Notes on Large Language Models

Gaspard Beaudouin

January 28, 2025

Contents

1 Introduction

This document serves as a **collection of notes** summarizing what I have learned about LLMs. Please note that this is a **draft**, with mistakes and several sections remain incomplete. It explores the fundamentals and various methods to enhance Large Language Models (LLMs) using post-training techniques.

2 Pre Training of Language Models

2.1 Tokenization

This subsection describes how raw text is split into tokens using Byte Pair Encoding (BPE). BPE merges frequent character sequences to progressively create subword units, balancing the trade-off between word-based and character-based tokenization.

Algorithm 1 Byte Pair Encoding (BPE) Pseudocode

```
1: Input: Vocabulary size $V$, training corpus $C$
2: Initialize: Each character in $C$ is a token
3: while number of merge operations performed $< V$ do
4:     Count all pairs of consecutive tokens in $C$
5:     Identify the pair $(x, y)$ with the highest frequency
6:     Merge $(x, y)$ into a new token $z$
7:     Replace all occurrences of $(x, y)$ in $C$ with $z$
8:     Record this merge operation
9: end while
10: Output: Merged tokens as the final BPE vocabulary
```

Andrej Karpathy made a very clean clean implementation of it on github.

2.2 Positional embedding

Positional embeddings allow the model to incorporate the order of tokens into its representation. One common approach is the sinusoidal positional embedding proposed in Vaswani et al., *Attention Is All You Need*.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

where:

- $pos \in \{0, 1, 2, \dots\}$ is the position index in the sequence.
- $i \in \left\{0, 1, \dots, \frac{d_{\text{model}}}{2} - 1\right\}$ is the index for the sine/cosine pairs.
- d_{model} is the total dimensionality of the embeddings.

2.3 Transformers

During your internship, the primary focus will be on Large Language Models (LLMs). The classical architecture used for these models is the Transformer (Encoder-Decoder), first introduced in Vaswani et al., *Attention Is All You Need* and later adapted in Devlin et al., *BERT* for BERT. The attention mechanism is at the heart of this architecture.

The paper Vaswani et al., *Attention Is All You Need* revolutionized Deep Learning by introducing the self-attention mechanism.

Given an input $X \in \mathbb{R}^{\text{batch} \times \text{block size} \times \text{n\_embed}}$, we project the queries (Q), keys (K), and values (V) using weight matrices $W_Q, W_K, W_V \in \mathbb{R}^{\text{n\_embed} \times \text{n\_embed}}$:

$$\mathbf{Q} = XW_Q, \quad \mathbf{K} = XW_K, \quad \mathbf{V} = XW_V$$

Each projection has the same dimensions:

$$Q, K, V \in \mathbb{R}^{\text{batch} \times \text{block size} \times \text{n\_embed}}$$

The attention scores are computed as:

$$\textbf{scores} = \frac{QK^T}{\sqrt{d_k}} + M,$$

where $d_k = \frac{\text{n\_embed}}{\text{n\_heads}}$ and M is a mask (for causal or padded tokens).

Applying softmax yields attention weights A:

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)$$

The attention output is then:

$$\textbf{Attention}(Q, K, V) = A\,V$$

In multi-head attention, each head (indexed by i) is computed separately:

$$\text{head}_i = \text{Attention}(QW_Q^i,\, KW_K^i,\, VW_V^i),$$

where W_Q^i, W_K^i, W_V^i are the projection matrices for head i. These heads are concatenated and projected:

$$\text{MultiHead}(Q, K, V) = \text{Concat}\left(\text{head}_1, \ldots, \text{head}_h\right) W_O,$$

with $W_O \in \mathbb{R}^{\text{n\_embed} \times \text{n\_embed}}$.

Recent variants of Transformers and attention mechanisms (e.g., Ye et al., *Differential Transformer*) build on this original formulation.

2.4 GPT Architecture

The GPT architecture is a Decoder-only Transformer model. Attempts were made using encoder decoder, like T-5, but decoder only architecture turned out to be more effective. Its input is a sequence of tokens (obtained from a tokenizer). Let T be the length of the tokenized sequence. The goal is to predict the $(T+1)$-th token based on the first T.

- **Embedding:** Tokens are mapped to vectors via an embedding table of size vocab_size × n_embed. Positional embeddings are then added to these vectors.
- **Transformer Blocks:** The embedded sequence passes through N Transformer blocks. Each block has:
  - A *masked* multi-head self-attention layer,
  - A layer normalization + residual connection,
  - A feed-forward layer (often a two-layer MLP) + another residual connection.
- **Output:** After the final block, a linear layer projects each token embedding to a distribution over the vocabulary. Softmax yields the probability of each possible next token.

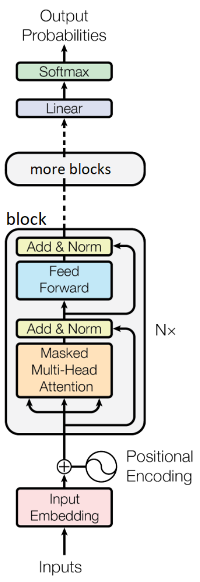


Figure 1: Decoder architecture for a Transformer-based Language Model.

2.5 Training

At each time step t, the model predicts the next token given the preceding tokens. Formally, let y_i be the correct token at position i. The cross-entropy loss is:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{V} \not\vdash_{[j=y_i]} \log(p_{ij}),$$

where:

- N is the total number of tokens in the dataset (or batch),
- V is the vocabulary size,
- p_{ij} is the predicted probability of token j at position i.

An alternative view, commonly seen in language modeling, is:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p_{\mathrm{LM}}(x_i \mid x_{<i}; \theta).$$

2.6 Generation

To generate text, GPT typically uses an autoregressive decoding strategy:

1. Feed a (prompt) sequence of tokens into the model.
2. Use the final embedding (which has information from all previous tokens, thanks to self-attention) to predict the next token.
3. Append the predicted token to the sequence and feed the sequence back into the model.
4. Repeat until the desired sequence length is reached (or until a special end-of-sequence token is generated).

Sampling strategies (greedy, beam search, nucleus sampling, etc.) can be used to pick the next token.

2.7 Training dataset

LLMs are typically trained on massive, diverse text corpora. These might include internet-scraped text (e.g., Common Crawl), Wikipedia articles, books, code repositories, and more. Multilingual training corpora are also common to produce models that can handle multiple languages.

2.8 Optimization

ADAM (Adaptive Moment Estimation) is an optimizer that combines the benefits of momentum optimization and adaptive learning rates (like RMSProp). It computes two moments: the first is the mean of the gradient (similar to momentum), and the second is the mean of the squared gradients, which adjusts the learning rate.

The ADAM equations are:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla L(w_t) \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(w_t))^2$$

The terms m_t and v_t are biased, so they are corrected by:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The weight update is then:

$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where: - m_t is the first moment estimate (mean of gradients), - v_t is the second moment estimate (mean of squared gradients), - η is the learning rate, - β_1 and β_2 are coefficients for the moments, - ϵ is a small term to prevent division by zero.

Explanation:

- Optimizer with momentum: It accelerates convergence by accumulating "velocity" based on past gradients.
- ADAM: It combines momentum accumulation with dynamically adjusted learning rates, making the optimization process more efficient and stable for neural networks. **Less dependence with lr than SGD.**

Batch size

Important to determine the correct size of the batch. We can use gradient accumulation to have bigger batch size artificially. It reduces the time of communication between host and devise (only one gradient descent by making an average of each mini batch). We make one epoch faster with bigger batch size, but if it is too big, the convergence is too slow (the average gradient norm is small).

Hyperparameter optimization

For HPO, there are different options. For example, search grid, random search (allow a bigger variance for the parameters). It is parallelised. Also we can used gaussian processes for smarter optimiation but sequential. There are other techniques such as population based training search.

2.9 GPU Distribution

Dataloader Optimization

In deep learning, optimizing data loading can significantly improve training efficiency. The DataLoader in frameworks like PyTorch is responsible for loading and feeding data to the model. However, naive data loading can create a bottleneck during training, especially with large datasets.

Techniques for Dataloader Optimization:

- Batch Loading: Instead of loading one data point at a time, load mini-batches of data. This reduces overhead from repeatedly loading individual samples.
- Pre-fetching: Pre-fetching allows loading the next batch of data while the current batch is being processed by the model, thereby hiding data loading time.
- Parallel Data Loading: Using multiple workers (threads) for data loading can significantly speed up the process. If each worker loads a subset of the data, they can feed the data to the model in parallel:

$$\text{DataLoader}(dataset, batch\_size, num\_workers)$$

- Asynchronous Data Loading: By overlapping data loading and model training using asynchronous I/O, the time spent loading data can be hidden by the forward and backward computations. `num_workers`:

  Using more CPU processes to load the batches, we can prepare more data ahead of time and send it to the GPU as soon as it is available.

  `pin_memory=True`:

  To be transferred to the GPU, the data is first moved from the pageable memory to the pinned memory on the CPU.

GPU Memory

During the training, the GPU needs to memorize many things (model parameters, gradients, outputs of each layers (activations) for backpropagation, momentum for optimizers). The GPUs needs a lot of ram memory, else: "CUDA out of memory". Gradient accumulation is a possibility, and reduce batch size.

TENSOR CORES AND MIXED PRECISON to tackle this issue

Tensor cores are specialized for matrix computing (while CUDA cores are specialized for vector computing). Each Tensor core is capable of processing 64 operations in 1 clock time.

Mixed precision (on gradients) : we can use FP16 (half precsion on 16 bits : sign + 5 bits for exponent, 8 for mantise) to make compute, instead of float 32. But if the gradient is too small: consdiered as zero, hence leading to instabilities. The gradient is distrubuted on quite low values: we need a scaler for backpropagation. We can used GradScaler() in PyTorch during training.

Gradient Checkpointing

Gradient checkpointing is a technique used to reduce memory usage during the backpropagation phase, which is especially useful in very deep neural networks.

