RLHF and PPO

Umar Jamil

Downloaded from: https://github.com/hkproj/rlhf-ppo
License: Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0):
https://creativecommons.org/licenses/by-nc/4.0/legalcode

Not for commercial use

Topics

- Short intro to language models
- AI alignment
- Reinforcement Learning
  - The RL setup
  - Connection between RL and language models
  - Reward model
  - Trajectories
  - Policy Gradient Optimization
  - Reducing Variance
  - Advantage estimation
  - Importance sampling
  - Off-Policy Learning
- PPO
  - Loss function
  - Reward hacking
- Code walkthrough

Prerequisites

- Basics of probability and statistics.
- Basics of deep learning (gradient descent, loss functions, etc.)
- Basic knowledge of RL (agent, state, environment, reward, etc.)
- Understanding of the Transformer model and language models

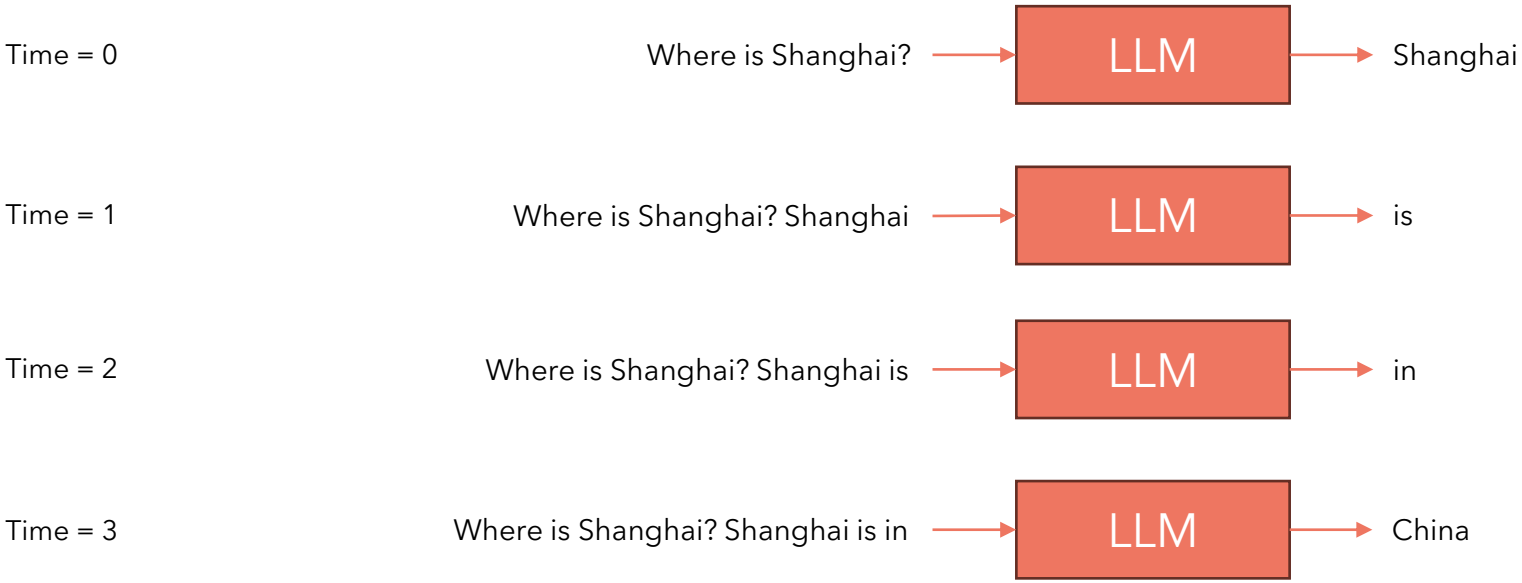
Short intro to language models

A language model is a probabilistic model that assign probabilities to sequence of words.
In practice, a language model allows us to compute the following:

A language model outputs a list of probabilities, one for every token in the vocabulary, indicating how likely a token is the next one.

Iteratively generating the next token

To generate a complete answer to a prompt, a language model is iteratively queried by adding the previously chosen token to the input.



