fine-tuned in a parameter-efficient way by tuning the weights of the adapters and leaving the rest of the model's weights frozen.[3]

Fine-tunning



Fine-tuning (deep learning)

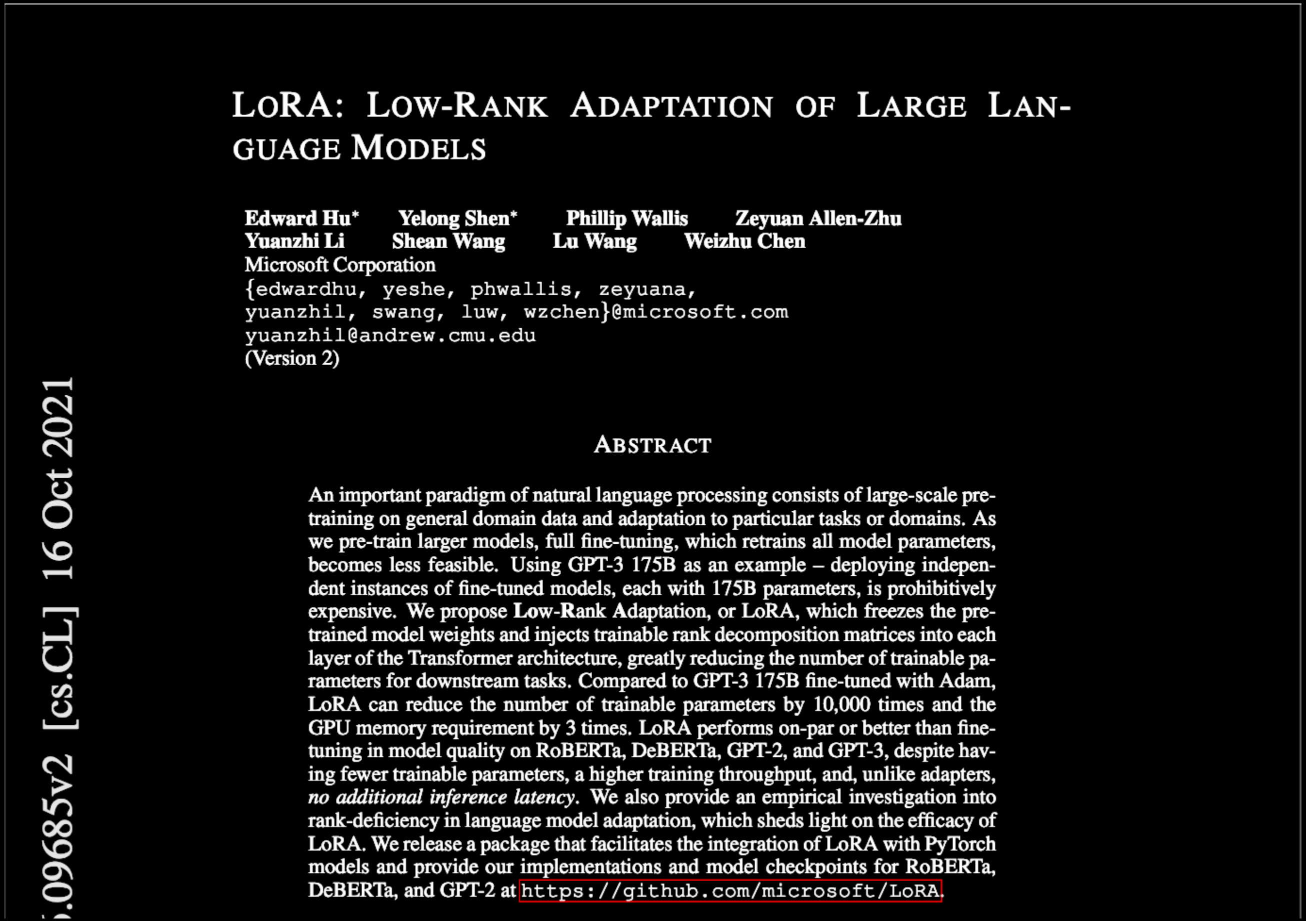
From Wikipedia, the free encyclopedia

In deep learning, **fine-tuning** is an approach to transfer learning in which the weights of a pre-trained model are trained on new data.[1] Fine-tuning can be done on the entire neural network, or on only a subset of its layers, in which case the layers that are not being fine-tuned are "frozen" (not updated during the backpropagation step).[2] A model may also be augmented with "adapters" that consist of far fewer parameters than the original model, and fine-tuned in a parameter-efficient way by tuning the weights of the adapters and leaving the rest of the model's weights frozen.[3]

Problem: We need to “fine tune” a full model

The model is large => Fine tuning requires a lot of resources

Adaptation



LoRA: Low-Rank Adaptation of Large Language Models

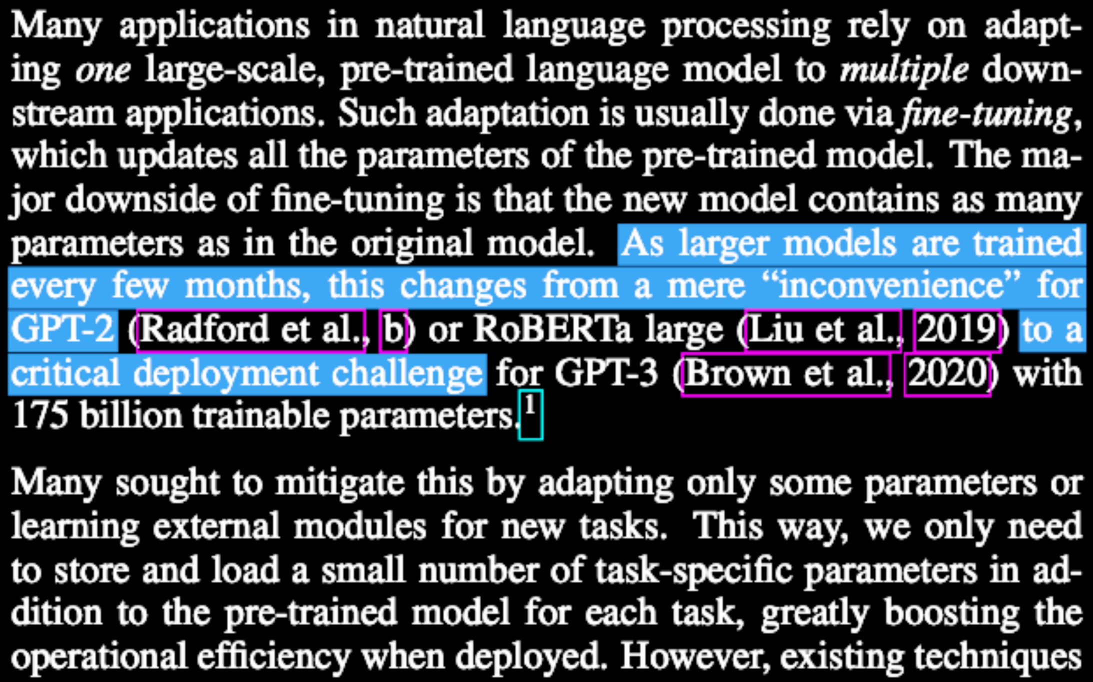
Edward Hu* **Yelong Shen*** **Phillip Wallis** **Zeyuan Allen-Zhu**
Yuanzhi Li **Shean Wang** **Lu Wang** **Weizhu Chen**
Microsoft Corporation
{edwardhu, yeshe, phwallis, zeyuana,
yuanzhil, swang, luw, wzchen}@microsoft.com
yuanzhil@andrew.cmu.edu
(Version 2)