Normally, during the forward pass, the intermediate activations from each layer are stored in memory to compute the gradients during backpropagation. However, this can be very memory-intensive for deep models. Gradient checkpointing alleviates this by saving memory at the cost of some additional computation.

Instead of storing all activations, only a subset of them are saved. The intermediate activations are recomputed from the saved ones during the backward pass. This technique reduces memory usage from $O(n)$ to $O(\sqrt{n})$, where n is the number of layers.

Mathematically, given a deep network with L layers, the idea is to store only k layers, where k is much smaller than L, and recompute the remaining layers' activations during backpropagation.

Distribution

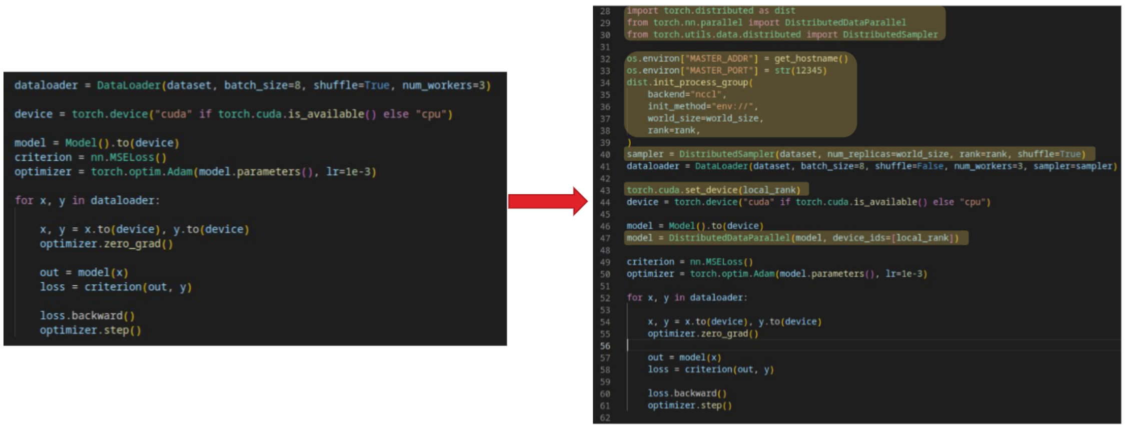
Data parallelism Parallelism (one example: multiprocessing): multiple instances of the same program are executed on separated computing devices, with separated memory spaces. Giving them different data or code allows the system to speedup the computation, provided that a separation can be done and the same result can be obtained. This generally implies communications between the processes (nvidia nccl library). Hence, data parallelism makes the compute faster and require less memory.

We train on batches because a GPU can use Streaming Multiprocessors. But we can also use multiples GPUs.

Distributed Data Parallelism: we need to synchronize the GPUs so they train the same model together, by making an "Allreduce average" of the gradients before the backpropagation. A significant portion of the memory allocation is dedicated to activations. Distributed data parallelism helps mitigating that issue.

`world_size` is the number of GPUs on which we want to distribute batches. `rank` are the their ids. The `DistributedSampler(dataset; num_replicas=world_size, rank=rank, shuffle=True)` handles the distribution of the dataset on the different processes (1 process on 1 GPU). `torch.cuda.set_device(local_rank)` says to torch we now run multiple GPUs.

`model=DistributedDataParallel(model, device_ids=[local_rank]` to says PyTorch we run the model on multiple GPUS with multipe processes, and Pytorch will make the Allreduce Average of the gradients. This wraps the model in DDP, enabling PyTorch to replicate the model across all devices and synchronize the gradients after each backward pass.



```python
dataloader = DataLoader(dataset, batch_size=8, shuffle=True, num_workers=3)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = Model().to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

for x, y in dataloader:

    x, y = x.to(device), y.to(device)
    optimizer.zero_grad()

    out = model(x)
    loss = criterion(out, y)

    loss.backward()
    optimizer.step()
```

```python
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel
from torch.utils.data.distributed import DistributedSampler

os.environ["MASTER_ADDR"] = get_hostname()
os.environ["MASTER_PORT"] = str(12345)
dist.init_process_group(
    backend="nccl",
    init_method="env://",
    world_size=world_size,
    rank=rank,
)
sampler = DistributedSampler(dataset, num_replicas=world_size, rank=rank, shuffle=True)
dataloader = DataLoader(dataset, batch_size=8, shuffle=False, num_workers=3, sampler=sampler)

torch.cuda.set_device(local_rank)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = Model().to(device)
model = DistributedDataParallel(model, device_ids=[local_rank])

criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

for x, y in dataloader:

    x, y = x.to(device), y.to(device)
    optimizer.zero_grad()

    out = model(x)
    loss = criterion(out, y)

    loss.backward()
    optimizer.step()
```

Figure 2: Distributed Data Parallelism

For the execution, (torch run or on a Slurm cluster srun/ sbatch) before python train.py. We can acces variables such as `world_size`, `rank`, `local_rank` , `master_adrr`, `master_port` with Slurm.

ZeRO - Zero Redundancy Optimizer : pooling of memory (parameters, optimizer states, gradients) DeepSpeed's optimization ZeRO reduces the memory footprint of the model without increasing communications volume (just a little bit).

Model/Pipeline parallelism

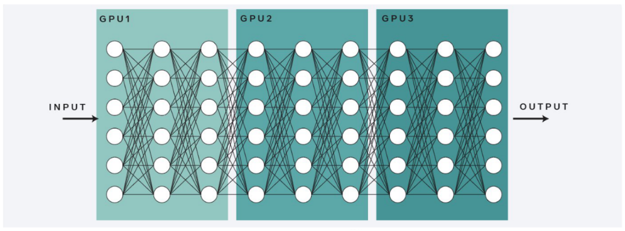


Figure 3: Model Parallelism

Tensor parallelism

Orthogonal to data parallelism (splitting a tensor along columns or rows) allows using multiple GPU simultaneously (no bubble) and shards activations, gradients, and optimizers across those devices. However it requires regular synchronization points such as AllReduce or AllGather.

Distributed Training refers to the general concept of distributing the training process across multiple devices or nodes. It can involve different parallelization strategies, such as data parallelism, model parallelism, or hybrid parallelism..

3 Supervised Fine Tuning

3.1 General idea

It is very similar to the pre training, the same loss is used, but the dataset is more specific, in order to have a useful model. Generally, an SFT dataset consists of users' questions and corresponding responses from the model.

There is a need to have a good chat template, delimiting the user query from the model response with <special tokens>.

We also do use a DataCollator designed for QA, that takes as input a batch of samples from the dataset. These samples typically include tokenized data: Each sample might have keys such as inputids, labels, and other optional fields.

We keep the same loss as before:

$$\mathcal{L}_{\text{sft}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p_{\text{LM}}(o_i | q, o_{<i}; \theta)$$

3.2 The dataset

3.3 Lora Fine tuning

E. J. Hu et al., *LoRA: Low-Rank Adaptation of Large Language Models*: LoRA (Low-Rank Adaptation) fine-tuning is a method designed to adapt large pre-trained language models (LLMs) by injecting low-rank transformations into specific layers. This approach makes fine-tuning more memory-efficient by introducing fewer trainable parameters, particularly useful for resource-constrained setups. Instead of updating all parameters, LoRA fine-tunes only additional, low-rank matrices, which effectively capture the task-specific knowledge with minimal overhead.

1. Pre-trained Layer Representation
   Assume we have a pre-trained model with a linear layer represented as:
   $$\mathbf{y} = \mathbf{W}\mathbf{x}$$
   where $\mathbf{W} \in \mathbb{R}^{d\times d}$ is the weight matrix of the layer, $\mathbf{x} \in \mathbb{R}^d$ is the input, and $\mathbf{y} \in \mathbb{R}^d$ is the output.

2. Low-Rank Decomposition
   Instead of fine-tuning $\mathbf{W}$ directly, LoRA introduces two trainable matrices, $\mathbf{A} \in \mathbb{R}^{d\times r}$ and $\mathbf{B} \in \mathbb{R}^{r\times d}$, where $r \ll d$, to approximate the necessary updates:
   $$\Delta\mathbf{W} = \mathbf{A}\mathbf{B}$$
   Here, $\Delta\mathbf{W}$ is the low-rank adaptation matrix that we add to $\mathbf{W}$ during fine-tuning.

3. LoRA Fine-tuning Forward Pass
   During the forward pass with LoRA, we update the original layer's output by adding this low-rank adaptation:
   $$\mathbf{y} = (\mathbf{W} + \alpha \cdot \mathbf{A}\mathbf{B})\mathbf{x}$$
   where α is a scaling factor to control the update's impact, often a hyperparameter set to adjust the magnitude of the low-rank addition.

4. Optimization
   In LoRA, only the parameters $\mathbf{A}$ and $\mathbf{B}$ are optimized, leaving $\mathbf{W}$ frozen. This reduces the number of trainable parameters from $d \times d$ to $r \times (d + d)$, which is significantly smaller due to $r \ll d$.

5. Gradient Update
   The gradient descent update for $\mathbf{A}$ and $\mathbf{B}$ would be:
   $$\mathbf{A} \leftarrow \mathbf{A} - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{A}}, \quad \mathbf{B} \leftarrow \mathbf{B} - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{B}}$$
   where η is the learning rate, and $\mathcal{L}$ is the loss function.

Using the chain rule: $\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \alpha \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial (\alpha \mathbf{A}\mathbf{B}\mathbf{x})} \cdot \frac{\partial (\alpha \mathbf{A}\mathbf{B}\mathbf{x})}{\partial \mathbf{A}}$

$$\mathbf{A} \leftarrow \mathbf{A} - \eta \cdot \alpha \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \mathbf{B}\mathbf{x}^T, \quad \mathbf{B} \leftarrow \mathbf{B} - \eta \cdot \alpha \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \mathbf{A}^T\mathbf{x}$$

By training only **A** and **B**, LoRA fine-tuning provides an efficient, low-rank adaptation that captures task-specific information without updating the large number of parameters in **W**.

