Large Language Models: Hugging Face

Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux

Some Hugging Face propaganda

- Models
- Pipeline and tasks
- Tokenizers
- Datasets
- Fine-tuning
- Inference

MODELS

Three options

(1) **Inference-as-a-service:** through an API
(2) **On the cloud:** as managed service or custom deployment
(3) **Locally:** possible for small enough models (Ollama)

Hugging Face enables all three options!

HTTPS://HUGGINGFACE.CO/

It is a primarily a GitHub for models, but also develops a lot of useful packages and resources!

Architectures

- **CPUs:** While generally slower for LLMs, they are more accessible and cost-effective for smaller models or less demanding tasks.
- **GPUs:** The most common choice for LLMs, offering significant performance improvements due to their parallel processing capabilities.
- **TPUs:** Google's specialized hardware designed for machine learning, providing even faster performance than GPUs for certain models.
- **Distributed Systems:** Multiple processors (CPUs, GPUs, or TPUs) working together to handle large models or high inference demands. Use accelerate: https://huggingface.co/docs/accelerate/index
- **Edge Devices:** Smaller, less powerful devices like smartphones and IoT devices can run optimized LLMs for specific tasks.

How to get GPU resources *for free*?

Anyone: Google colab, Kaggle, Codesphere, Sagemaker…

Academics:

- grid5000 https://www.grid5000.fr/
- Jean-Zay https://www.edari.fr/

How much memory for a 3B model?

The memory required to hold a 3B parameter LLM in memory depends heavily on the **data type** used to store the model weights:

Full Precision (FP32):

- Each parameter requires 32 bits (4 bytes)
- Total memory: 3 billion parameters * 4 bytes/parameter = 12 GB

Heavily Quantized (INT4):

- Each parameter requires 4 bits (0.5 bytes)
- Total memory: 3 billion parameters * 0.5 bytes/parameter = 1.5 GB

This is only for holding the model in memory! You should aim at 4x this for inference and fine-tuning.

Pipeline and tasks

Tasks

1. Text Classification

   - Sentiment analysis: Determining the emotional tone of a text (positive, negative, neutral).
   - Topic classification: Categorizing text into predefined topics.
   - Spam detection: Identifying unsolicited or unwanted messages.
   - Natural language inference: Determining the relationship between two sentences (entailment, contradiction, neutral).

2. Token Classification

   - Named entity recognition (NER): Identifying and classifying named entities in text (people, organizations, locations, etc.).
   - Part-of-speech (POS) tagging: Assigning grammatical tags to words (noun, verb, adjective, etc.).

3. Question Answering

   - Extractive question answering: Finding the answer to a question within a given text.
   - Multiple choice question answering: Selecting the best answer from a set of options.

More tasks

4. Text Generation

   - Text summarization: Generating a concise summary of a longer text.
   - Translation: Translating text from one language to another.
   - Dialogue generation: Creating conversational responses in a chatbot.
   - Code generation: Generating code in various programming languages.

5. Text2Text Generation

   - Paraphrasing: Rewriting a text while preserving its meaning.
   - Summarization: Generating a concise summary of a longer text.
   - Translation: Translating text from one language to another.

6. Fill-Mask

   - Masked language modeling: Predicting missing words in a text.

7. Feature Extraction

   - Generating embeddings: Creating numerical representations of text for use in other machine learning tasks.

Pipeline: concise but little control

```python
from transformers import pipeline

# Create a sentiment analysis pipeline
classifier = pipeline(task="sentiment-analysis",
                      model="distilbert/distilbert-base-uncased-finetuned-sst-2-english")

# Run inference
result = classifier("This course is f***ing great!")

# Print the result
print(result)
```

```
[{'label': 'POSITIVE', 'score': 0.9998470544815063}]
```

You can choose the device (CPU, GPU)

```python
from transformers import pipeline

# Create a sentiment analysis pipeline
classifier = pipeline(task="sentiment-analysis",
                      model="distilbert/distilbert-base-uncased-finetuned-sst-2-english",
                      device="cuda")

# Run inference
result = classifier("This course is f***ing great!")

# Print the result
print(result)
```

The same model can be used for different tasks!

!! IMPORTANT !! Set `torch_dtype="auto"` to load the weights in the data type defined in a model's `config.json` file to automatically load the most memory-optimal data type.

```python
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(model="distilbert/distilbert-base-uncased",
                                                           torch_dtype="auto")
```

```python
from transformers import AutoModelForTokenClassification

model = AutoModelForTokenClassification.from_pretrained("distilbert/distilbert-base-uncased",
                                                        num_labels = 5,
                                                       torch_dtype="auto")
```

WTF?

```
from transformers import AutoModelForTokenClassification

model = AutoModelForTokenClassification.from_pretrained("distilbert/distilbert-base-uncased",
                                                        num_labels = 5,
                                                        torch_dtype="auto")
```


```
Some weights of DistilBertForTokenClassification were not initialized from the model checkpoint at distilbert/dist
ilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

Hugging Face: “Don’t worry, this is completely normal! The pretrained head of the BERT model is discarded, and replaced with a randomly initialized classification head. You will fine-tune this new model head on your sequence classification task, transferring the knowledge of the pretrained model to it.”

Serverless inference API: slow but free

```python
from huggingface_hub import InferenceClient

client = InferenceClient(
    "cardiffnlp/twitter-roberta-base-sentiment-latest",
    token="TO BE FILLED HERE",
)

client.text_classification("Today is a great day")
```

```
[TextClassificationOutputElement(label='positive', score=0.9836677312850952),
 TextClassificationOutputElement(label='neutral', score=0.01135887298732996),
 TextClassificationOutputElement(label='negative', score=0.004973393864929676)]
```

Pipeline is good to get started

But soon you feel limited.

Let’s see how to get a bit more control!

Tokenizers

Only dealing with text here...

The tokenizer is for dealing with texts. For other formats:

- Speech and audio, use a Feature extractor to extract sequential features from audio waveforms and convert them into tensors.
- Image inputs use a ImageProcessor to convert images into tensors.
- Multimodal inputs, use a Processor to combine a tokenizer and a feature extractor or image processor.