AI alignment

A large language model typically is pretrained on a massive amount of data, for example the entire Wikipedia and billions of web pages. This gives the language model a vast “knowledge” of information to complete any prompt in a reasonable way. However, to use an LLM as a chat assistant (for example ChatGPT) we want to force the language model to follow a particular style. For example, we may want the following:

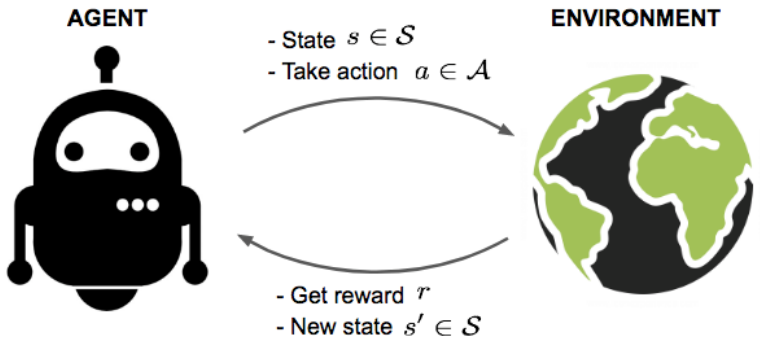
- Do not use offensive language
- Do not use racist expressions
- Answer questions using a particular style

The goal of AI alignment is to align the model’s behavior with a desired behavior.

Umar Jamil – https://github.com/hkproj/rlhf-ppo

Introduction to Reinforcement Learning

Reinforcement Learning is concerned with how an intelligent agent should take actions in an environment to maximize the cumulative reward.



Let me give you a concrete example with our channel's mascot.

The RL setup

Agent: the cat

State: the position of the cat (x, y) in the grid

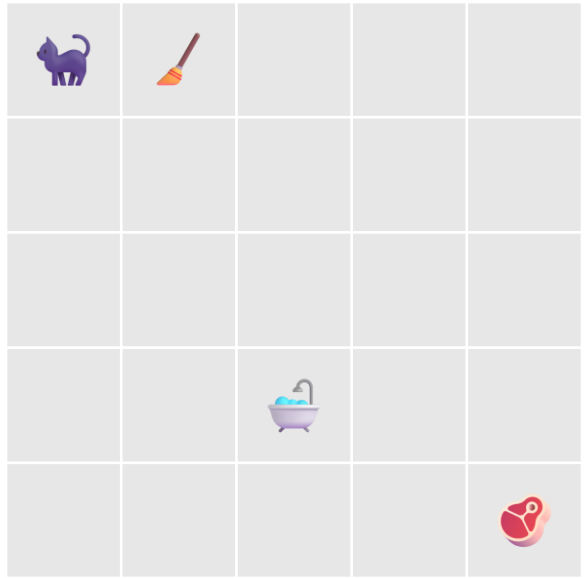
Action: at each position, the cat can move to one of the 4-directionally connected cells. If a move is invalid, the cell will not move and remain in the same position. Every time the cat makes a move, it results in a new state and a reward.

Reward model:

- A move to another empty cell results in a reward of 0.
- A move towards the broom, will result in a reward of -1.
- A move towards the bathtub will result in a reward of -10 and the cat fainting (episode over). The cat will be respawned at the initial position again.
- A move towards the meat will result in a reward of +100

Policy: a policy rules how the agent selects the action to perform given the state it is in: $a_t \sim \pi(\cdot \mid s_t)$

The goal in RL is to select a policy that maximizes the expected return when the agent acts according to it.



The RL setup: connection to language models

Agent: the language model itself

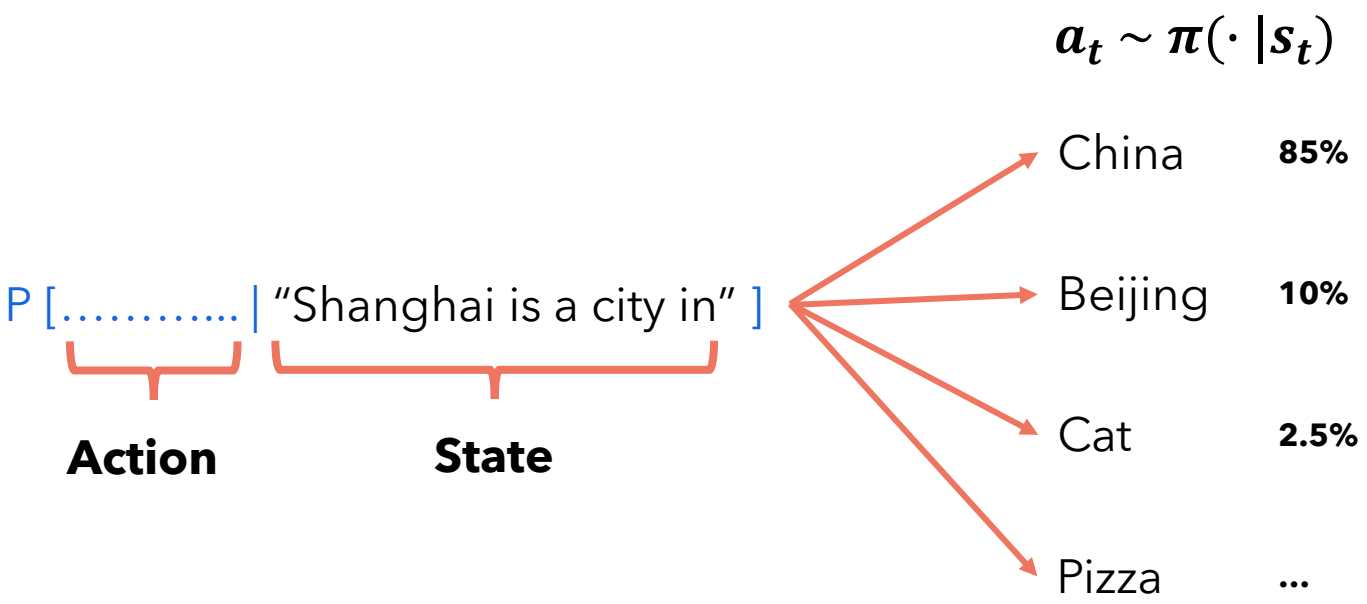
State: the prompt (input tokens)