4 Reinforcement Learning from Human feedback (response level)

4.1 The Dataset

To train the model, we generally need to have a dataset composed of different trajectories $\tau = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}$, and with rewards r_t associated to each action a_t.

For response level RLHF (**single step**), we generally have only **one reward for the full trajectory**, corresponding to r_T at the end of the response. Rewards are sparse.

This dataset must provide both the raw model outputs and the human preference signals:

- **Prompt-Response Pairs:** The dataset should contain prompts (or queries) posed to the language model and a set of candidate responses generated by the model for each prompt.
- **Human Comparisons:** For each prompt, humans compare two or more candidate responses and indicate which one is preferred. This yields pairwise preference data rather than absolute scores.
- **Aggregated Rewards:** Preference data is then converted into a reward function. For example, if response A is preferred to response B, then A might be assigned a higher reward score. Over many comparisons, one can infer a reward model that predicts the probability a response would be preferred by a human, or directly map preferences into scalar rewards.

Once we have this dataset, we can use it to train the model by first training the reward model, in order to do online Reinforcement Learning.

An other possibility is to use verifiable rewards Lambert et al., *Tulu 3: Pushing Frontiers in Open Language Model Post-Training* or even using execution feedbacks (unit tests) Shojaee et al., *Execution-based Code Generation using Deep Reinforcement Learning*, and so there is no need to train a reward model.

4.2 PPO

To align the LLM's responses, we can solicit user preferences on generated content and reward the model accordingly. Given a set of user queries and multiple candidate responses from the model, human annotators or users indicate which responses they find more helpful, safe, or aligned with given guidelines. These user preferences are transformed into scalar reward signals. By training the model's policy to produce responses that yield higher rewards, the model can be steered towards more desirable behaviors.

This general idea underlies methods such as Reinforcement Learning from Human Feedback (RLHF) and its variants. The goal is to optimize the model parameters to produce content that not only fits the training data distribution but also aligns with certain qualitative goals (e.g., helpfulness, correctness, and safety).

Here we formulate the trategy of LLM as a decision-making policy π_θ, parameterized by θ, which governs the selection of actions based on the current state. A trajectory $\tau = \{s_0, a_0, s_1, a_1, \ldots, s_T, a_T\}$ represents a sequence of states and actions over time.

The Markov Decision Process (MDP) is represented by the tuple $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$, with the following meanings:

- $\mathcal{S}$: The state space, where each state $s_t \in \mathcal{S}$ represents the current context at time t.
- $\mathcal{A}$: The action space at time t.
- $\mathcal{P}$: The state transition dynamics, $P(s_{t+1} \mid a_t, s_t)$ defines the probability of transitioning to a new state s_{t+1} given the current state s_t and the action a_t.
- $\mathcal{R}$: The reward function, which assigns rewards $r_t = r(s_t, a_t)$ based on the current state s_t and action a_t.
- γ : The discount factor, which determines how the model balances immediate rewards with longterm task-solving performance.

For response level RLHF, an action is a token, and a state is a sequence of token. I will continue with this example in this section.

In multi steps reasoning (math reasoning, coding, function calling), an action is a sequence of tokens, and a state as well (an action is a reasoning step). I will detail this in the next section.

Some training are **Online RL** because it involves real-time interaction with the environment to collect data and refine the policy, while **Offline RL** trains solely on a fixed dataset of past experiences without further interaction (DPO for example).

Most preference tuning algorithm are Online, with a reward model which is trained or designed.

The training can also be offline, where the generations and the rewards are generated before. (Yu et al., *StepTool: A Step-grained Reinforcement Learning Framework for Tool Learning in LLMs*).

Proximal Policy Optimization (PPO) Schulman, Wolski, et al., *Proximal Policy Optimization Algorithms* is a popular reinforcement learning algorithm that has been effectively applied to language model preference tuning, most notably in OpenAI's InstructGPT framework Ouyang et al., *Training language models to follow instructions with human feedback*.

To maximize the final task-solving performance, the model aims to optimize the expected reward $\overline{R_\theta}$, defined as:

$$J(\theta) = \overline{R_\theta} = \sum_\tau R(\tau)\pi_\theta(\tau) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)], \quad \text{where} \quad R(\tau) = \sum_{t=0}^{T-1} r_t$$

r_t will be defined later.

Policy Gradient Method:

The policy gradient method optimizes the policy π_θ by directly maximizing the expected reward $J(\theta)$. The gradient of $J(\theta)$ with respect to the policy parameters θ is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(\tau) R(\tau)\right].$$

These techniques directly optimize the policy of the agent—the mapping of states to actions—instead of learning a value function as in value-based methods. The central idea behind policy gradient methods is to improve the policy using the gradient ascent algorithm. In essence, these methods adjust the parameters of the policy in the This gradient is estimated using Monte Carlo sampling, and the policy parameters are updated via gradient ascent:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta),$$

where α is the learning rate, $J(\theta)$ represents the expected return when following policy π_θ and the gradient of policy performance $\nabla_\theta J(\theta)$ is called the policy gradient.

Here is the proof of the compute the gradient of the expected reward $R(\tau) = \sum_{t=0}^{T} r_t$.

$$\begin{aligned}
\nabla_\theta J(\theta) = \sum_\tau R(\tau)\nabla\pi_\theta(\tau) &= \sum_\tau R(\tau)\pi_\theta(\tau)\nabla \log \pi_\theta(\tau) \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[R(\tau)\nabla \log \pi_\theta(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau),(s_t,a_t)\sim\tau}\left[R(\tau)\sum_{t=1}^{T} \nabla \log \pi_\theta\left(a_t \mid s_t\right)\right]
\end{aligned}$$

A general form of policy gradient to **enhance learning efficiency and stabilize training**, is to replace $R(\tau)$ with ϕ_t. It can be formulated as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\Phi_t\right],$$

where Φ_t could be any of $\Phi_t = R(\tau)$ or $\Phi_t = \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$ or $\Phi_t = \sum_{t'=t}^{T} r(s_{t'}, a_{t'}) - b(s_t)$.

However, there can be a **high variance** with this method. This variance stems from the fact that different trajectories can result in diverse returns due to the stochasticity of the environment and the policy itself. **To reduce this variance, a common strategy is to use advantage function estimates in place of raw returns** in the policy gradient update rule.

The advantage function $A_t(a_t, s_t)$represents how much better it is to take a specific action at at state s_t, compared to the average quality of actions at that state under the same policy.

Mathematically, $A_t(a_t, s_t) = Q(s_t, a_t) - V(s_t)$, where $Q(s_t, a_t)$ is the action-value function, representing the expected return after taking action at at state s, and $V(s_t)$ is the value function, representing the average expected return at state s_t.

We now have :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A_t(a_t, s_t)\right], \tag{1}$$

Proximal Policy Optimization (PPO):
PPO is a more stable and efficient variant of the policy gradient method. It introduces a clipped objective function to prevent large updates to the policy parameters, ensuring stable training.

We will precise this later, but in a nutshell, the PPO objective is defined as:

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new and old policies, A_t is the advantage function, and ϵ is a small clipping parameter (e.g., $\epsilon = 0.2$). The advantage function A_t is computed as:

$$A_t = R(\tau) - V(s_t),$$

where $V(s_t)$ is the value function estimating the expected return from state s_t.

In response-level Reinforcement Learning from Human Feedback (RLHF), a full response is treated as a single action. There are two primary approaches for defining the reward r_t in this framework:

1. KL Penalty in the Loss Function:
Here, the KL divergence is incorporated only into the loss function during gradient descent, and not in the reward function. The loss function is given by:

$$\mathcal{L}(\theta) \simeq -\left(J(\theta) - \beta D_{KL}(\pi_{\theta_{\text{old}}}\|\pi_{\theta_{\text{new}}})\right),$$

where the reward r_t is defined as:

$$r_t = r(s_t, a_t) = \begin{cases} R_\psi(x; y) & \text{if } t = T-1, \text{ where } s_{t+1} = y \text{ is terminal,} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

So, when we compute the advantage, we have:

$$R(\tau) = R_\psi(x, y)$$

2. KL Penalty in the Reward Function:
Alternatively, the KL divergence can be directly incorporated into the reward function.

In this case, the intuition behind the loss is:

$$\mathcal{L}(\theta) \simeq -J(\theta),$$

In this case, the reward r_t is defined as:

$$r_t = r_{KL}(s_t, a_t) = \begin{cases} R_\psi(x; y) - \beta \log\left(\frac{\pi_{\theta_{\text{old}}}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\right) & \text{if } t = T-1, \text{ where } s_{t+1} = y \text{ is terminal,} \\ -\beta \log\left(\frac{\pi_{\theta_{\text{old}}}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\right) & \text{otherwise.} \end{cases} \tag{3}$$

This results in the cumulative reward for a trajectory τ being:

$$R(\tau) = R_\psi(x, y) - \beta \log\left(\frac{\pi_{\theta_{\text{old}}}(y|x)}{\pi_{\theta_{\text{ref}}}(y|x)}\right).$$

Note:
In this work, we primarily consider the KL penalty in the loss function. However, it is worth noting that the

KL penalty could also be incorporated into the reward function $r_{KL}(a_t, s_t)$ for computing the advantage A_t. (As did in Shao et al., *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models* with GRPo, and Yang et al., *Qwen2.5-Math Technical Report: Toward Mathematical Expert Model via Self-Improvement* in order to avoid complicated calculations of A_t .)

Using PPO, the idea is to solve this problem:

$$\max_\theta \mathbb{E}_{x\sim\mathcal{D}}\mathbb{E}_t\left[\sum_{t=0}^{T-1}\underbrace{r(s_t,a_t)-\beta\,\mathrm{KL}\left(\pi_\theta(\cdot|s_t),\pi_{\mathrm{ref}}(\cdot|s_t)\right)}_{r_{KL}(a_t,s_t)}\Big|s_0=s\right],$$

Here T is the random decision horizon (i.e., LLM outputs a full response after T reasoning steps and s_T is a terminate state that either contains the [EOS] token or is truncated due to the length limit)

where $\mathcal{D}$ is the distribution of training prompts x, a_t is the action (or the tokens generated) at time t, T is the decision horizon, $r(\cdot)$ is the ground truth reward function, π_θ is the parameterized generation policy of the LLM, π_{ref} is the initial reference policy, β is the KL coefficient. Notice that the definition on what is a single action can be very flexible, thus different definition of a action yields different algorithm designs and theoretical results.