```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-uncased")
tokenizer
```

```
BertTokenizerFast(name_or_path='google-bert/bert-base-uncased', vocab_size=30522, model_max_length=512, is_fast=Tr
ue, padding_side='right', truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pa
d_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True),  added_token
s_decoder={
        0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
}
```

```python
sequence = "In a hole in the ground there lived a hobbit."
print(tokenizer(sequence))
```

```
{'input_ids': [101, 1999, 1037, 4920, 1999, 1996, 2598, 2045, 2973, 1037, 7570, 10322, 4183, 1012, 102], 'token_ty
pe_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1]}
```

The nasty business of data preprocessing

You want to create batches (a lot more efficient!). This means that you may need to:

- Pad: add a special token [PAD] to make sure all inputs have the same size
- Truncate: if some inputs are larger than the context length of your model, you need to break them up into more inputs (but it's not that simple: better introduce some overlapping!)

Good news: Tokenizer does that for you!

```
batch_sentences = [
    "But what about second breakfast?",
    "Don't think he knows about second breakfast, Pip.",
    "What about elevensies?",
]
encoded_input = tokenizer(batch_sentences, padding=True, truncation=True)
print(encoded_input)
```

```
{'input_ids': [[101, 2021, 2054, 2055, 2117, 6350, 1029, 102, 0, 0, 0, 0, 0, 0], [101, 2123, 1005, 1056, 2228, 200
2, 4282, 2055, 2117, 6350, 1010, 28315, 1012, 102], [101, 2054, 2055, 5408, 14625, 1029, 102, 0, 0, 0, 0, 0, 0,
0]], 'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0], [1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]]}
```

Datasets

```python
from datasets import load_dataset

dataset = load_dataset("yelp_review_full")
dataset["train"][100]
```

```
{'label': 0,
 'text': 'My expectations for McDonalds are t rarely high. But for one to still fail so spectacularly...that takes
something special!\\nThe cashier took my friends\'s order, then promptly ignored me. I had to force myself in fron
t of a cashier who opened his register to wait on the person BEHIND me. I waited over five minutes for a gigantic
order that included precisely one kid\'s meal. After watching two people who ordered after me be handed their foo
d, I asked where mine was. The manager started yelling at the cashiers for \\"serving off their orders\\" when the
y didn\'t have their food. But neither cashier was anywhere near those controls, and the manager was the one servi
ng food to customers and clearing the boards.\\nThe manager was rude when giving me my order. She didn\'t make sur
e that I had everything ON MY RECEIPT, and never even had the decency to apologize that I felt I was getting poor
service.\\nI\'ve eaten at various McDonalds restaurants for over 30 years. I\'ve worked at more than one location.
I expect bad days, bad moods, and the occasional mistake. But I have yet to have a decent experience at this stor
e. It will remain a place I avoid unless someone in my party needs to avoid illness from low blood sugar. Perhaps
I should go back to the racially biased service of Steak n Shake instead!'}
```

Fine-tuning

TrainingArguments

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="checkpoints",
    learning_rate=2e-5,
    eval_strategy="epoch",
    save_strategy="epoch",
    logging_strategy="epoch",
    per_device_train_batch_size=32,
    per_device_eval_batch_size=16,
    num_train_epochs=10,
    weight_decay=0.01,
    report_to="none",
)
```

Trainer

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=small_train_dataset,
    eval_dataset=small_eval_dataset,
    compute_metrics=compute_metrics,
)
```

```
trainer.train()
```

Parameter-Efficient Fine-tuning (PEFT)

Load models, two options

Option 1: load a PEFT adapter

```
from transformers import AutoModelForCausalLM, AutoTokenizer

peft_model_id = "ybelkada/opt-350m-lora"
model = AutoModelForCausalLM.from_pretrained(peft_model_id)
```

Option 2: Load the model and its adapter

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model_id = "facebook/opt-350m"
peft_model_id = "ybelkada/opt-350m-lora"

model = AutoModelForCausalLM.from_pretrained(model_id)
model.load_adapter(peft_model_id)
```

```python
from transformers import AutoModelForSeq2SeqLM

model = AutoModelForSeq2SeqLM.from_pretrained("bigscience/mt0-small")
```

```python
from peft import LoraConfig

peft_config = LoraConfig(
    lora_alpha=16,
    lora_dropout=0.1,
    r=64,
    bias="none",
    task_type="CAUSAL_LM",
)
```

```python
from peft import get_peft_model

model = get_peft_model(model, peft_config)
model.print_trainable_parameters()
```

```
trainable params: 2,752,512 || all params: 302,929,280 || trainable%: 0.9086
```

The rest is as before: TrainingArguments and Trainer

Inference

Paged attention

Key idea: when generating many tokens with the same prompt, many values can be cached! The keys and queries are augmented with new values rather than recomputed.

Doing it efficiently in a GPU-friendly way is non-trivial…

vLLM

Faster inference: https://github.com/vllm-project/vllm

Retrieval-augmented Generation (RAG)

The idea

Allow the LLM to access an external source of knowledge, later referred to as corpus.

Main application: navigate long documents (manuals, regulations,…)

Does it reduce hallucination? Yes. Is it *the* solution? **No!**

Generic framework

We have a vectorstore, which is a database (key, value) where:

- **Keys** are “embedding vectors”
- **Values** are “chunks”

An *embedding vector* is a vector of float of fixed dimension.

A *chunk* is a piece of text extracted from the corpus.