.09685v2 [cs.CL] 16 Oct 2021

Abstract

An important paradigm of natural language processing consists of large-scale pre-training on general domain data and adaptation to particular tasks or domains. As we pre-train larger models, full fine-tuning, which retrains all model parameters, becomes less feasible. Using GPT-3 175B as an example – deploying independent instances of fine-tuned models, each with 175B parameters, is prohibitively expensive. We propose **Lo**w-**R**ank **A**daptation, or LoRA, which freezes the pre-trained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture, greatly reducing the number of trainable parameters for downstream tasks. Compared to GPT-3 175B fine-tuned with Adam, LoRA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times. LoRA performs on-par or better than fine-tuning in model quality on RoBERTa, DeBERTa, GPT-2, and GPT-3, despite having fewer trainable parameters, a higher training throughput, and, unlike adapters, *no additional inference latency*. We also provide an empirical investigation into rank-deficiency in language model adaptation, which sheds light on the efficacy of LoRA. We release a package that facilitates the integration of LoRA with PyTorch models and provide our implementations and model checkpoints for RoBERTa, DeBERTa, and GPT-2 at https://github.com/microsoft/LoRA.

Adaptation



Many applications in natural language processing rely on adapting *one* large-scale, pre-trained language model to *multiple* downstream applications. Such adaptation is usually done via *fine-tuning*, which updates all the parameters of the pre-trained model. The major downside of fine-tuning is that the new model contains as many parameters as in the original model. As larger models are trained every few months, this changes from a mere "inconvenience" for GPT-2 (Radford et al., b) or RoBERTa large (Liu et al., 2019) to a critical deployment challenge for GPT-3 (Brown et al., 2020) with 175 billion trainable parameters.[1]

Many sought to mitigate this by adapting only some parameters or learning external modules for new tasks. This way, we only need to store and load a small number of task-specific parameters in addition to the pre-trained model for each task, greatly boosting the operational efficiency when deployed. However, existing techniques

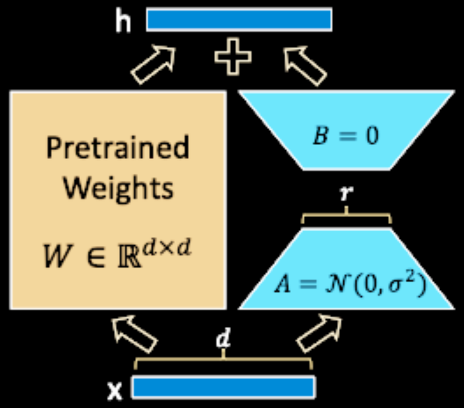
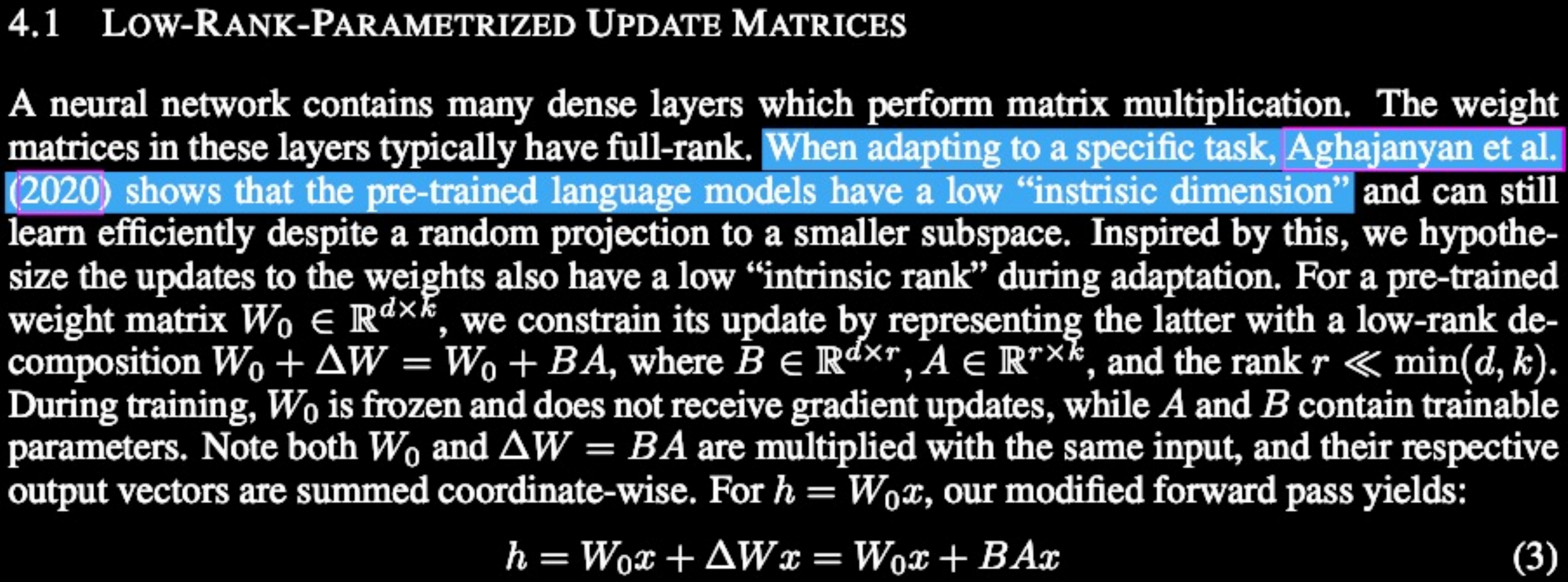

h
$B = 0$
r
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$A = \mathcal{N}(0, \sigma^2)$
d
x

Figure 1: Our reparametrization. We only train A and B.