Reward Model Training A reward model can be trained for Online RL. Here we take the example of single step and simple preference tuning.

The Reward Model R_ψ is trained using supervised learning. Depending on the type of reward data available, we can define the training objective as:

- For pairwise ranking:

$$\arg\min_{\psi\in\Psi}\sum_{(x,y^+,y^-)\in\mathcal{D}}\log\sigma\left(R_\psi(x,y^+)-R_\psi(x,y^-)\right),$$

  where (x, y^+, y^-) are triplets such that y^+ is preferred over y^-, and $\sigma(z)=\frac{1}{1+e^{-z}}$ is the sigmoid function.

- For continuous reward regression:

$$\arg\min_{\psi\in\Psi}\sum_{(x,y,r)\in\mathcal{D}}\left(R_\psi(x,y)-r\right)^2,$$

  where r is the scalar reward provided for the response y.

Value Model Training The Value Model V_ϕ is trained to predict the expected return (cumulative reward) for a given state. Its training objective is defined as:

$$\arg\min_{\phi\in\Phi}\sum_{x\in\mathcal{D}}\sum_{t=0}^{n}\left(V_\phi(s_t)-\sum_{l=0}^{T-t}\gamma^l r_t\right)^2,$$

Advantage Function Definition The Advantage function A_ϕ quantifies the benefit of taking a specific action compared to the baseline value of the current state. It is calculated using Generalized Advantage Estimation (GAE) Schulman, Moritz, et al., *High-Dimensional Continuous Control Using Generalized Advantage Estimation*:

$$\hat{A}(s_t,a_t)=\sum_{l=0}^{T-t}(\gamma\lambda)^l(r_{t+l}+\gamma V_\phi(s_{t+l+1})-V_\phi(s_{t+l})),$$

If we continue with the example of **response level RLHF**, then with $\gamma = 1$ and $r(s_t, a_t) = 0$ for $t \in \{0, ..., T-1\}$, and with $r(s_T, a_T) = R_\phi(y)$ we have

$$\hat{A}(s_t,a_t)=Q(s_t,a_t)-V(s_t)=\sum_{t'=t}^{T}r_{t'}-V_\phi(s_t)=r_t+\gamma V_\phi(s_{t+1})-V_\phi(s_t))$$

More detail on PPO The policy model π_θ is optimized using PPO to maximize the likelihood of actions with positive advantages and minimize those with negative advantages:

An action is a token for LM.

$$\mathcal{L}_{\text{policy gradient}}(\theta) = -\mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=1}^{T} A_\phi(s_t, a_t)\log\pi_\theta(a_t, s_t)]$$

The key idea behind PPO is to update the model's policy in a way that avoids overly large updates that could destabilize training. This is done by introducing a clipped objective function.

$\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new and old policies, A_t is the advantage function at time step t, and ϵ is a small hyperparameter (e.g., 0.1 or 0.2) that defines the clipping range. By using a clipped objective, PPO ensures that policy updates do not move too far from the old policy distribution, thereby providing stable and efficient training.

With

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

$$\mathcal{L}_{\text{PPO-penalty}}(\theta) = -\mathbb{E}_t\left[r_t(\theta)\hat{A}_t - \beta KL\left(\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right)\right] \tag{4}$$

$$\mathcal{L}_{\text{PPO-clip}}(\theta) = -\mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{5}$$

$$\mathcal{L}_{\text{PPO-combined}}(\theta) = -\hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right) - \beta KL\left(\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right)\right] \tag{6}$$

If we use $r_t = r(s_t, a_t)$ (no KL penalty in the reward), we will probably need to use a loss with an explicit penalty, such as $\mathcal{L}_{\text{PPO-penalty}}$ or$\mathcal{L}_{\text{PPO-combined}}$.

On the contrary, if we use a KL penalty incorporated in the reward $r_{KL}(s_t, a_t)$, we should use the loss $\mathcal{L}_{\text{PPO-clip}}$.

4.3 REINFORCE ++: A small variation of PPO

With J. Hu, *REINFORCE++: A simple and efficient appproach for aligning Large Language Models*, we keep the same method execpt that we don't train a actor critic model (value model), we just take:

$$A_t(s_t, a_t) = R_\psi(x, y) - \beta\sum_{i=t}^{T} log(\frac{\pi_{\theta_{old}}(a_i|s_i)}{\pi_{\theta_{ref}}(a_i|s_i)})$$

with as in (**??**) and (4),

$$r_t = r_{KL}(s_t, a_t) = \begin{cases} R_\psi(x; y) - \beta log(\frac{\pi_{\theta_{old}}(a_t|s_t)}{\pi_{\theta_{ref}}(a_t|s_t)}) & \text{if } t = T-1, \text{ where } s_{t+1} = y \text{ is terminal,} \\ -\beta log(\frac{\pi_{\theta_{old}}(a_t|s_t)}{\pi_{\theta_{ref}}(a_t|s_t)}) & \text{otherwise.} \end{cases}$$

$$\mathcal{L}_{\text{PPO-clip}}(\theta) = \hat{\mathbb{E}}_t \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t(s_t, a_t), \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t(s_t, a_t)\right)$$

4.4 REINFORCE

Here, we don't need to compute advantages. Given that in LLM applications, $R_\psi(x, y)$ is only obtained at the end of the full sequence, it may be more appropriate to model the entire generation as a single action, as opposed to each token. We then try to maximize the problem:

$$\mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(\cdot|x)}\left[(R(\tau))\nabla_\theta\log\pi_\theta(y|x)\right]$$

$$\mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(\cdot|x)}\left[(R_\psi(y, x) - b)\nabla_\theta\log\pi_\theta(y|x)\right]$$

4.5 DPO

Direct Preference Optimization (DPO) Rafailov et al., *Direct Preference Optimization: Your Language Model is Secretly a Reward Model* is another recent approach designed to train language models directly from human preferences without relying on complex reinforcement learning loops or separate reward models.

The core idea of DPO is to incorporate preference information directly into the training objective without explicitly constructing a reward model or running a traditional RL algorithm. DPO uses the principle that if response A is preferred over response B, then the model's log probability for A should be increased relative to B:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x,a,b)\sim D}\left[\log\sigma\Big(\log\pi_\theta(a|x) - \log\pi_\theta(b|x)\Big)\right],$$

where $\sigma(\cdot)$ is the logistic sigmoid function, and (x, a, b) are triplets drawn from the preference dataset D, indicating that for the same prompt x, the response a was preferred to b.

By directly comparing pairs of responses and adjusting the model parameters to reflect these preferences, DPO aims to simplify the alignment process. It avoids training a separate reward model or using a full RL loop. Instead, it leverages a preference-based pairwise loss that encourages the model to improve on preferred responses, pushing the probability mass towards responses that align with user preferences.

4.6 Comparaison

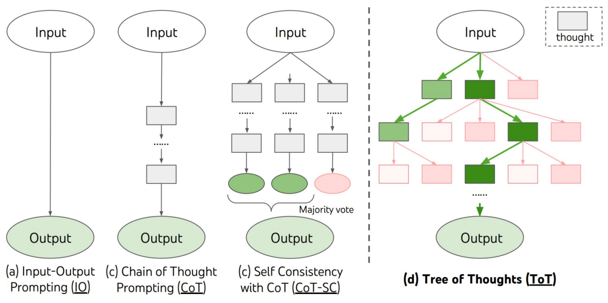
Compared with PPO, DPO is more efficient in terms of compute, speed, and engineering efforts. DPO does not need the extra stage of training a reward model, and during policy training it does not need to decode online responses (which is usually slow) or train an additional value model. Meanwhile, PPO trains on online data generated by the current policy, while DPO trains on static, pre-generated offline data. This may limit exploration in DPO and thus hurt the training quality. PPO and Reinforce are online RL algorithms.

5 Improving reasoning with reasoning steps

5.1 Chain Of Thoughts

The introduction of CoT prompting, introduced in Wei et al., *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models* has paved the way of a new paradigm. This class of methods boils down to asking the LLM nicely to reveal its internal thoughts (e.g. "Let's think step by step").

Another possibility is to use multiple LLM calls: a simple variation on CoT would be "CoT self-consistency", i.e. sample multiple CoT traces in parallel and use majority voting to find the "right" answer.


Input
Input
Input
Input
thought
Majority vote
Output
Output
Output
Output
(a) Input-Output Prompting (IO)
(c) Chain of Thought Prompting (CoT)
(c) Self Consistency with CoT (CoT-SC)
(d) Tree of Thoughts (ToT)

Further complexifies the above (in CS terms we go from linear list -> tree): build up an m-ary tree of intermediate thoughts (thoughts = intermediate steps of CoT); at each thought/node:

1. run the "propose next thoughts- next steps" prompt (or just sample completions m times)
2. evaluate those thoughts (either independently or jointly) (very good, don't know, bad) with voting
3. keep the top m

If the reasoning needs to have 3 steps (for example in the game 24 explained in the paper), the tree will have 3 levels. cons: very expensive and slow pro: works with off-the-shelf LLMs

But later, some training methods were based on CoT.

5.2 SFT on reasoning traces

It is a normal fine tuning, using standard SFT loss. The model will learn to make reasoning steps before giving the result. Special tokens likle `<think>` `<\think>` are often used to delimitate the thinking process.

The thing is that it can be difficult to have a good Q&A dataset with reasoning traces.