Preprocessing

This is the process of populating the dataset. It involves:

- **Text cleaning and splitting**: from a document to a set of chunks
- **Embedding**: computing embeddings for each chunk

Preprocessing: text splitting

Naive: fix chunk length and split

Better:

- split on new lines
- include chunk overlap

Preprocessing: embedding

Pre-LLM:

- Word2vec: uses a simple RNN
- GloVe: Global Vectors for Word Representation

Post-LLM:

- BERT, for instance ModernBert (see MTEB for benchmarks)

Preprocessing: contextual embedding

A simple idea by Anthropic (Claude): instead of embedding the chunk itself, we:

- Ask an LLM to produce a description of the chunk (using the chunk **and** the full document)
- Embed the description + the chunk

Processing

At inference time:

- **Query**: from the prompt we construct queries to the database
- **Retrieve**: we retrieve the most relevant chunks
- **Answer:** we use the added contents to formulate an answer to the prompt

Processing: Query

Naive: turn the prompt into a query

Better:

- Ask an LLM to formulate a query from the prompt
- HyDE: ask an LLM to generate a hypothetical document, embed this document, and retrieve similar documents

Processing: Retrieve

Naive:

- Choose a notion of similarity between embedding vectors
- Retrieve the K-nearest neighbours

Better:

- For scaling: use approximation algorithms
- For diversity: use an SVM

Processing: notions of similarity

There are many notions of similarity:

- Cosine similarity
- Dot product
- Euclidean distance

Remark: when vectors are normalized, cosine similarity coincides with dot product

Processing: Answer

Naive: add the most relevant chunks to the prompt

Better:

- Ask an LLM, called a reranker, to filter and rearrange the most relevant chunks
- Perform a BM25 on the side, which operates on keywords, and merge the resulting most relevant chunks

From unstructured to structured RAG

We only discussed the case where the corpus is unstructured.
If it is structured, there are more specific techniques…

In practice

Two frameworks for building RAGs:

- LangChain
- LlamaIndex

They do everything for you, sometimes not leaving enough control…

The DeepSeek special

References

DeepSeek papers:

- MoE: https://arxiv.org/abs/2401.06066 (early 2024)
- GRPO: https://arxiv.org/abs/2402.03300 (early 2024)
- MLA: https://arxiv.org/abs/2405.04434 (mid 2024)
- MTP: https://arxiv.org/abs/2412.19437 (end of 2024)
- DeepSeek-R1: https://arxiv.org/abs/2501.12948 (early 2025)

Implementations:

- Open-R1: https://github.com/huggingface/open-r1
- verl: https://github.com/volcengine/verl
- trl: https://github.com/huggingface/trl

Two directions

(1) Architecture:

- Mixture of Experts (MoE)
- Multi-head Latent Attention (MLA)
- Multi-Token Prediction (MTP)

(2) Group relative policy optimization (GRPO)

Mixture of experts

The general idea of mixture of experts (MoE)

The model is composed of:

- A set of **experts**, which are independent submodels
- A **router** (also called gating network)

The input is fed to the router, which determines a weight for each expert. The input is fed to the K experts with highest weights. Their outputs are aggregated using these weights.

Intuitively, each expert specialises, and the router is able to predict which expert computes relevant information.

Pros and cons

+ **Specialization**: Experts can specialize in different aspects of the problem, leading to better performance than a single, general-purpose model.
+ **Scalability**: MoE can scale to handle very complex problems by adding more experts.
+ **Efficiency**: For a given input, only a subset of the experts needs to be activated, which can improve efficiency compared to a model where all parameters are used for all inputs.
+ **Improved Capacity**: MoEs can have a much larger total capacity (number of parameters) than a single model, without a proportional increase in computational cost per example.
- **Complexity**: Training MoE models can be more complex than training standard models, as the gating network and the experts need to be trained jointly.
- **Data Sparsity**: If the experts are too specialized, they might not receive enough training data, leading to poor performance. This is related to the routing decision.
- **Routing Challenge**: The gating network needs to learn to route inputs effectively. Poor routing can lead to suboptimal performance.

MoE for transformers

- MoE layers replace MLP layers
- Each expert is an MLP
- The router is also an MLP with a softmax

Each token of the input is fed to the router, which determines a weight for each expert. The token is fed to one or two experts with highest weights. Their outputs is aggregated using these weights.

Deepseek's MoE: fine-grained expert segmentation

Issue: each token gets sent to a very small number of experts.

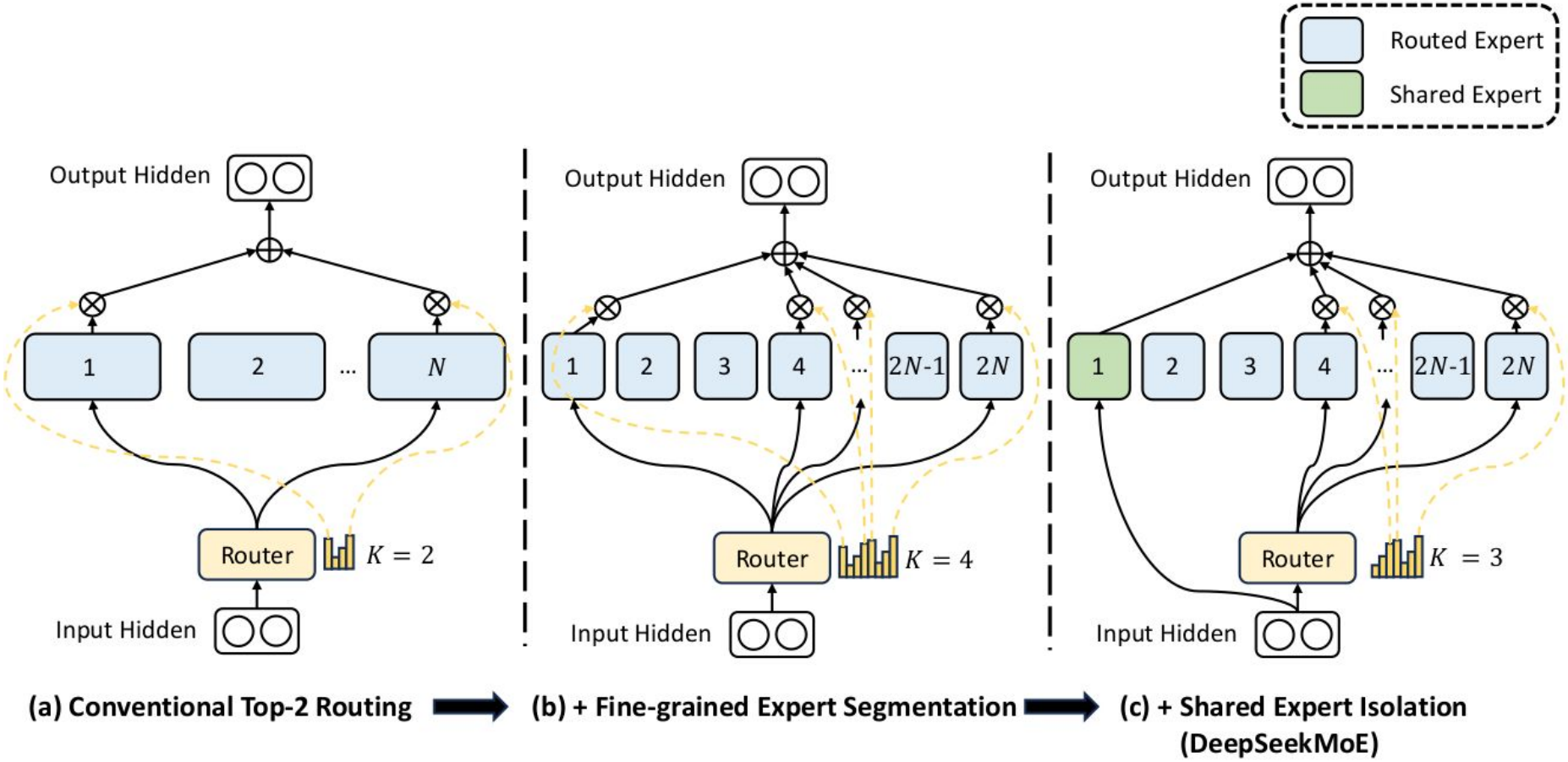
Idea: keeping the number of parameters constant, we increase the number of experts. Nothing else changes, but this way, each token gets sent to more experts.