Adaptation



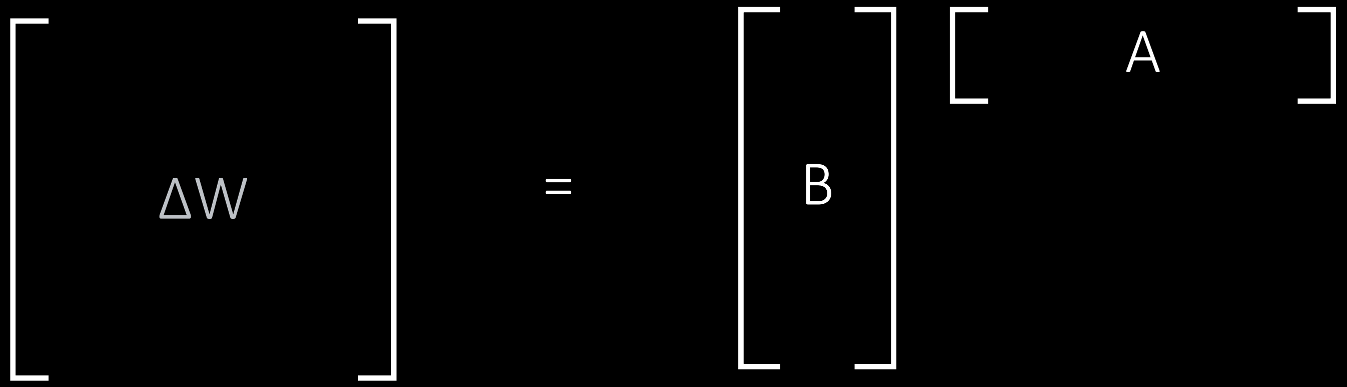
4.1 Low-Rank-Parametrized Update Matrices

A neural network contains many dense layers which perform matrix multiplication. The weight matrices in these layers typically have full-rank. When adapting to a specific task, Aghajanyan et al. (2020) shows that the pre-trained language models have a low "instrisic dimension" and can still learn efficiently despite a random projection to a smaller subspace. Inspired by this, we hypothesize the updates to the weights also have a low "intrinsic rank" during adaptation. For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, we constrain its update by representing the latter with a low-rank decomposition $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times k}$, and the rank $r \ll \min(d, k)$. During training, W_0 is frozen and does not receive gradient updates, while A and B contain trainable parameters. Note both W_0 and $\Delta W = BA$ are multiplied with the same input, and their respective output vectors are summed coordinate-wise. For $h = W_0 x$, our modified forward pass yields:

$$h = W_0 x + \Delta W x = W_0 x + BAx \tag{3}$$

Adaptation

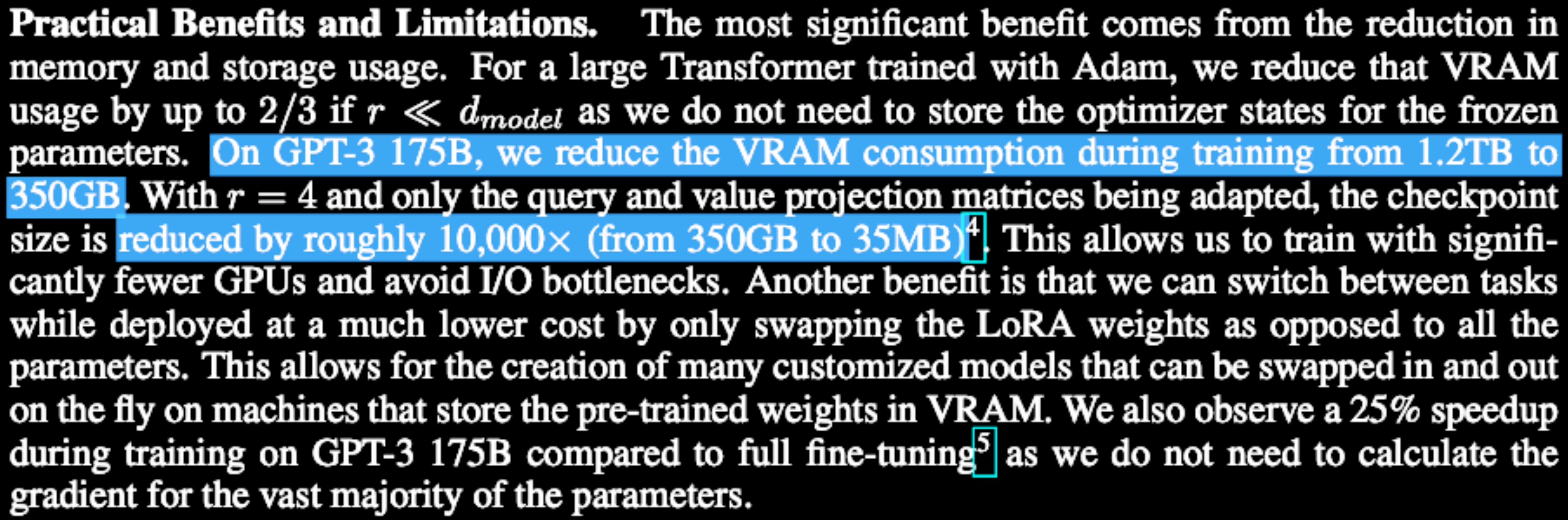
$$h = W_0 x + \underbrace{\Delta W}_{} x = W_0 x + \underbrace{BA}_{} x$$



$$B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}$$

In the paper they show that B and A can be very low rank (e.g. 8, 4)
They even show adaptation with r=1 (i.e. B and A are vectors)

Adaptation


Practical Benefits and Limitations. The most significant benefit comes from the reduction in memory and storage usage. For a large Transformer trained with Adam, we reduce that VRAM usage by up to $2/3$ if $r \ll d_{model}$ as we do not need to store the optimizer states for the frozen parameters. On GPT-3 175B, we reduce the VRAM consumption during training from 1.2TB to 350GB. With $r = 4$ and only the query and value projection matrices being adapted, the checkpoint size is reduced by roughly 10,000× (from 350GB to 35MB)[4]. This allows us to train with significantly fewer GPUs and avoid I/O bottlenecks. Another benefit is that we can switch between tasks while deployed at a much lower cost by only swapping the LoRA weights as opposed to all the parameters. This allows for the creation of many customized models that can be swapped in and out on the fly on machines that store the pre-trained weights in VRAM. We also observe a 25% speedup during training on GPT-3 175B compared to full fine-tuning[5] as we do not need to calculate the gradient for the vast majority of the parameters.