Use LLMs to generate a good (question, rationale, answer) dataset

STaR: Self-Taught Reasoner Zelikman et al., *STaR: Bootstrapping Reasoning With Reasoning* Ground Zero. Instead of always having to CoT prompt, bake that into the default model behavior. Given a dataset of (question, answer) pairs, manually curate a few CoT traces and use them as fewshot examples to generate (rationale, answer) tuples, given a question, for the rest of the dataset ("bootstrap reasoning"). Do SFT on (question, rationale, answer) triples. Do multiple iterations of this i.e. collect data retrain and keep sampling from that new policy (in practice I've observed people use 3 iterations). Con: can only learn from correct triples.

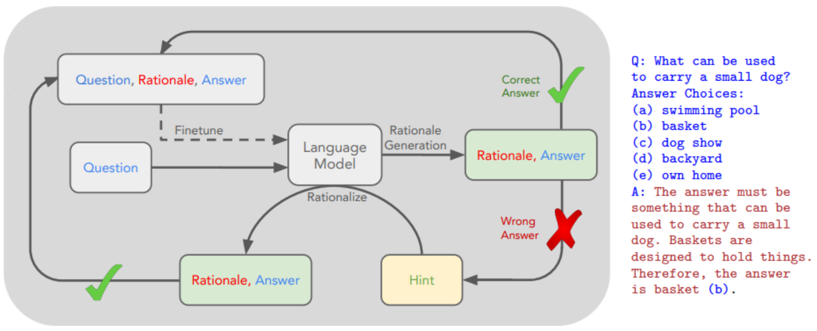

Question, Rationale, Answer
Finetune
Question
Language Model
Rationale Generation
Rationale, Answer
Correct Answer
Wrong Answer
Rationalize
Rationale, Answer
Hint
Q: What can be used to carry a small dog?
Answer Choices:
(a) swimming pool
(b) basket
(c) dog show
(d) backyard
(e) own home
A: The answer must be something that can be used to carry a small dog. Baskets are designed to hold things. Therefore, the answer is basket (b).

Figure 1: An overview of STaR and a STaR-generated rationale on CommonsenseQA. We indicate the fine-tuning outer loop with a dashed line. The questions and ground truth answers are expected to be present in the dataset, while the rationales are generated using STaR.

5.3 RL on reasoning steps

On simple extension to the previous training can be use DPO on the rejected rationales.

V-STaR: Hosseini et al., *V-STaR: Training Verifiers for Self-Taught Reasoners* Collect both correct/incorrect samples. Incorrect ones can be used to train a verifier LLM via DPO. During inference additionally use the verifier to rank the generations.

We can also apply exaclty the same PPO algorithm described before.

We now have two options:

- an action a_t is a token, and a state s_t is also a token.
- an action a_t is a sequence of tokens, and a state s_t is also a sequence of tokens.

In the second case, the difficulty is to get a reward for every reasoning step. It is easier to get a reward in the first case.

The idea is to minimize:

$$\mathcal{L}_{\text{simple}}(\theta) = -\mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=1}^{T} A_\phi(s_t, a_t) \log \pi_\theta(a_t, s_t)]$$

In pratice we will use the PPO loss, as described before.

A PPO considering an action as a sequence of tokens is detailed later with Function Calling and O1 Coder.

5.4 An example: O1 coder

Y. Zhang et al., *o1-Coder: an o1 Replication for Coding* is introduced as an attempt to replicate OpenAI's O1 model with a specific focus on coding tasks. It integrates RL and Monte Carlo Tree Search (MCTS) to enhance System-2 reasoning capabilities for code generation. Here, an action is a sequence of tokens.

Introduction

The O1 model employs RL to address the lack of reasoning data by combining pretraining with exploration strategies. The aim is to generate reasoning-enhanced code datasets, fine-tune models, and optimize reasoning pathways.

Framework Overview

The training process involves the following key steps:

1. Training a Test Case Generator (TCG) to create diverse and valid test cases.
2. Using MCTS to synthesize reasoning-enhanced datasets.
3. Fine-tuning the policy model on valid reasoning steps.
4. Iteratively improving the Process Reward Model (PRM) and the policy model using RL.
5. Generating new reasoning data to enhance self-play.

Key Formulas

1. Direct Preference Optimization (DPO) Objective:

$$L_{\text{DPO}}(\gamma_{\text{TCG}}; \gamma_{\text{TCG}}^{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim D_{\text{pref}}}\left[\log\sigma\left(\beta\log\frac{\gamma_{\text{TCG}}(y_w|x)}{\gamma_{\text{TCG}}^{\text{ref}}(y_w|x)} - \beta\log\frac{\gamma_{\text{TCG}}(y_l|x)}{\gamma_{\text{TCG}}^{\text{ref}}(y_l|x)}\right)\right]$$

where $\sigma(x)$ is the sigmoid function, β is a scaling factor, and D_{pref} represents the preference dataset.

2. Reward Value for Terminal Nodes:

$$v_m^i = \alpha \cdot \text{compile} + (1-\alpha)\cdot \text{pass}$$

where:

- α balances compilation success and test pass rate,
- compile is binary (1 for success, 0 for failure),
- $\text{pass} = \frac{\text{Num}_{\text{passed}}}{\text{Num}_{\text{test cases}}}$.

3. Supervised Fine-Tuning (SFT) Loss:

$$L_{\text{SFT}} = -\sum_{(Q_i, S_i^j, C_i')} \log\pi_\theta(S_i^{1:m}\circ C_i'|Q_i)$$

where $\circ$ denotes concatenation of reasoning steps $S_i^{1:m}$ and final code C_i'.

4. Point-wise Loss for Process Reward Model (PRM):

$$L_{\text{point-wise}}^{\text{PRM}} = -\mathbb{E}_{(Q_i, S_i^{1:j-1}, S_i^j, v_i^j)\sim D}\left[v_i^j\log r(Q_i, S_i^{1:j}) + (1-v_i^j)\log(1-r(Q_i, S_i^{1:j}))\right]$$

5. Pair-wise Loss for PRM:

$$L_{\text{pair-wise}}^{\text{PRM}} = -\mathbb{E}_{(Q_i, S_i^{1:j-1}, S_i^{\text{win}}, S_i^{\text{lose}})\sim D_{\text{pair}}}\left[\log\sigma\left(r(Q_i, S_i^{1:j-1}, S_i^{\text{win}}) - r(Q_i, S_i^{1:j-1}, S_i^{\text{lose}})\right)\right]$$

6. Aggregated Reward Function:

$$\phi(R_i, r_i^{1:m}) = \alpha(t) \cdot R_i + (1 - \alpha(t)) \cdot \frac{1}{m} \sum_{j=1}^{m} \gamma^j r_i^j$$

where:

- $\alpha(t)$ adjusts weights between outcome-based and intermediate rewards,
- γ is a discount factor for future rewards.

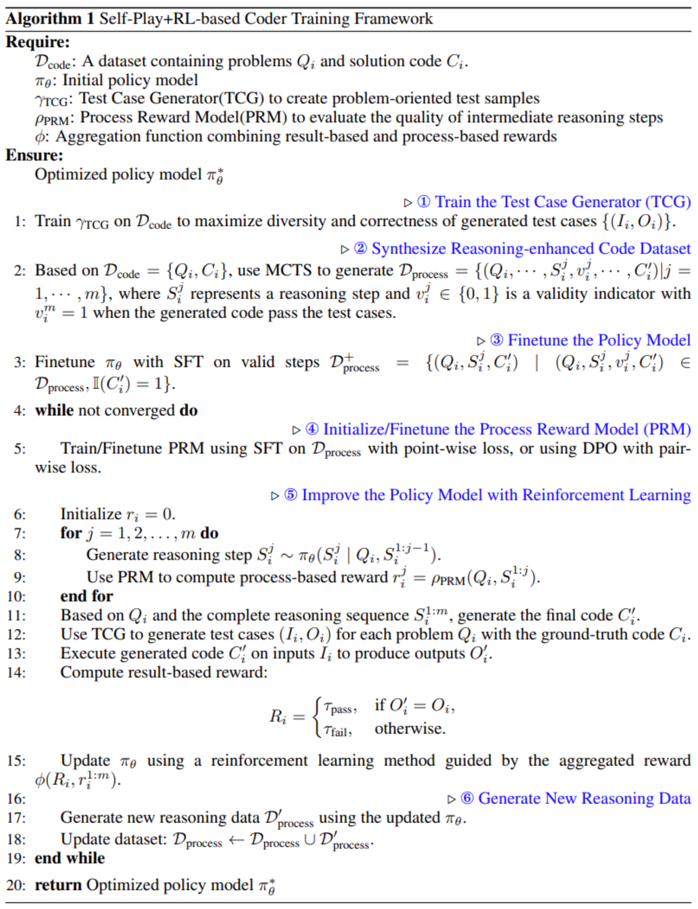
7. PPO We can then apply PPO with the reward $\phi(R_i, r_i^{1:m})$ for an answer.
Here the action space is a sequence of tokens.
Given the token-level representation of states s_t and actions a_t, the clipped surrogate objective for PPO is defined as:

$$L_{\text{PPO}}(\theta) = \mathbb{E}_t \left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$ is the probability ratio between the current and old policies,
- A_t is the advantage function,
- ϵ is a hyperparameter controlling the clipping range.