DeepSeek's MoE: shared expert isolation

Issue: tokens assigned to different experts may require common knowledge.

Idea: introduce shared experts that are used for each token.

(a): N experts (b): 2N experts (c): 2N routed + 1 shared



Hidden under the carpet

We need to make sure that:

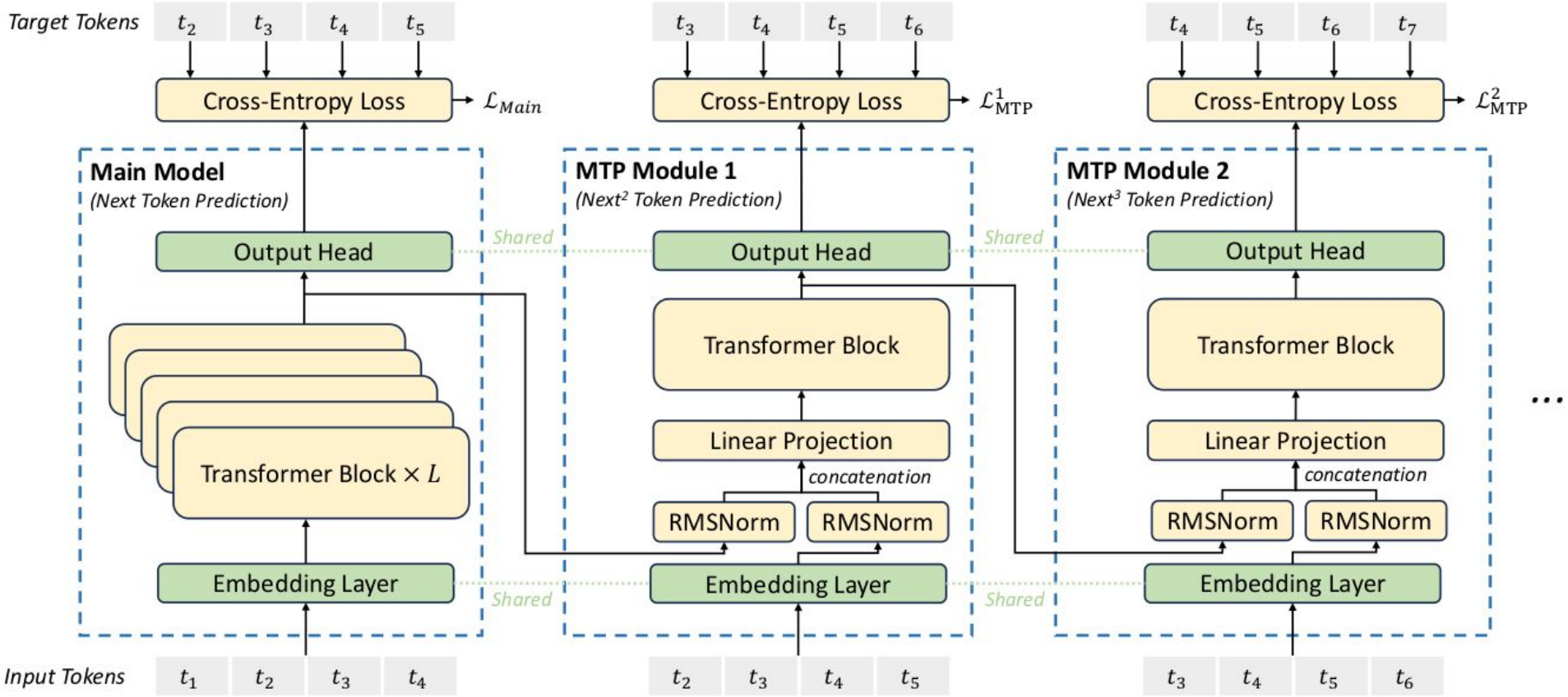
- Each expert gets enough training to avoid *routing collapse*
- Tokens inside a sequence are spread to different experts to avoid *computation bottlenecks*

Classical solution: auxiliary losses, which degrade performances.

DeepSeek's approach: adding dynamic bias for each expert

Multi-token prediction

Can we predict multiple tokens at once?