Recap: Pre-training / Training / Adapting

- Pre-Training: *GPT model*
  Model is trained to learn the language
  (e.g. predict the next word in a sentence).

- Training: *GPT-Instruct model*
  Model is trained to learn to answer questions / follow directions
  (PPO, Reinforcement Learning from Human Feedback).

- Fine-tuning:
  The model is trained to learn for domain specific knowledge
  (e.g. Training on Cancer Biology papers).

- Adaptation / LoRA: Same as fine-tuning, but using an optimized method

GPT: Generative Pre-trained Transformer

GPT

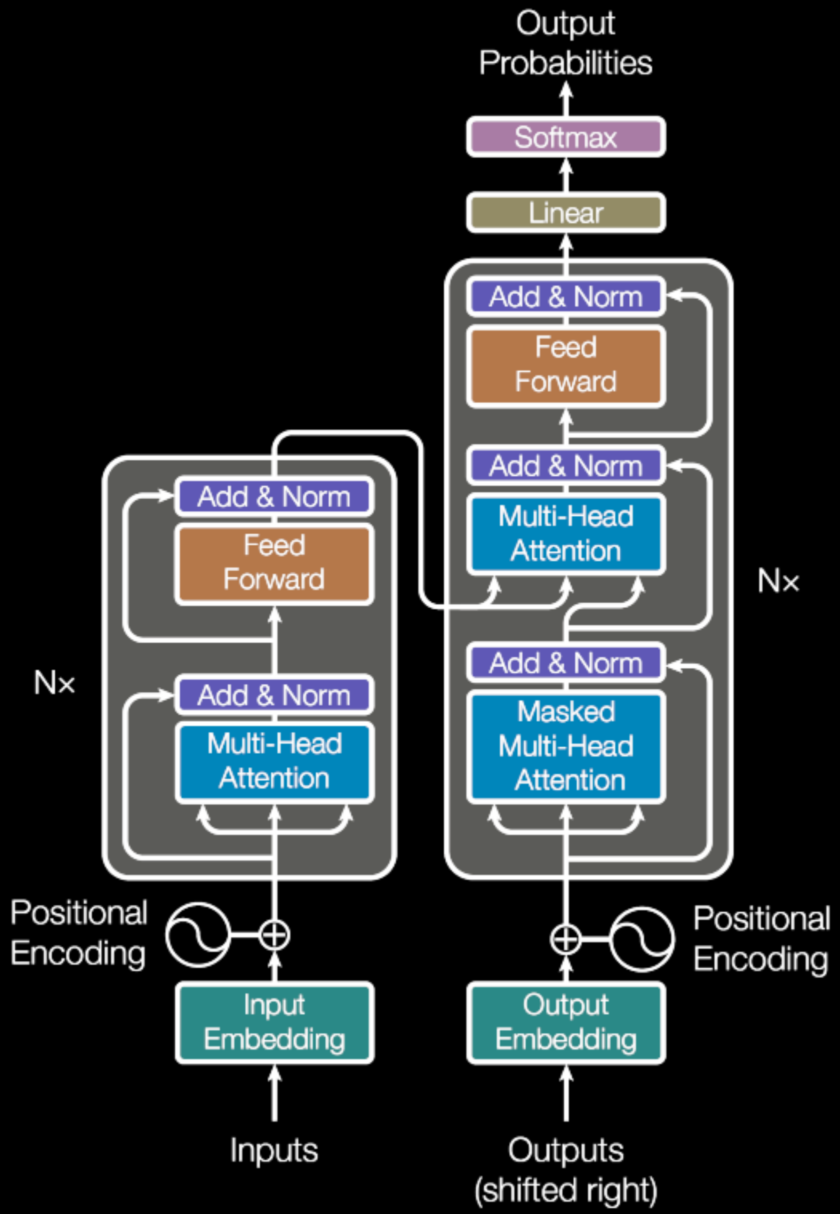


Figure 1: The Trans

GPT

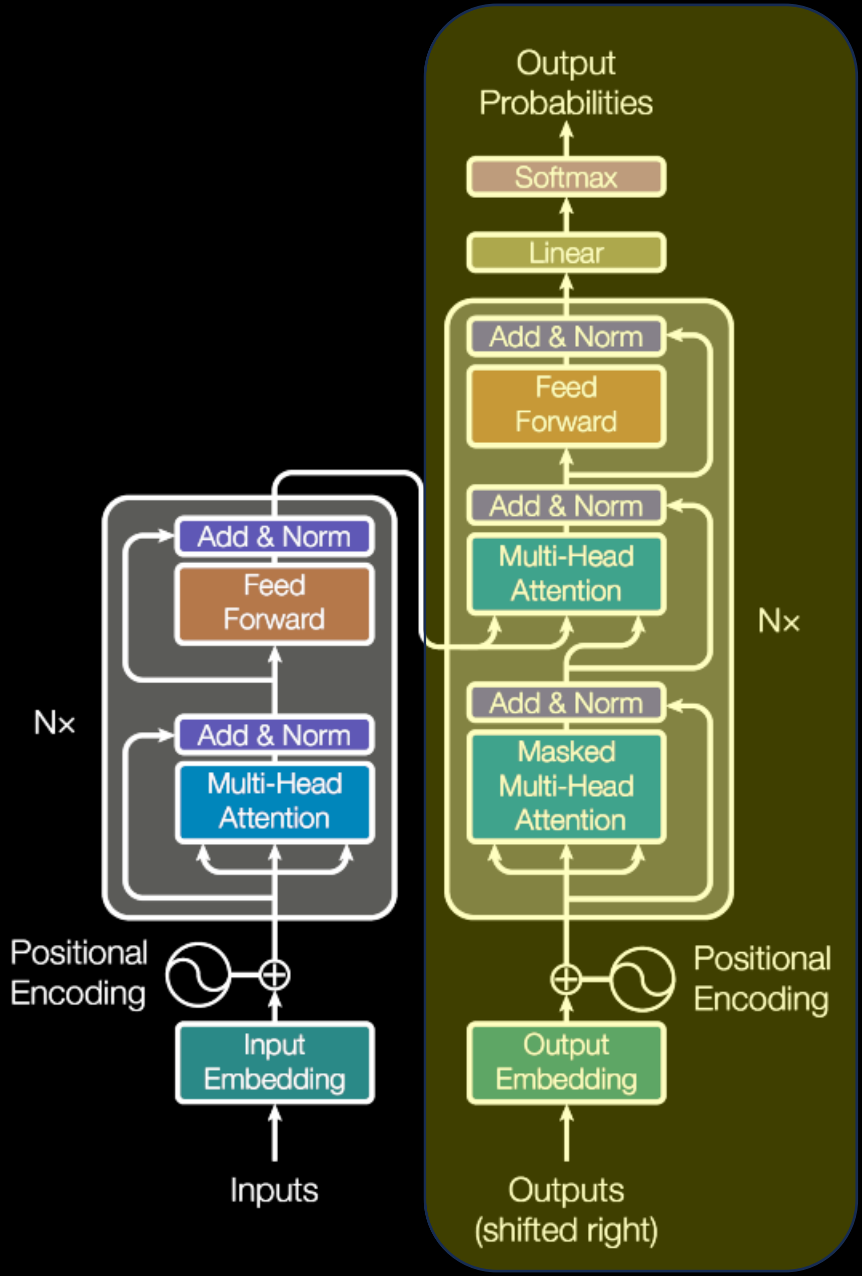
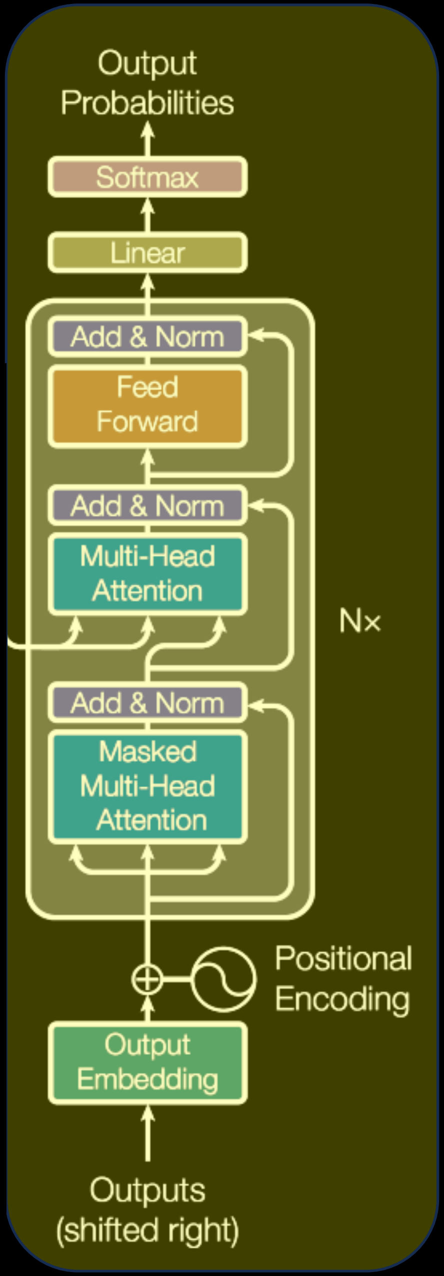