The token sequences are updated iteratively, ensuring that each reasoning step remains aligned with the overall policy improvement goal.



```
Algorithm 1 Self-Play+RL-based Coder Training Framework
Require:
    $\mathcal{D}_{\text{code}}$: A dataset containing problems $Q_i$ and solution code $C_i$.
    $\pi_\theta$: Initial policy model
    $\gamma_{\text{TCG}}$: Test Case Generator(TCG) to create problem-oriented test samples
    $\rho_{\text{PRM}}$: Process Reward Model(PRM) to evaluate the quality of intermediate reasoning steps
    $\phi$: Aggregation function combining result-based and process-based rewards
Ensure:
    Optimized policy model $\pi_\theta^*$
                                                  ▷ ① Train the Test Case Generator (TCG)
 1: Train $\gamma_{\text{TCG}}$ on $\mathcal{D}_{\text{code}}$ to maximize diversity and correctness of generated test cases $\{(I_i, O_i)\}$.
                                                  ▷ ② Synthesize Reasoning-enhanced Code Dataset
 2: Based on $\mathcal{D}_{\text{code}} = \{Q_i, C_i\}$, use MCTS to generate $\mathcal{D}_{\text{process}} = \{(Q_i, \cdots, S_i^j, v_i^j, \cdots, C_i')|j = 1, \cdots, m\}$, where $S_i^j$ represents a reasoning step and $v_i^j \in \{0, 1\}$ is a validity indicator with $v_i^m = 1$ when the generated code pass the test cases.
                                                  ▷ ③ Finetune the Policy Model
 3: Finetune $\pi_\theta$ with SFT on valid steps $\mathcal{D}_{\text{process}}^+ = \{(Q_i, S_i^j, C_i') \mid (Q_i, S_i^j, v_i^j, C_i') \in \mathcal{D}_{\text{process}}, \mathbb{I}(C_i') = 1\}$.
 4: while not converged do
                                                  ▷ ④ Initialize/Finetune the Process Reward Model (PRM)
 5:     Train/Finetune PRM using SFT on $\mathcal{D}_{\text{process}}$ with point-wise loss, or using DPO with pair-wise loss.
                                                  ▷ ⑤ Improve the Policy Model with Reinforcement Learning
 6:     Initialize $r_i = 0$.
 7:     for $j = 1, 2, \ldots, m$ do
 8:         Generate reasoning step $S_i^j \sim \pi_\theta(S_i^j \mid Q_i, S_i^{1:j-1})$.
 9:         Use PRM to compute process-based reward $r_i^j = \rho_{\text{PRM}}(Q_i, S_i^{1:j})$.
10:     end for
11:     Based on $Q_i$ and the complete reasoning sequence $S_i^{1:m}$, generate the final code $C_i'$.
12:     Use TCG to generate test cases $(I_i, O_i)$ for each problem $Q_i$ with the ground-truth code $C_i$.
13:     Execute generated code $C_i'$ on inputs $I_i$ to produce outputs $O_i'$.
14:     Compute result-based reward:
            $R_i = \begin{cases} \tau_{\text{pass}}, & \text{if } O_i' = O_i, \\ \tau_{\text{fail}}, & \text{otherwise.} \end{cases}$
15:     Update $\pi_\theta$ using a reinforcement learning method guided by the aggregated reward $\phi(R_i, r_i^{1:m})$.
16:                                               ▷ ⑥ Generate New Reasoning Data
17:     Generate new reasoning data $\mathcal{D}_{\text{process}}'$ using the updated $\pi_\theta$.
18:     Update dataset: $\mathcal{D}_{\text{process}} \leftarrow \mathcal{D}_{\text{process}} \cup \mathcal{D}_{\text{process}}'$.
19: end while
20: return Optimized policy model $\pi_\theta^*$
```

Figure 4: O1 coder algorithm

6 AI Agents and Function calling

6.1 Introduction

What is an AI agent ?

It is an autonomous system designed to perform tasks, with a certain degree of independence, interacting with users or systems through natural language leveraging external tools, accessing real-time information, and adapting over time to improve its performance.
Function calling can be considered a sub-branch of AI agents. It enables interaction with external code such as APIs or custom functions. It can an be **single-turn** (user, assistant) or **multi-turn** (user, assistant, user, assistant . . .) Reasoning involves multi-turn AI agent, because the agent responds to questions or completes tasks through a series of steps

6.2 Evaluation

6.2.1 Function calling benchmarks (single turn)

We evaluate those models mainly on a benchmark called Berkely Function Calling Leaderboard.

Patil et al., "Gorilla: Large Language Model Connected with Massive APIs"

This is clearly the main benchmark for this task. For example, the new models from Apple Intelligence are evaluated on it. This benchmarks consists of multiple questions, and giving different tools the model should call. There are two metrics. First, the AST evaluation first constructs the abstract syntax tree of the function call, then extracts the arguments and checks if they match the ground truth's possible answers. The other one is to execute the functions calls and to check the output.

The Berkeley Function-Calling Leaderboard (BFCL) evaluates LLMs using two main categories: Abstract Syntax Tree (AST) Evaluation and Executable Function Evaluation. The AST evaluation focuses on the syntactic accuracy of the generated function calls, ensuring that the model's output matches a predefined function documentation in structure and parameters. This includes checks for correct function names, required parameters, and appropriate data types. The Executable Function Evaluation goes a step further by running the generated function calls to verify their operational correctness. This executable test ensures that the functions not only compile but also execute correctly, providing the expected results, which is crucial for practical applications where real-time performance is essential.

Wu et al., *Seal-Tools*

6.2.2 AI agents benchmarks (multi turn)

Lu et al., *ToolSandbox*

This benchmark is made for more complex scenarios. To answer a query, the agent send calls to an Execution environment. The Milestones represent predefined key events that need to happen in this trajectory. The metrics used is score ≈ a similarity between the messages bus and the Milestones. And if a Minefield happens (which an event that should not occur), the score becomes 0. This benchmark show that open source and proprietary models have a significant performance gap.

A typical test case starts with the User speaking to the Agent. From then on, the role being addressed gets to speak next, until the end state is reached. Upon receiving a User request, an Agent can decide to respond to the User asking for more information, or inform the Execution Environment to execute a Tool, providing intended tool name and arguments. The Execution Environment executes the Tool in an code. InteractiveConsole, (Foundation, 2024), which depending on the Tool modifies the world state stored in the Execution Context, and responds to the Agent. Once the User decides the task has been completed, it informs the Execution Environment to execute the endconversation tool, which puts the system in the end state, ready to be evaluated based on the dialog's similarity to Milestones and Minefields.

The User role is implemented with an LLM (GPT4o).

Toolssandbox contains 1032 test cases meticulously crafted by 2 internal domain experts to capture challenging tool-use scenarios, with human authored and carefully calibrated Milestones and Minefields to support evaluation.

Yan et al., "BFCL V3: Multi-Turn Multi-Step Function Calling Evaluation"

Gorilla released a new version for their benchmark, in order to evaluate multi-turn and multi-step calls.

In contrast to multi-turn (user t0, assistant t1, user t2, assistant t3, ..), multi-step is where the LLM can break the response down into multiple steps (user t0, assistant t1, assistant t2,..). This new paradigm mimics real-world agentic behaviors where AI assistants might have to plan execution paths, request and extract critical information, and handle sequential function invokes to complete a task.

Metrics

Instead of only checking if each individual function output is correct, we compare the attributes of the system's state after every turn against the expected state. If the model successfully brings the system to the correct state at the end of each turn, it passes the evaluation.

In earlier versions of BFCL (V1 and V2), response-based evaluation was used, where models were assessed based on their immediate function responses, such as return values or Abstract Syntax Tree (AST) structures. However, this approach has limitations:

- Inconsistent Trajectories: In multi-turn tasks, models may take different, valid paths that deviate from the expected ones, making it hard to predict or enforce a specific trajectory (e.g., listing files with ls before creating a directory with mkdir).
- Error Recovery: When models encounter errors and take recovery actions, response-based evaluation may unfairly penalize them, as it expects strict trajectory or response equivalence.
- Redundant Actions: The model might perform extra, unnecessary actions that are reasonable in context, but response-based evaluation penalizes these redundancies even if they don't affect the task outcome.

6.3 SFT Training

Here are some datasets used.

Liu et al., *APIGen*

They present an automated data generation pipeline. They collected 3,673 real world APIs, and generated 60,000 high-quality entries.

To do so, they:

1. Sample some real world REST APIs documentations (mainly from ToolBench)
2. They filter and remove APIs with bad documentation (lacking parameters ...)
3. They generate Q&A pairs with LLMs with a variery of query style (simple, parallel ...)
4. Filtering: the process removes low-quality data points due to formatting issues, execution errors, or failure to pass the semantic check (to be sure the generated data aligns with the query's objective).
5. Data points passing all stages are added back to the seed dataset as high-quality examples to enhance future generation diversity

They made experiments by fine tuning DeepSeek-Coder-1.3B-instruct and DeepSeek-Coder-7B-instruct-v1.5 using the xLAM (AgentOhana) training pipeline. They evaluate on BFCL. they conducted an ablation study by adding the datasets that were filtered out by stage 3 (semantic checker) and stage 2 (execution checker) back to the training set, simulating situations where generated data is used without the rigorous verification process. The performance comparison on the BFCL benchmark, shown in Fig. 5, reveals that using these filtered datasets for training harms the final performance which demonstrates the effectiveness of the APIGen framework in filtering out low-quality data.

J. Zhang, Lan, Murthy, et al., *AgentOhana*

Data Standardization and Unification for LAA: input of step i, Action: output of step i, Observation: next observation of step i, . . . They also have an agent (Mistral) that rates the quality of of the trajectory, not only the final output.

J. Zhang, Lan, Zhu, et al., *xLAM*

Another notable model excelling in Function Calling is presented in Lin et al., *Hammer: Robust Function-Calling for On-Device Language Models via Function Masking*. The authors performed a LoRA-based supervised fine-tuning (SFT) of Qwen 2.5 on an augmented version of the APIGen dataset. Specifically, they constructed an "irrelevance dataset" derived from APIGen by removing the tool used in the answer from the list of available tools and modifying the output to be empty. This enabled the model to learn when no function call is required. Additionally, they introduced a "masking" technique for the functions by altering the names of the available tools and their parameters. This approach encourages the model to focus more on the descriptions of the tools rather than relying solely on their names.

6.4 RL Training: DPO for single turn from execution feedback

Qiao et al., "Making Language Models Better Tool Learners with Execution Feedback" The idea here is to use the results of the executions of the tools to get a reward.

Training is carried out in two stages.