Multi-head Latent attention

It's a long (and still developing) story

- It starts with KV cache, which caches keys and values when generating long sequences
- But KV cache uses a lot of memory, so different methods were proposed to reduce memory, such as Multi-Query Attention (MQA) vs Grouped-Query Attention (GQA)
- Multi-head Latent Attention (MLA) is another attempt to lower memory, by projecting up and down in a latent space

Some pointers

- https://huggingface.co/blog/kv-cache-quantization
- https://towardsdatascience.com/deepseek-v3-explained-1-multi-head-latent-attention-ed6bee2a67c4/

Group relative Policy optimization (GRPO)

The three stages of understanding

(1) What you tell your grandparents about Deepseek
(2) The high-level ideas
(3) The fineprints

What is the goal?

Short version: post-training **reasoning** models

(not all models need to reason!)

Long version: we start from either a foundation model or an instruct model, and we want to teach the model to **reason** to solve maths, logic, or programming tasks.

Important: We will ask the LLM to think before giving an answer (chain of thoughts).

FOR YOUR GRANDPARENTS

The version for your grandparents (1/4)

I’m teaching my 5 year old daughter additions. Here are three approaches. In each case I give her an example (“12 + 19 = ?”) and I ask her to think and give me an answer.

- v0: I explain how I perform the addition (“12 + 19 = 31: I first add the units, remember the carry…”)
- v1: I evaluate her reasoning and reward her when both are correct.
- v2: I ignore the reasoning and reward her when the answer is correct.

The version for your grandparents (2/4)

Believe it or not:

- v0 is absolutely useless, she gets bored very quickly with my fathersplaining
- v1 does not work so much either because she doesn’t like me correcting her reasoning, it is a bit too abstract
- v2 works a lot better: the answers become more and more correct over time, although her explanations are not very convincing (even when the result is correct)

Somehow in v2 I rely on her to improve her reasoning, I do not impose my way of reasoning on her

The version for your grandparents (3/4)

- v0 is called “supervised fine-tuning”
- v1 is called “reinforcement learning with human feedback”, because it uses an advanced reward model (me!)
- v2 is a “reinforcement learning with rule-based reward model”

The observation about my daughter’s explanations mirrors the DeepSeek’s observations: the model needs reasoning to improve its performance, but it becomes ununderstandable, the model develops its own language

The version for your grandparents (4/4)

To move towards the “group relative policy optimization” developed by DeepSeek, the analogy breaks: I can ask the same question to an LLM and collect different answers (my daughter refuses to do that!), because they are stochastic models

The algorithm goes as follows: I collect 100 responses for a fixed question, compute the average score (only based on answer’s correctness), and then reward based on the “advantage” of each response, which is its difference to the average score

High-level ideas

Different approaches for post-training

- Supervised fine-tuning (SFT)
- Reinforcement Learning from Human Feedback (RLHF)
- Direct Preference Optimization (DPO)
- Group Relative Policy Optimization (GRPO)

What the different approaches have in common

Each approach defines a loss function to be minimised, which can be more of less complex and directly related to the objective (solving problems!).

All approaches will apply the following algorithm:

- Sample some data (more precisely, a batch of data)
- Compute the gradient of the loss with respect to the parameters of the model
- Apply a gradient step to update the parameters

Option 1: Supervised fine-tuning (SFT)

The baseline approach:

- **Collect data**: construct a dataset of pairs (prompt, response)
- **Train**: classical fine-tuning, teach the model how to respond to each prompt

Issues with SFT

- Need to have a clean, large dataset
- Not well suited for reasoning: there are many ways of getting to the right answer
- Does not take into account aligning with human preferences
- May induce catastrophic forgetting: we can include a penalty term in the loss for not deviating too much from the original model

Option 1 bis: distillation

Distillation is a technique for training a smaller, faster, and more efficient model (the “student”) by transferring knowledge from a larger, more complex model (the “teacher”)

There are (at least) two different understandings of what this means:

- The traditional one
- The data augmentation one

Option 1 bis: data-augmentation distillation

In “data-augmentation” distillation, the teacher is used to generate the dataset (either both questions and responses, or only responses).

This is particularly interesting for reasoning models, because good reasoning is hard to come by!

Option 1 bis: traditional distillation

In “traditional” distillation, we have two targets:

- The **hard target** is the ground truth
- The **soft target** is the logits of the teacher

The student learns by minimising a loss consisting of two terms:

- Cross-entropy loss to match the hard target
- Distillation loss to match the soft target

Option 2: Reinforcement Learning from Human Feedback (RLHF)

Reference: https://arxiv.org/abs/1909.08593

- **Collect data**: construct a dataset of pairs (prompt, sets of responses)
- **Collect human data**: ask humans for each prompt to rank responses
- **Train a reward model**: the reward model takes as input a prompt and a response, and returns a reward (numerical score)
- **Train the model**: fine-tune the model with an RL algorithm to optimize rewards

Flash introduction to Reinforcement Learning

An agent evolves in an (unknown) environment by taking actions through a policy. In a single step, from state s playing action a we get reward r and go to state s'

The goal of the agent is to maximise the total reward:

π: policy

$\rho = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$: trajectory

Objective:

$$\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} r_t \right]$$

Which RL algorithm for RLHF?

RLHF is parameterized by the RL algorithm used. Classical choices include:

- Deep Q-Networks (DQN), the classic
- Proximal Policy Optimization (PPO), the default option

Issues with RLHF

- Humans are expensive!
- Training a reward model is hard, unreliable, and costly
- Need to be careful about rewards
- RL itself is hard

A small note

Here we see RLHF just as a fine-tuning algorithm. A slightly different point of view on RLHF:

- **Pre-training**: teaching the LLM language (through language modelling, meaning next token prediction)
- **Fine-tuning**: instructing the LLM on downstream tasks
- **Alignement**: ensures that the model aligns with human values

Option 3: Direct preference optimization (DPO)

Reference: https://arxiv.org/abs/2305.18290

Key idea: get rid of the reward model

- **Collect data**: construct a dataset of pairs (prompt, pairs of responses), with one response preferred the other
- **Train**: maximize the probability of generating preferred responses

Issues with DPO

Requires a human to determine which of the two responses are better

Option 4: Group Relative Policy Optimization (GRPO)

Reference: https://arxiv.org/abs/2402.03300

Key idea: Instead of trying to assign an absolute “goodness” score to each response (like a reward model does), GRPO focuses on relative comparisons within a group of responses.