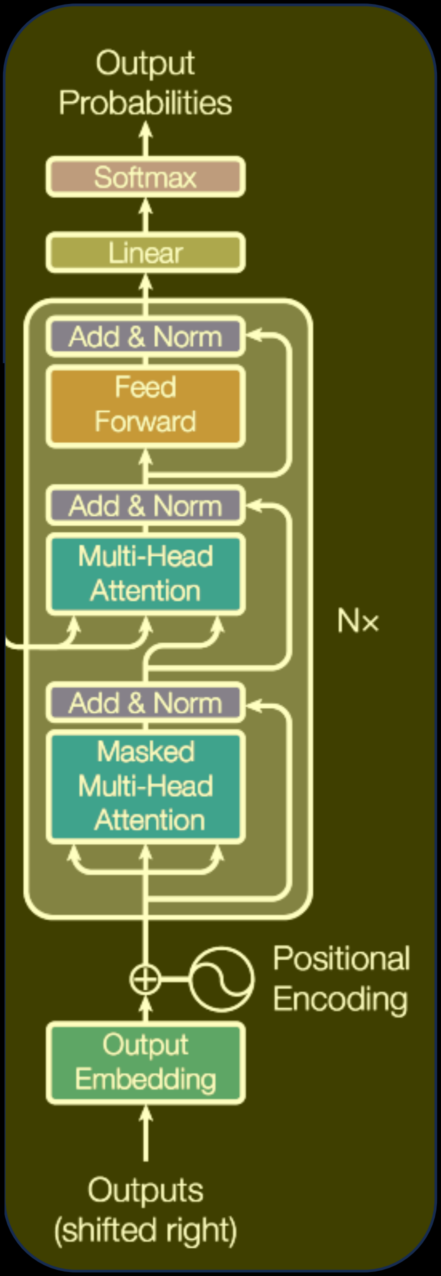
Figure 1: The Trans

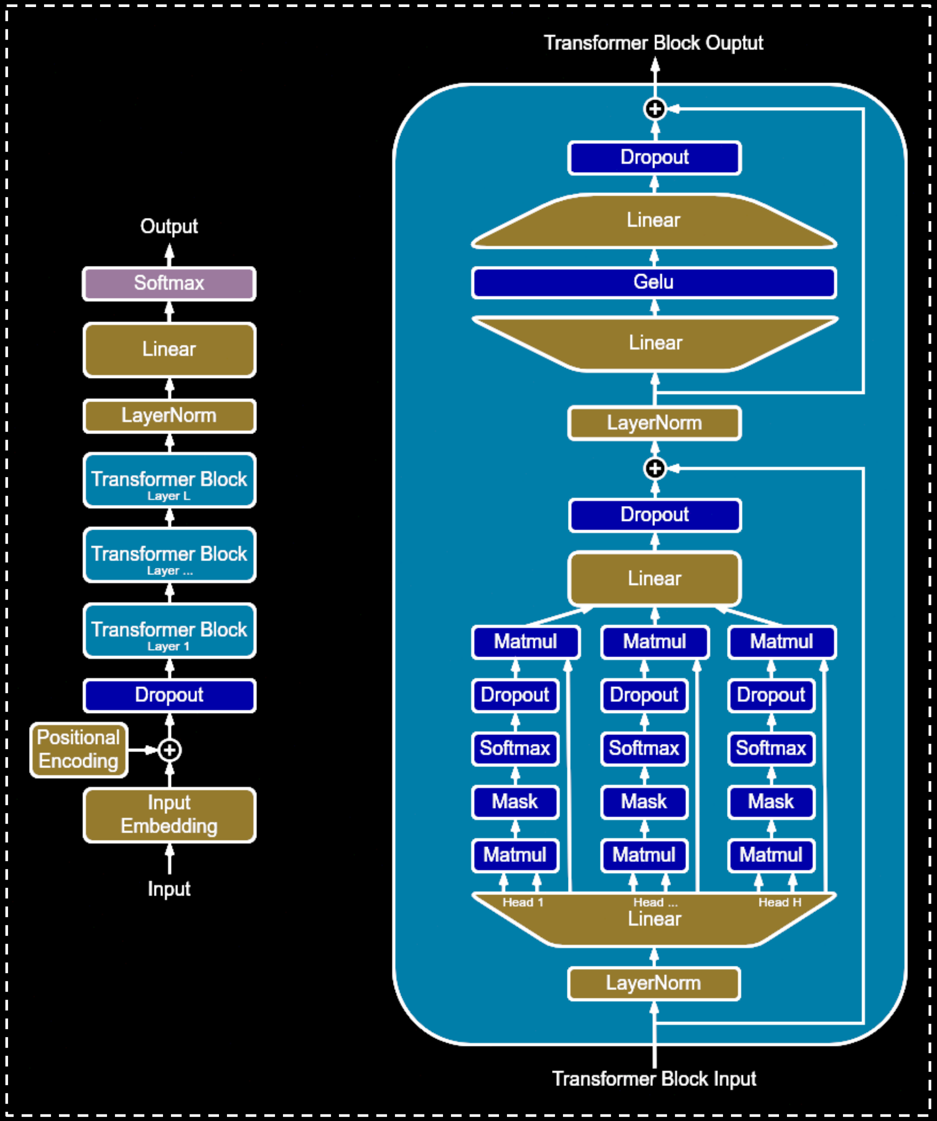
GPT



Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Add & Norm
Masked Multi-Head Attention
Positional Encoding
Output Embedding