Action: which token is selected as the next token

Reward model: the language model should be rewarded for generating "good responses" and should not receive any reward for generating "bad responses".

Policy: In the case of language models, the policy is the language model itself! Because it models the probability of the action space given the current state of the agent: $a_t \sim \pi(\cdot|s_t)$

Let's look at how we can define the reward model for our language model



Umar Jamil – https://github.com/hkproj/rlhf-ppo

Reward model for language models

It is not easy to create a reward model for language models, because this would require us to create a dataset of prompts and responses and assign a universally accepted “reward” for each answer.

Question (Prompt)	Answer (Text generated by the language model)	Reward (0.0 ~ 1.0)
Where is Shanghai?	Shanghai is a city in China	???
Explain gravity like I’m 5	Gravity is what pulls things toward each other. It's why you stay on the ground and planets orbit the sun.	???
What is 2+2?	4	???

People are so good good at finding a common ground for agreement, but unfortunately, they’re good at comparing.

Reward model by comparison

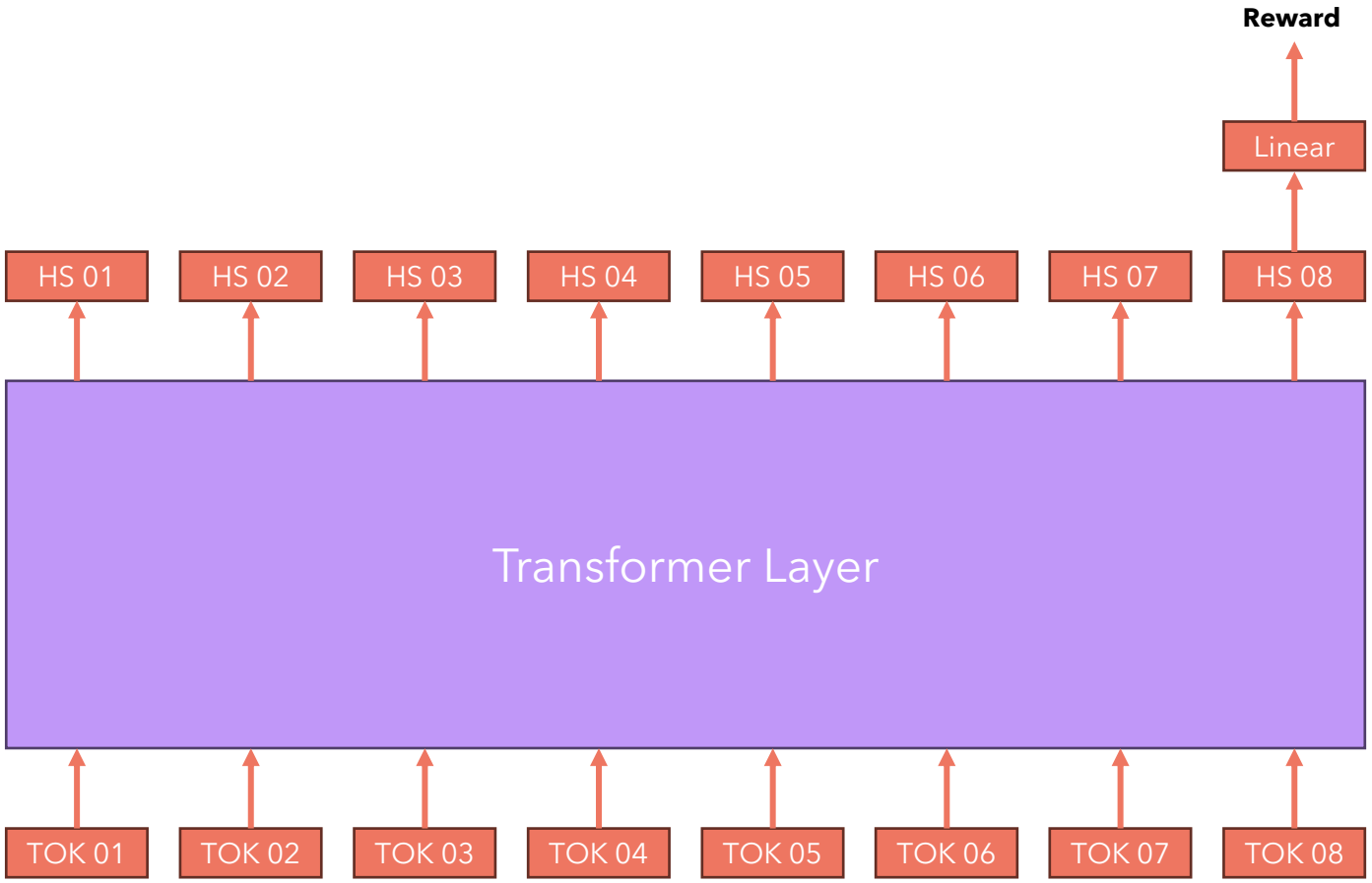
Imagine if we could instead create a dataset of queries and answers and we could ask people to just select which one they prefer. This would be much easier!