Option 4: Group Relative Policy Optimization (GRPO)

Key assumption: we can evaluate the final result (**but not the reasoning!**). Examples:

- maths problems: the answer is a number
- code problems: the answer is a piece of code, which can be executed and tested against example inputs
- logical / reasoning problems: the answer is a value

In other words: **rule-based reward model** instead of **model-based reward model**

Option 4: Group Relative Policy Optimization (GRPO)

- **Collect data**: construct a dataset of pairs (prompt, answer)

!!!IMPORTANT!!! It is a lot easier if you do not need to provide the reasoning, just the answer!

- **Group evaluation**: given a prompt, ask the model to generate a group of responses. Evaluate each response only based on the answer (not the reasoning!). Averaging yields a reference point
- **Relative advantage**: Compute the advantage of each response relative to the group
- **Train**: update the probability of generating responses based on their advantages

Observation about GRPO

The first experiment is to fine-tune the V3 model using GRPO, leading to R1-Zero. The results are:

- The performances on answers are impressive
- The thinking time grows larger over the course of the training
- The model learns reflection and develops reasoning approaches
- ***BUT*** explanations become poor and it mixes language

THE FINE PRINTS

Option 1: Supervised fine-tuning (SFT)

$\mathcal{D}$: distribution of pairs (prompt, response)

π_θ: model with parameters θ

Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\frac{1}{|y|}\sum_{t=1}^{|y|}\log \pi_\theta(y_t \mid x, y_{<t})\right]$$

Option 2: Reinforcement Learning from Human Feedback (RLHF)

Three steps:

- Learning a reward model
- Adding KL-constraint to the reward
- Applying an RL algorithm

Step 1: Learning a reward model

$\mathcal{D}$: distribution of triples (prompt, better response, worse response)

r_ϕ: reward model with parameters ϕ.

r^*: latent reward model

$r^*(x, y)$ is the reward of response y to prompt x

Bradley-Terry model:

$$p^*(y_1 > y_2 \mid x) = \frac{\exp(r^*(x, y_1))}{\exp(r^*(x, y_1)) + \exp(r^*(x, y_2))}$$

σ: logistic function $\sigma(x) = \frac{1}{1+\exp(-x)}$

Loss:

$$\mathcal{L}(\phi) = -\mathbb{E}_{(x,y_b,y_w)\sim\mathcal{D}} \left[\log \sigma(r_\phi(x, y_b) - r_\phi(x, y_w))\right]$$

Step 2: Adding a KL-constraint to the reward

We have learned a reward model r(x,y)

Important: r (typically) gives very sparse reward, only when y is entirely generated!

To keep close to the original model, we add a “per-token KL penalty”

Step 2: Adding a KL-constraint to the reward

KL = Kullback-Leibler divergence

D_KL(P || Q) measures how far is the model probabilistic distribution Q far from the true distribution P

See: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

Step 2: Adding a KL-constraint to the reward

$\mathcal{D}$: distribution of prompts
r: learned reward model
π_{ref}: reference model (frozen)
π_θ: model with parameters θ
RL loss

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D},y\sim\pi_\theta(x)}\left[r(x,y)\right]$$

KL-constrained loss

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D},y\sim\pi_\theta(x)}\left[r(x,y) - \beta D_{\text{KL}}(\pi_\theta(y \mid x) \mid\mid \pi_{\text{ref}}(y \mid x))\right]$$

Step 2: Adding a KL-constraint to the reward

A small detail:

We use a (differentiable) estimate for the KL term:

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(x)}\left[r(x,y) - \beta \sum_{t=1}^{|y|} \log \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}\right]$$

Step 3: Proximal Policy Optimization (PPO)

First: We'll discuss PPO for RL

Second: We'll see what this means for LLMs

Policy gradient methods

PPO is a “policy gradient method”, meaning it iterates over policies and applies gradient descent to improve policies:

π_θ: policy with parameters θ

$\rho = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$: trajectory

Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} r_t \right]$$

The advantage function

Key quantity: the advantage function quantifies the *relative* benefit of an action

π: current policy
$V_\pi(s)$: expected return from s using π
$Q_\pi(s, a)$: expected return from s playing a and then using π
Advantage function:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$$

Computing the gradient

Lemma: π_θ: policy with parameters θ

$\rho = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$: trajectory

Gradient:

$$\nabla_\theta \mathcal{L} = -\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} A_{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right]$$

We use here the advantage because it minimises variance, but we could use other so-called “baselines”. See here for a proof: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

The hacky loss

We started from the loss defined as:

$$\mathcal{L}(\theta) = -\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} r_t \right]$$

And then computed its gradient. Now, it turns out we can define a *different* function, which is not at all a loss (it can be positive or negative!), but has the right gradient:

$$\mathcal{L}(\theta) = -\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} A_{\pi_\theta}(s_t, a_t) \log \pi_\theta(a_t \mid s_t) \right]$$

Advantage estimates

Key question: how do we estimate the advantage?

Probabilistic estimates are about finding a tradeoff between:

- **Bias**: how close if your estimate to the true value? (*unbiased* = equal in expectation)
- **Variance**: how much your estimates depend on chance?

Generalized Advantage Estimation (GAE)

Idea: GAE adds a discount lambda over future steps to balance *bias* (how accurate is the estimate) and *variance* (how the estimates vary).