The first one is a normal SFT on a dataset composed of $\mathcal{D}_{tool} = (question, output)$.

where

$$\text{output} = \begin{cases} \text{answer} & use\_tool = \text{false} \\ \text{tool call} & use\_tool = \text{true} \end{cases}$$

This dataset is obtained by using a normal QA dataset (question, answer), and if GPT fails to answer the question, we generate function calls with GPT with the tools (WikiSearch for example).

$$\mathcal{L}_{\text{clone}}(\theta) = \sum_{(s,q,t,a)\in D_{\text{tool}}} - \log p_{\text{LM}}(output|s, q; \theta)$$

The second phase used execution feedbacks.

For each question q, we have k different candidate responses y_i $(1 \leq i \leq k)$ marshaled from other LLMs (e.g. ChatGPT, LLaMA) or human experts. We rank them with that relation of order: TrueYes > TrueNo > FalseYes > FalseNo. (First Yes if good answer, and second Yes if good tool call).

So we then score each y_i with the LLM we train (parametrized by θ):

$$p_i(\theta) = \frac{\sum_t \log p_{\text{LM}}(y_{i,m} \mid q, y_{i,<m}; \theta_{\text{clone}})}{\|y_i\|}$$

where m denotes the m-th token of y_i, p_i is the conditional log probability of y_i and $\|y_i\|$ refers to the length-normalized factor.

$$\mathcal{L}_{\text{rank}}(\theta) = \sum_{r_i<r_j} \max(0, p_i(\theta) - p_j(\theta))$$

We also keep using the normal loss to not deviate from the original model.

$$\mathcal{L}_{\text{sft}}(\theta) = -\sum_m \log p_{\text{LM}}(o_m|s, q, o_{<m}; \theta_{\text{clone}})$$

Eventually, the loss used is :

$$\mathcal{L}_{\text{RLEF}} = \alpha \cdot \mathcal{L}_{\text{rank}} + \mathcal{L}_{\text{sft}}$$

They evaluate on Math datset, and QA dataset:

They simply check for the last number predicted by the model for the math reasoning task and check whether the correct answer is within the first twenty words for other tasks.

6.5 RL Training: PPO with Multiple Steps

This section is an extract from Yu et al., *StepTool: A Step-grained Reinforcement Learning Framework for Tool Learning in LLMs*.

Problem Set Up

We frame the tool selection strategy for LLMs as a Markov Decision Process (MDP) $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$:

- $\mathcal{S}$: State space, where $s_t \in \mathcal{S}$ represents the context or environment responses at time t.
- $\mathcal{A}$: Action space, with $a_t \in \mathcal{A}$ representing tool calls or terminal responses.
- $\mathcal{P}$: Transition dynamics, $P(s_{t+1} \mid a_t, s_t)$, describing state evolution.
- R: Reward function, assigning $r_t = R(s_t, a_t)$ based on the current state-action pair.

- γ: Discount factor, balancing immediate and long-term rewards.

The policy π_θ selects actions based on states. A trajectory $\tau = \{s_1, a_1, \dots, s_T, a_T\}$ reflects LLM interactions with tools over time. The goal is to maximize the expected reward $\overline{R_\theta} = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$. Using policy gradient, the gradient of $\overline{R_\theta}$ is computed as:

$$\begin{aligned}\nabla \overline{R_\theta} &= \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau) \nabla \log \pi_\theta(\tau)\right] \\ &= \mathbb{E}_{\tau, (s_t, a_t) \sim \tau}\left[R(\tau) \sum_{t=1}^{T} \nabla \log \pi_\theta(a_t \mid s_t)\right].\end{aligned}$$

Replacing $R(\tau)$ with the advantage function $\hat{A}(s_t, a_t)$ improves learning efficiency:

$$\hat{A}(s_t, a_t) = G_t^n - V(s_t) = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t} r_T - V(s_t),$$

where G_t^n is the estimated future reward, and $V(s_t)$ is the value function. The proposed StepTool framework uses step-grained reward shaping and optimization for multi-step tasks.

Step-grained Reward Shaping

Step-grained rewards include intermediate rewards for tool interactions and final task-solving rewards:

$$\hat{r}_t = \begin{cases} \alpha \cdot \hat{r}_t^{\mathrm{SC}} + \hat{r}_t^{\mathrm{Con}}, & t < T \\ \hat{r}_t^{\mathrm{IS}}, & t = T \end{cases}$$

where α is a scaling factor. These rewards can be derived using automated models, annotations, or advanced tools like GPT-4.

Step-grained Optimization

Extending to token-level granularity, the gradient becomes:

$$\nabla \overline{R_\theta} = \mathbb{E}_{\tau, (s_t, a_t) \sim \tau}\left[\sum_{t=1}^{T} \hat{A}(s_t, a_t) \sum_{i=1}^{L_t} \nabla \log \pi_\theta(a_t^i \mid s_t, a_t^{1:i-1})\right],$$

where $\hat{A}(s_t, a_t)$ uses step-grained rewards $\hat{\boldsymbol{r}}_t$. The objective function is:

$$L_\theta(\pi) = \mathbb{E}_{\tau, (s_t, a_t) \sim \tau}\left[\sum_{t=1}^{T} \hat{A}(s_t, a_t) \sum_{i=1}^{L_t} \log \pi_\theta(a_t^i \mid s_t, a_t^{1:i-1})\right].$$

This approach extends RLHF Ouyang et al., *Training language models to follow instructions with human feedback*, which optimizes single-step tasks, to multi-step scenarios.

A Practical Instantiation with PPO

StepTool employs Proximal Policy Optimization (PPO) with Generalized Advantage Estimation (GAE) for stability:

$$\begin{aligned}\hat{A}(s_t, a_t) &= \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots, \\ \delta_t &= \hat{r}_t + \gamma V(s_{t+1}) - V(s_t).\end{aligned}$$

The PPO-clip objective introduces clipping to limit policy updates:

$$\mathcal{L}_\theta^{\mathrm{PPO}}(\pi) = \hat{\mathbb{E}}_{\tau, (s_t, a_t) \sim \tau}\left[\min\left(r_t \hat{A}(s_t, a_t), \mathrm{clip}(r_t, 1-\epsilon, 1+\epsilon)\hat{A}(s_t, a_t)\right)\right],$$

where r_t is the policy ratio, and ϵ controls deviation from the old policy. A KL divergence penalty further stabilizes training, ensuring robust multi-step tool learning.

7 Other techniques

7.1 Mixture of Experts

This architecture is now widely, used, as in Jiang et al., *Mixtral of Experts*.

The **Mixture of Experts (MoE)** architecture combines the predictions of multiple expert networks, each specialized in a different region of the input space. A gating network determines the contribution of each expert to the final output.

- **Expert Networks**: A set of N expert networks $\{E_i(x)\}_{i=1}^N$, each producing an output for the input x.
- **Gating Network**: A gating network $G(x)$ that outputs a probability distribution over the experts.

1. **Gating Network Output**:
$$G(x) = \text{softmax}(W_g x + b_g)$$
where W_g and b_g are the weights and biases of the gating network.

2. **Expert Network Output**: Each expert $E_i(x)$ produces an output:
$$E_i(x) = f_i(W_i x + b_i)$$
where W_i and b_i are the weights and biases of the i-th expert.

3. **Final Output**: The final output y is a weighted sum of the expert outputs:
$$y = \sum_{i=1}^{N} G_i(x) \cdot E_i(x)$$
where $G_i(x)$ is the gating weight for the i-th expert.

The MoE model is trained using gradient-based optimization, minimizing a loss function with respect to the parameters of both the gating network and the expert networks.

7.2 Distillation

7.3 Quantization

7.4 Model Merging

7.5 KV Cache

7.6 Flash Attention

7.7 Differential transformers

Ye et al., *Differential Transformer*

They propose a novel attention mechanism, inspired by noise-canceling headphones, which amplifies attention to the relevant context while canceling noise. They obtain incredible results for long context tasks, such as information retrieval, much better than normal Transformer (their eval is made with Multi-needle retrieval results in 64k length). They trained a 3B model but they didn't released the weights, we cannot be sure this architecture will be game changing, it will require more testing.

Given an input sequence $x = (x_1, ..., x_N) \in \mathbb{R}^N$ we embed it into $X = (\mathbf{x_1}, ..., \mathbf{x_N}) \in \mathbb{R}^{N \times d_{model}}$

The critical design is that we use a pair of softmax functions to cancel the noise of attention scores.

Then the differential attention operator DiffAttn$(\cdot)$ computes outputs via:

$$[Q_1; Q_2] = XW^Q, \quad [K_1; K_2] = XW^K, \quad V = XW^V \tag{7}$$

$$\text{DiffAttn}(X) = \left(\text{softmax}\left(\frac{Q_1 K_1^T}{\sqrt{d}}\right) - \lambda\, \text{softmax}\left(\frac{Q_2 K_2^T}{\sqrt{d}}\right)\right) V \tag{8}$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times 2d}$ are parameters, and λ is a learnable scalar. In order to synchronize the learning dynamics, we re-parameterize the scalar λ as:

$$\lambda = \exp(\lambda_1 + \lambda_2) - \exp(\lambda_2 + \lambda_3) + \lambda_{\text{init}} \tag{9}$$

where $\lambda_1, \lambda_2, \lambda_3 \in \mathbb{R}^d$ are learnable vectors, and $\lambda_{\text{init}} \in (0, 1)$ is a constant used for the initialization of λ. We empirically find that the setting $\lambda_{\text{init}} = 0.8 - 0.6 \times \exp(-0.3 \cdot (l - 1))$ works well in practice, where $l \in [1, L]$ represents the layer index. It is used as the default strategy in our experiments.

We also explore using the same λ_{init} (e.g., 0.8) for all layers as another initialization strategy. As shown in the ablation studies (Section 3.8), the performance is relatively robust to different initialization strategies.

Differential attention takes the difference between two softmax attention functions to eliminate attention noise. The idea is analogous to differential amplifiers proposed in electrical engineering, where the difference between two signals is used as output, so that we can null out the common-mode noise of the input. In addition, the design of noise-canceling headphones is based on a similar idea. We can directly reuse FlashAttention as described in Appendix A, which significantly improves model efficiency.