Outputs (shifted right)

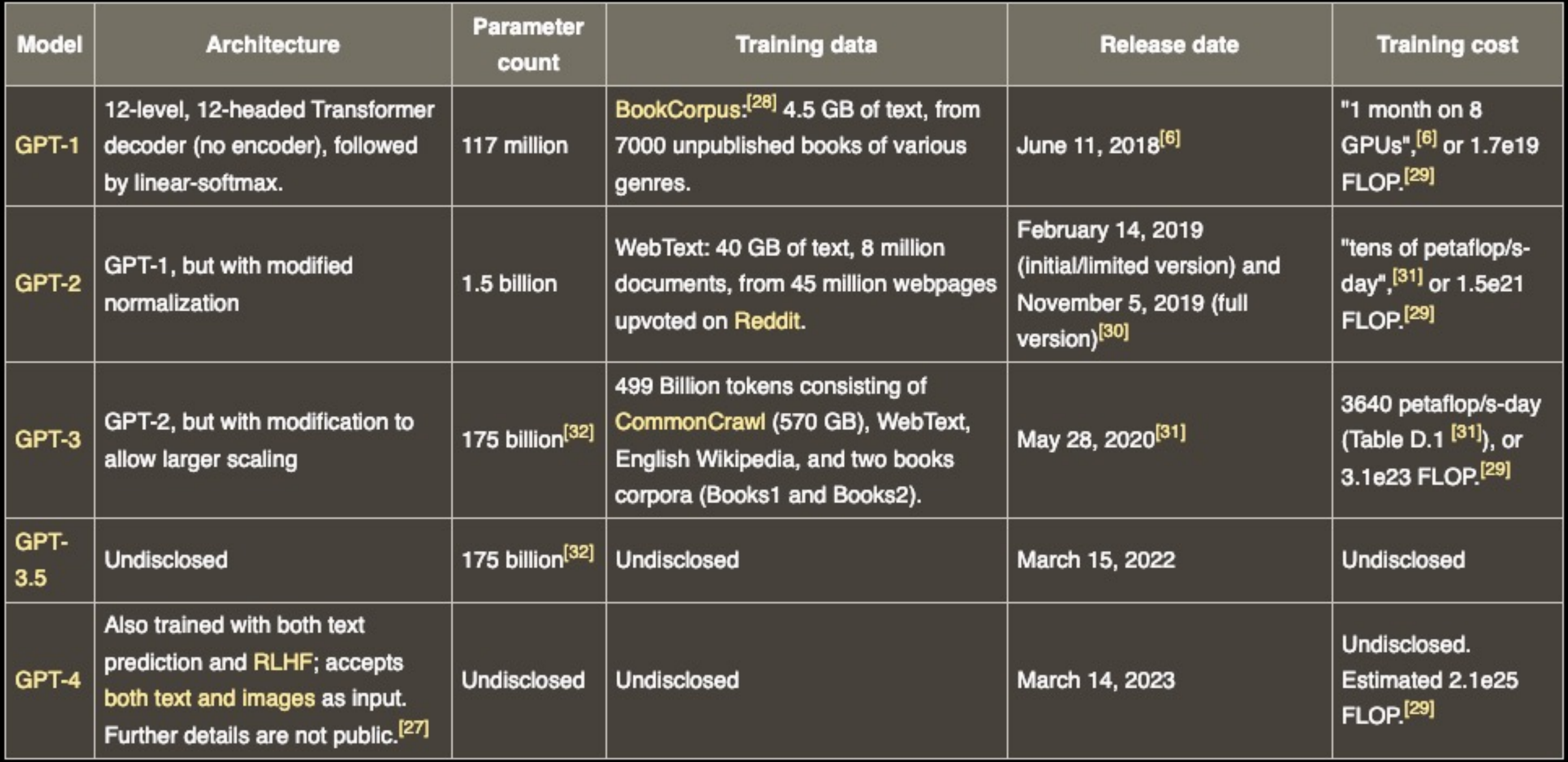
GPT


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Add & Norm
Masked Multi-Head Attention
Positional Encoding
Output Embedding
Outputs (shifted right)