- For lambda = 0: only considers the next step (*high bias, zero variance*)
- For lambda = 1: considers all future steps (*no bias, high variance*)

Reference: https://arxiv.org/abs/1506.02438

Generalized Advantage Estimation (GAE)

The case lambda = 0:

$V_{\pi,\gamma}$: latent value function for policy π with discount γ

V: approximate value function

$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$: TD residual of V with discounted γ

If $V = V_{\pi,\gamma}$, then δ_t is an unbiased estimate of $A_{\pi,\gamma}$:

$$\begin{aligned}
\mathbb{E}_{s_{t+1}\sim\pi}\left[\delta_t\right] &= \mathbb{E}_{s_{t+1}\sim\pi}\left[r_t + \gamma V_{\pi,\gamma}(s_{t+1}) - V_{\pi,\gamma}(s_t)\right] \\
&= \mathbb{E}_{s_{t+1}\sim\pi}\left[Q_{\pi,\gamma}(s_t, a_t) - V_{\pi,\gamma}(s_t)\right] \\
&= A_{\pi,\gamma}(s_t, a_t)
\end{aligned}$$

Generalized Advantage Estimation (GAE)

General case

$\lambda \in [0, 1]$ hyperparameter

$V_{\pi,\gamma}$: latent value function for policy π with discount γ

V: approximate value function

$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$: TD residual of V with discounted γ

GAE:

$$\widehat{A}(t, \gamma, \lambda) = \sum_{\ell=0}^{\infty} (\gamma\lambda)^{\ell} \delta_{t+\ell}$$

Generalized Advantage Estimation (GAE)

Long story short for GAE:

- If we have an estimate for the (discounted) value function,
- Then we can estimate the advantage

All neat and tidy, but this means that we need a model for estimating the value function…

Vanilla policy gradient

At this point we have a “vanilla policy gradient” algorithm.

Key issue: after each update we need to recompute the gradient. If we perform multiple updates, we face performance collapse (by diverging too far from the existing policy).

Question: can we learn more from data, making multiple updates on the same datapoint?

Surrogate objective

The surrogate objective focuses on the update

π_{ref}: reference policy (frozen)
π_θ: policy with parameters θ
Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(s,a)\sim\pi_\theta}\left[\frac{\pi_\theta(a \mid s)}{\pi_{\text{ref}}(a \mid s)} \cdot A_{\pi_\theta}(s,a)\right]$$

Trust region policy optimization (TRPO)

The surrogate objective was introduced for the TRPO algorithm, which added a KL-divergence constraint.

The algorithm is complicated to implement…

Reference: https://arxiv.org/abs/1502.05477

Proximal Policy Optimization (PPO)

The first contribution of PPO: introducing the **clipped surrogate objective**

Key idea: limit the amount the policy can change directly in the objective

Reference: https://arxiv.org/abs/1707.06347

Proximal Policy Optimization (PPO)

π_{ref}: reference policy (frozen)
π_θ: policy with parameters θ
Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(s,a)\sim\pi_\theta}\left[\min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\text{ref}}(a \mid s)} \cdot A_{\pi_\theta}(s,a),\ g(\epsilon, A_{\pi_\theta}(s,a))\right)\right]$$

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0, \\ (1-\epsilon)A & \text{if } A < 0. \end{cases}$$

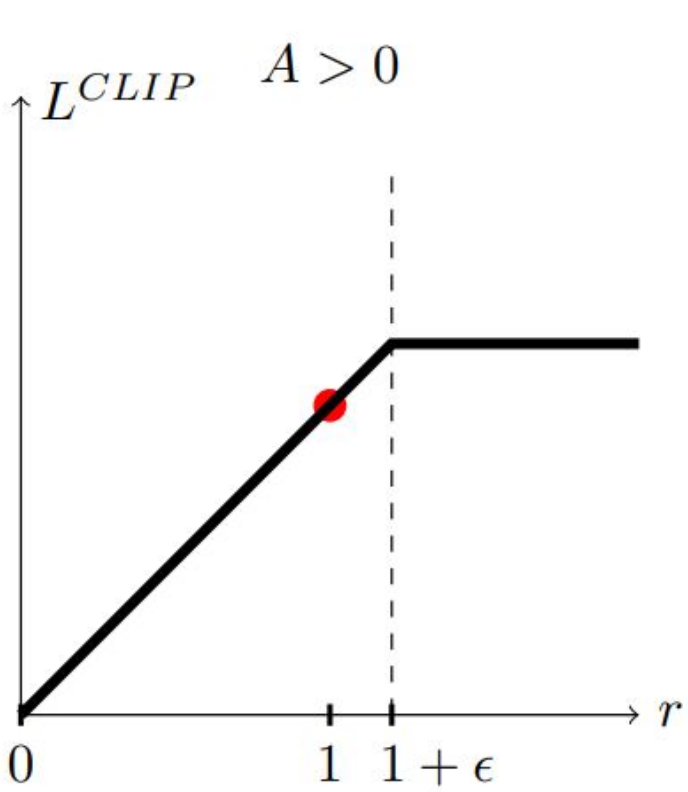
Proximal Policy Optimization (PPO)

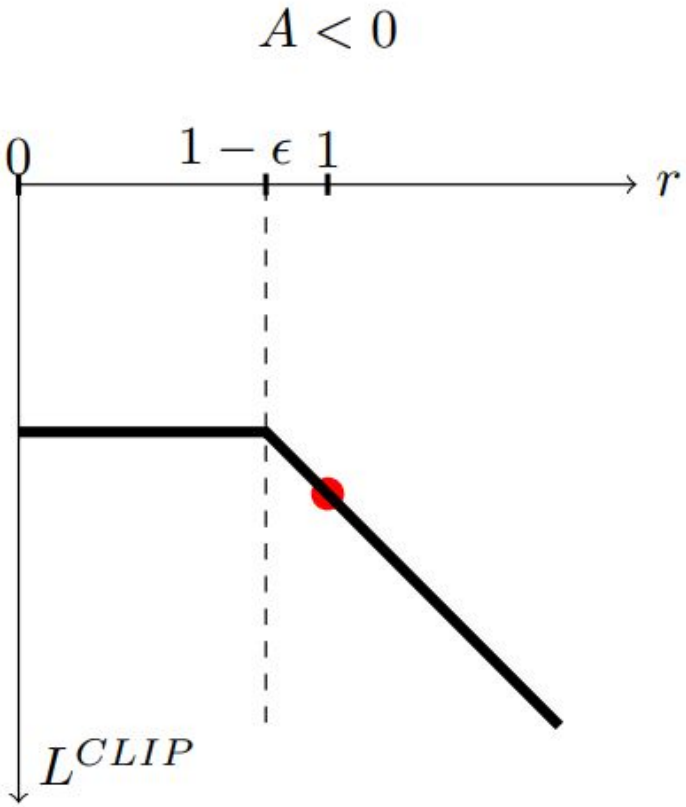
A special case to understand:

π_{ref}: reference policy (frozen)
π_θ: policy with parameters θ
If $A_{\pi_\theta}(s, a)$ is positive:

$$\min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\text{ref}}(a \mid s)},\ 1+\epsilon\right) \cdot A_{\pi_\theta}(s, a)$$

The loss does not grow beyond $(1+\epsilon) \cdot A$


L^{CLIP}
$A > 0$
r
0
1
$1+\epsilon$


$A < 0$
0
$1-\epsilon$
1
r
L^{CLIP}

Proximal Policy Optimization (PPO)

The consequence of the clipped surrogate objective is that we can perform multiple updates using the same data points!

Even better, they can be performed in parallel.

PPO is the standard out-of-the-box SOTA algorithm for RL.

Proximal Policy Optimization (PPO)

Note: PPO can be degenerated to a simple “optimization of the surrogate objective” algorithm by taking at each training step:

reference policy = frozen current policy

(It is a bit of waste but bad implementations do that…)

Step 3: Proximal Policy Optimization (PPO)

Now, let's see what this means for LLMs.

The "state" is the context, the "action" is the next token.

$\mathcal{D}$: distribution of prompts
π_{ref}: reference model (frozen)
π_θ: model with parameters θ
Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D},y\sim\pi_{\text{ref}}(x)}\left[\frac{1}{|y|}\sum_{t=1}^{|y|}\min\left(\frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}\cdot A_{\pi_\theta}((x, y_{<t}), y_t),\ g(\epsilon, A_{\pi_\theta}((x, y_{<t}), y_t))\right)\right]$$

Option 4: Group Relative Policy Optimization (GRPO)

Remember, somewhere in the middle of step 3 for PPO:

- *"We need to estimate advantages (to compute the gradient)"*
- *"GAE can do that if we have estimates for the value function"*

But this is computationally very expensive: it requires training another model for value estimates!

Starting point of GRPO: can we get rid of the value function?

Option 4: Group Relative Policy Optimization (GRPO)

Key idea: the value function is used as a “baseline”.

Here is another (unbiased!) estimate for the value function:

- Given a prompt, generate several responses
- Compute their individual rewards (using a rule-based reward model)
- An estimate of the value function is the average score

Option 4: Group Relative Policy Optimization (GRPO)

$\mathcal{D}$: distribution of prompts
G: number of responses generated for a single prompt
π_{ref}: reference model (frozen)
π_θ: model with parameters θ
Loss:

$$\mathcal{L}(\phi) = -\mathbb{E}_{x\sim\mathcal{D},(y_i)_{i\in[1,G]}\sim\pi_{\text{ref}}(x)}\left[\frac{1}{G}\sum_{i=1}^{G}F(x,y_i)\right]$$

where

$$F(x,y) = \frac{1}{|y|}\sum_{t=1}^{|y|}\min\left(\frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}\cdot A_{\pi_\theta}((x,y_{<t}),y_t),\ g(\epsilon, A_{\pi_\theta}((x,y_{<t}),y_t))\right)$$

Option 4: Group Relative Policy Optimization (GRPO)

The advantage is not anymore computed using GAE, it is based on this simple computation:

x: prompt

$y_1, \ldots, y_G$: responses

$r_1, \ldots, r_G$: rewards according to the rule-based reward model

Advantage:

$$A((x, y_{i,<t}), y_{i,t}) = \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r}) + \epsilon}$$

Option 4: Group Relative Policy Optimization (GRPO)

Another (minor) difference with PPO:

In PPO, the KL-penalty term for not deviating was added to the reward

In GRPO, the reward is unchanged and the KL-penalty term is added to the loss using a different estimate:

$$D_{\mathrm{KL}}(\pi_\theta(y \mid x) \,||\, \pi_{\mathrm{ref}}(y \mid x)) = \sum_{t=1}^{|y|} \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\mathrm{ref}}(y_t \mid x, y_{<t})} - \log \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\mathrm{ref}}(y_t \mid x, y_{<t})} - 1$$

Option 4: Group Relative Policy Optimization (GRPO)

GRPO is “just” a simpler way of estimating the advantage, it is not specific to the “clipped surrogate objective” (TRPO / PPO style), it can also be adapted to the original objective (more like “vanilla policy gradient”). Recall the formula:

π_θ: policy with parameters θ

$\rho = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$: trajectory

Gradient:

$$\nabla_\theta \mathcal{L} = -\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} A_{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right]$$

Option 3: Direct preference optimization (DPO)

Interesting exercise: understand how the loss is derived, see https://arxiv.org/abs/2305.18290

$\mathcal{D}$: distribution of triples (prompt, better response, worse response)
π_{ref}: reference model (frozen)
π_θ: model with parameters θ
σ: logistic function $\sigma(x) = \frac{1}{1+\exp(-x)}$
Loss:

$$\mathcal{L}(\phi) = -\mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}}\left[\log\sigma\left(\beta\frac{1}{|y^+|}\sum_{t=1}^{|y^+|}\log\frac{\pi_\theta(y_t^+ \mid x, y_{<t}^+)}{\pi_{\text{ref}}(y_t^+ \mid x, y_{<t}^+)} - \beta\frac{1}{|y^-|}\sum_{t=1}^{|y^-|}\log\frac{\pi_\theta(y_t^- \mid x, y_{<t}^-)}{\pi_{\text{ref}}(y_t^- \mid x, y_{<t}^-)}\right)\right]$$