Multi-Head Differential Attention We also use the multi-head mechanism in Differential Transformer. Let h denote the number of attention heads. We use different projection matrices W_i^Q, W_i^K, W_i^V, $i \in [1, h]$ for the heads. The scalar λ is shared between heads within the same layer. Then the head outputs are normalized and projected to the final results as follows:

$$\text{head}_i = \text{DiffAttn}(X; W_i^Q, W_i^K, W_i^V, \lambda) \tag{10}$$

$$\overline{\text{head}_i} = (1 - \lambda_{\text{init}}) \cdot \text{LN}(\text{head}_i) \tag{11}$$

$$\text{MultiHead}(X) = \text{Concat}(\overline{\text{head}_1}, \ldots, \overline{\text{head}_h}) W^O \tag{12}$$

where λ_{init} is the constant scalar in Equation (2), $W^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ is a learnable projection matrix, and $\text{LN}(\,\cdot\,)$ uses RMSNorm for each head.

The overall architecture stacks L layers, where each layer contains a multi-head differential attention module, and a feed-forward network module. We describe the Differential Transformer layer as:

$$Y^l = \text{MultiHead}(\text{LN}(X^l)) + X^l \tag{4}$$

$$X^{l+1} = \text{SwiGLU}(\text{LN}(Y^l)) + Y^l \tag{5}$$

where $\text{LN}(\cdot)$ is RMSNorm **46**, $\text{SwiGLU}(X) = (\text{swish}(XW^G) \odot XW_1)W_2$, and $W^G, W_1 \in \mathbb{R}^{d_{\text{model}} \times \frac{8}{3} d_{\text{model}}}$, $W_2 \in \mathbb{R}^{\frac{8}{3} d_{\text{model}} \times d_{\text{model}}}$ are learnable matrices.

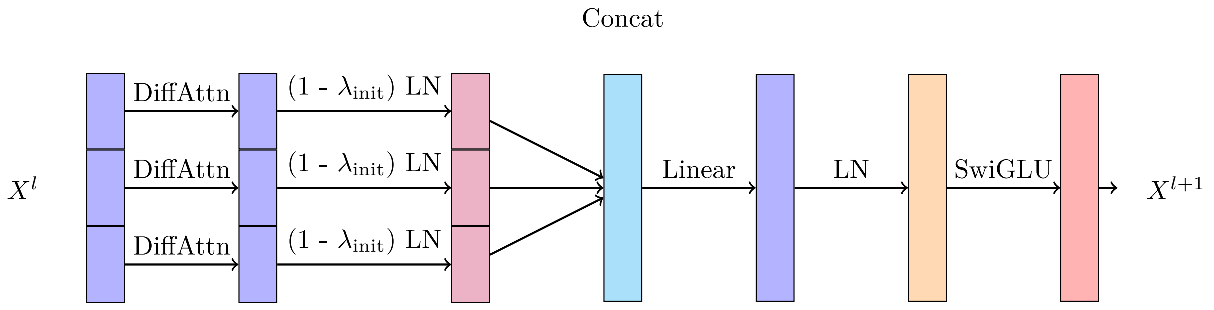

Concat
X^l
DiffAttn
DiffAttn
DiffAttn
(1 - λ_{init}) LN
(1 - λ_{init}) LN
(1 - λ_{init}) LN
Linear
LN
SwiGLU
X^{l+1}

Figure 5: Overall architecture

8 Inference Time scaling

A new paradigm has emerged this the intoruction of GPT o1. Tailored for reasoning problems, the idea is to find the best reasoning path using tree algorithms.

8.1 Tree of Thoughts

8.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a sequential decision-making algorithm widely used in tasks requiring exploration and exploitation of decision spaces. In the context of the O1-CODER framework, MCTS is employed to improve reasoning in coding tasks by guiding the generation of intermediate reasoning steps, refining them, and iteratively improving the policy model.

MCTS operates by constructing a search tree incrementally, exploring potential decision paths, and refining them based on simulated outcomes. It consists of the following steps:

Selection Navigate the tree from the root to a leaf node using a selection policy such as Upper Confidence Bounds for Trees (UCT):

$$UCT = Q(s,a) + C \cdot \sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{13}$$

where:

- $Q(s,a)$: Average reward of taking action a from state s,
- $N(s)$: Number of visits to state s,
- $N(s,a)$: Number of times action a has been taken from s,
- C: Exploration parameter.

Expansion If the selected leaf node is not terminal, add a new node to the tree.

Simulation Perform a rollout by randomly or heuristically sampling actions to simulate task completion and obtain a reward.

Backpropagation Update the values of nodes along the path with the result of the simulation.

References

Devlin, Jacob et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* en. arXiv:1810.04805 [cs]. May 2019. URL: http://arxiv.org/abs/1810.04805 (visited on 09/12/2024).

Hosseini, Arian et al. *V-STaR: Training Verifiers for Self-Taught Reasoners.* 2024. arXiv: 2402.06457 [cs.LG]. URL: https://arxiv.org/abs/2402.06457.

Hu, Edward J. et al. *LoRA: Low-Rank Adaptation of Large Language Models.* 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

Hu, Jian. *REINFORCE++: A simple and efficient appproach for aligning Large Language Models.* Dec. 2024. DOI: 10.13140/RG.2.2.15090.62409.

Jiang, Albert Q. et al. *Mixtral of Experts.* 2024. arXiv: 2401.04088 [cs.LG]. URL: https://arxiv.org/abs/2401.04088.

Lambert, Nathan et al. *Tulu 3: Pushing Frontiers in Open Language Model Post-Training.* 2024. arXiv: 2411.15124 [cs.CL]. URL: https://arxiv.org/abs/2411.15124.

Lin, Qiqiang et al. *Hammer: Robust Function-Calling for On-Device Language Models via Function Masking.* 2024. arXiv: 2410.04587 [cs.LG]. URL: https://arxiv.org/abs/2410.04587.

Liu, Zuxin et al. *APIGen: Automated Pipeline for Generating Verifiable and Diverse Function-Calling Datasets.* en. arXiv:2406.18518 [cs]. June 2024. URL: http://arxiv.org/abs/2406.18518 (visited on 09/12/2024).

Lu, Jiarui et al. *ToolSandbox: A Stateful, Conversational, Interactive Evaluation Benchmark for LLM Tool Use Capabilities.* en. arXiv:2408.04682 [cs]. Aug. 2024. URL: http://arxiv.org/abs/2408.04682 (visited on 09/12/2024).

Ouyang, Long et al. *Training language models to follow instructions with human feedback.* 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155.

Patil, Shishir G. et al. "Gorilla: Large Language Model Connected with Massive APIs". In: *arXiv preprint arXiv:2305.15334* (2023).

Qiao, Shuofei et al. "Making Language Models Better Tool Learners with Execution Feedback". en. In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers).* Mexico City, Mexico: Association for Computational Linguistics, 2024, pp. 3550–3568. DOI: 10.18653/v1/2024.naacl-long.195. URL: https://aclanthology.org/2024.naacl-long.195 (visited on 09/12/2024).

Rafailov, Rafael et al. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model.* 2024. arXiv: 2305.18290 [cs.LG]. URL: https://arxiv.org/abs/2305.18290.

Schulman, John, Philipp Moritz, et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation.* 2018. arXiv: 1506.02438 [cs.LG]. URL: https://arxiv.org/abs/1506.02438.

Schulman, John, Filip Wolski, et al. *Proximal Policy Optimization Algorithms.* 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

Shao, Zhihong et al. *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models.* 2024. arXiv: 2402.03300 [cs.CL]. URL: https://arxiv.org/abs/2402.03300.

Shojaee, Parshin et al. *Execution-based Code Generation using Deep Reinforcement Learning.* 2023. arXiv: 2301.13816 [cs.LG]. URL: https://arxiv.org/abs/2301.13816.

Vaswani, Ashish et al. *Attention Is All You Need.* en. arXiv:1706.03762 [cs]. Aug. 2023. URL: http://arxiv.org/abs/1706.03762 (visited on 09/12/2024).

Wei, Jason et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.* 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

Wu, Mengsong et al. *Seal-Tools: Self-Instruct Tool Learning Dataset for Agent Tuning and Detailed Benchmark.* May 14, 2024. arXiv: 2405.08355[cs]. URL: http://arxiv.org/abs/2405.08355 (visited on 09/12/2024).

Yan, Fanjia et al. "BFCL V3: Multi-Turn Multi-Step Function Calling Evaluation". In: 2024.

Yang, An et al. *Qwen2.5-Math Technical Report: Toward Mathematical Expert Model via Self-Improvement.* 2024. arXiv: 2409.12122 [cs.CL]. URL: https://arxiv.org/abs/2409.12122.

Ye, Tianzhu et al. *Differential Transformer.* 2024. arXiv: 2410.05258 [cs.CL]. URL: https://arxiv.org/abs/2410.05258.

Yu, Yuanqing et al. *StepTool: A Step-grained Reinforcement Learning Framework for Tool Learning in LLMs.* 2024. arXiv: 2410.07745 [cs.CL]. URL: https://arxiv.org/abs/2410.07745.

Zelikman, Eric et al. *STaR: Bootstrapping Reasoning With Reasoning.* 2022. arXiv: 2203.14465 [cs.LG]. URL: https://arxiv.org/abs/2203.14465.

Zhang, Jianguo, Tian Lan, Rithesh Murthy, et al. *AgentOhana: Design Unified Data and Training Pipeline for Effective Agent Learning*. en. arXiv:2402.15506 [cs]. Mar. 2024. URL: http://arxiv.org/abs/2402.15506 (visited on 09/12/2024).

Zhang, Jianguo, Tian Lan, Ming Zhu, et al. *xLAM: A Family of Large Action Models to Empower AI Agent Systems*. en. Sept. 2024. URL: https://arxiv.org/abs/2409.03215v1 (visited on 09/12/2024).

Zhang, Yuxiang et al. *o1-Coder: an o1 Replication for Coding*. 2024. arXiv: 2412.00154 [cs.SE]. URL: https://arxiv.org/abs/2412.00154.