Transformer Block Ouptut
Output
Softmax
Linear
LayerNorm
Transformer Block Layer L
Transformer Block Layer ...
Transformer Block Layer 1
Dropout
Positional Encoding
Input Embedding
Input
Dropout
Linear
Gelu
Linear
LayerNorm
Dropout
Linear
Matmul
Dropout
Softmax
Mask
Matmul
Head 1
Head ...
Head H
Linear
LayerNorm
Transformer Block Input

Model Scales

OpenAI's "GPT-n" series

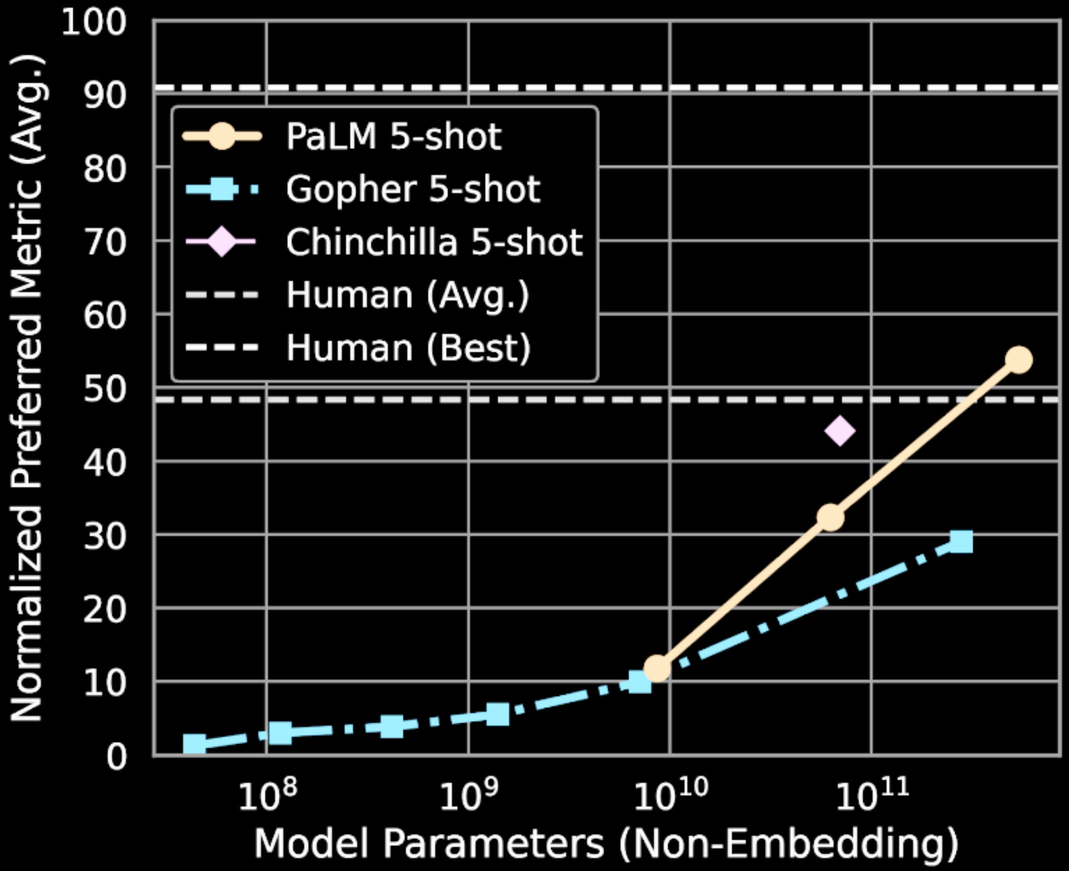


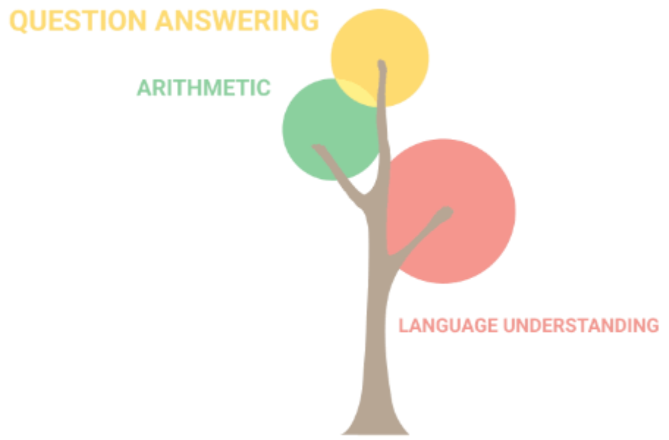
Model	Architecture	Parameter count	Training data	Release date	Training cost
GPT-1	12-level, 12-headed Transformer decoder (no encoder), followed by linear-softmax.	117 million	BookCorpus:[28] 4.5 GB of text, from 7000 unpublished books of various genres.	June 11, 2018[6]	"1 month on 8 GPUs",[6] or 1.7e19 FLOP.[29]
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.	February 14, 2019 (initial/limited version) and November 5, 2019 (full version)[30]	"tens of petaflop/s-day",[31] or 1.5e21 FLOP.[29]
GPT-3	GPT-2, but with modification to allow larger scaling	175 billion[32]	499 Billion tokens consisting of CommonCrawl (570 GB), WebText, English Wikipedia, and two books corpora (Books1 and Books2).	May 28, 2020[31]	3640 petaflop/s-day (Table D.1 [31]), or 3.1e23 FLOP.[29]
GPT-3.5	Undisclosed	175 billion[32]	Undisclosed	March 15, 2022	Undisclosed
GPT-4	Also trained with both text prediction and RLHF; accepts both text and images as input. Further details are not public.[27]	Undisclosed	Undisclosed	March 14, 2023	Undisclosed. Estimated 2.1e25 FLOP.[29]

GPT-3 training cost around $5 million

Training a 540-Billion Parameter Language Model with Pathways

PaLM demonstrates the first large-scale use of the Pathways system to scale training to 6144 chips, the largest TPU-based system configuration used for training to date. The training is scaled using **data parallelism** at the Pod level across two **Cloud TPU v4 Pods**, while using standard data and model parallelism within each Pod. This is a significant increase in scale compared to most previous LLMs, which were either trained on a single TPU v3 Pod (e.g., **GLaM**, **LaMDA**), used pipeline parallelism to scale to 2240 A100 GPUs across GPU clusters (**Megatron-Turing NLG**) or used multiple TPU v3 Pods (**Gopher**) with a maximum scale of 4096 TPU v3 chips.





8 billion parameters

Model scales

GPT-4 “Leaked details”

Model size:

- 1.8 trillion parameters across 120 layers
- Mixture of Expert (MoE):, which consists of 16 different experts working together, each with ~110b parameters and trained for a specific task/field
- It was trained on 13 trillion tokens
- Context length: 8K initially, then 32K after fine-tuning.
- Batch size of 16 million by the end, with each expert seeing a subset of tokens

Hardware & Costs:

- ~ 25,000 Nvidia A100 GPUs for training
- MFU reported at 32% to 36% (low)
- Trained for 90 to 100 days
- Estimated cost to train GPT4 using today’s equivalent hardware (H100 Tensor Core GPU from Nvidia) would be around $63 million.
- Sam Altman stated that the cost of training GPT-4 was more than **$100 million**

Open-Source LLMS

- Leaked (NOT Open): LLaMA v1 (Large Language Model Meta AI): 7b, 13b, 65b

- Open version of LLaMA:
  1. Framework / Code: Lit-LlaMA
  2. Dataset: Red Pajama
  3. Model's weights: Open LlaMA

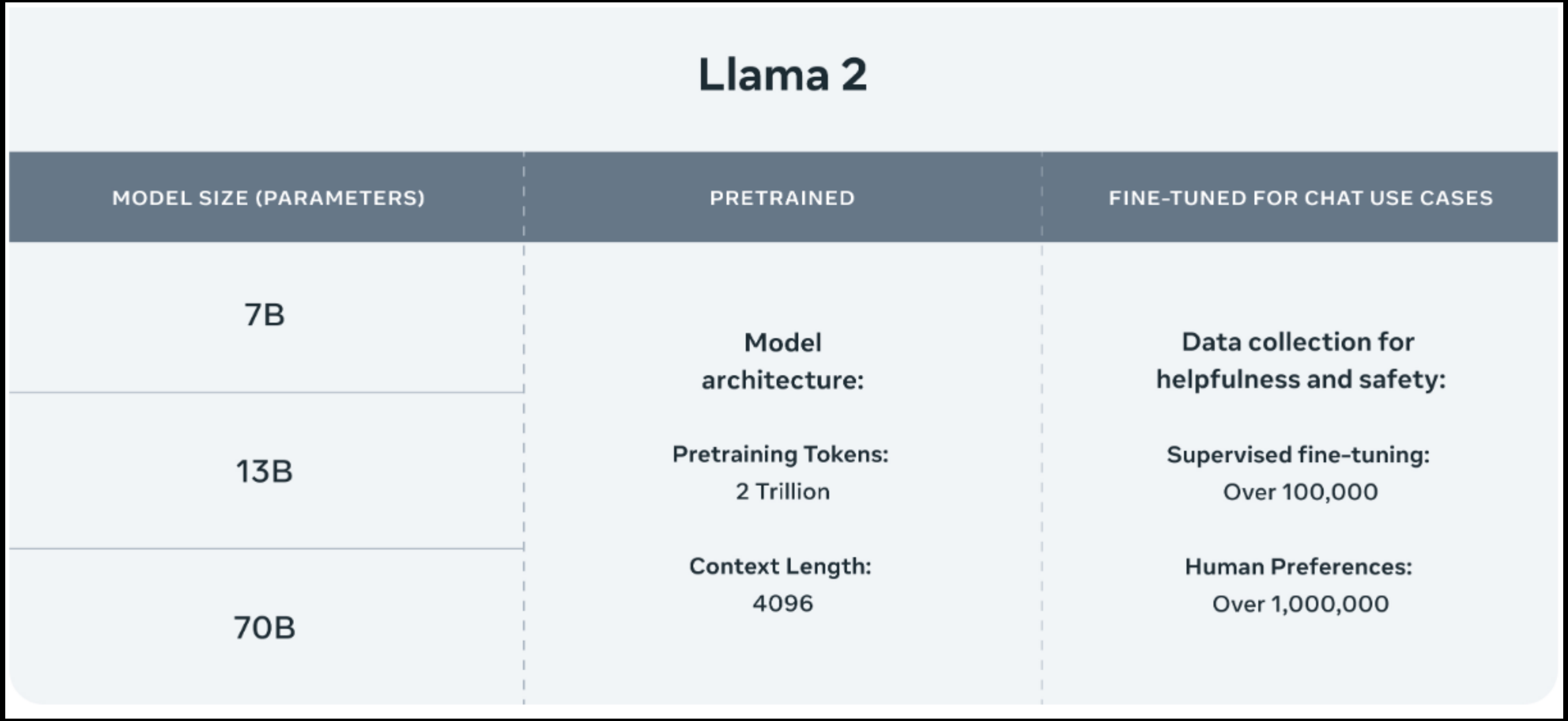
- Falcon (TiiUAE): 7b, 40b

- LLaMA v2 (Meta)

- FreeWilly2 (Stability.ai): LLaMA2 trained using Orca-style dataset

Hugging Face leaderboard: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

Open-Source LLMS



Llama 2

MODEL SIZE (PARAMETERS)	PRETRAINED	FINE-TUNED FOR CHAT USE CASES
7B	**Model architecture:** **Pretraining Tokens:** 2 Trillion **Context Length:** 4096	**Data collection for helpfulness and safety:** **Supervised fine-tuning:** Over 100,000 **Human Preferences:** Over 1,000,000
13B		
70B		

End of Part 3