Question (Prompt)	Answer 1	Answer 2	Chosen
Where is Shanghai?	Shanghai is a city in China	Shanghai does not exist	1
Explain gravity like I'm 5	Gravity is a famous restaurant	Gravity is what pulls things toward each other. It's why you stay on the ground and planets orbit the sun.	2
What is 2+2?	4	2+2 is a very complicated math problem…	1

Using a dataset like this, we can train a model to assign a score to a given answer. **Let's see how it's done**.

Reward model architecture

When we give a series to tokens as input to a language model, which is commonly a Transformer model, it generates a list of hidden states, each corresponding to one input token, which is an embedding that "captures" the information of all the tokens that come before it. The hidden states are then converted into logits through a Linear layer and then into probabilities by using the softmax function. To generate the reward for a response, we can just use the hidden state of the last token of the response, send it to a Linear layer (with only one output feature) and use it as the value of the reward associated with the input.



Linear layer (one output feature)

Hidden states

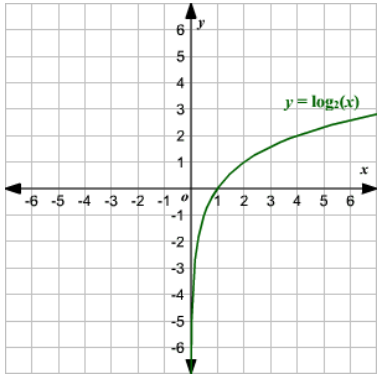
Attention mechanism

Input tokens: question + answer

Umar Jamil – https://github.com/hkproj/rlhf-ppo

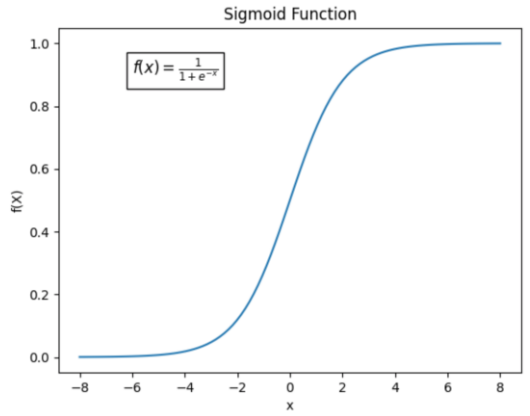
Reward model loss

To optimize our language model's behavior by using RL, we need a score model that gives a numeric value for each response generated by the language model. Now that we have a dataset that defines which answer we like based on a query (prompt), we can build a neural network that gives us a numeric score for each response.



Response → Reward Model → Reward (numeric)

$$Loss = -\log\ \sigma\big(r(x, y_w) - r(x, y_l)\big)$$



We have two cases:

- $r(x, y_w) > r(x, y_l)$ → Sigmoid will return a value > 0.5 → Loss will be a small negative number
  - The loss will be small when the order is correct
- $r(x, y_w) < r(x, y_l)$ → Sigmoid will return a value < 0.5 → Loss will be a very large negative number
  - The loss will be very large when the order is wrong

This loss forces the model to give high rewards to "winning" responses and low rewards to "losing" responses, because that's the only way for the model to minimize the loss.

Reward model: practical implementation

In HuggingFace, we can train a custom reward model by using a ***RewardTrainer*** and an ***AutoModelForSequenceClassification***, which is an language model with a special linear layer on top. We just ask the language model to output the hidden state corresponding to the last token, send it to a linear layer that calculates the reward and then train the language model according to the loss we described in our previous slide.

```python
def compute_loss(
    self,
    model: Union[PreTrainedModel, nn.Module],
    inputs: Dict[str, Union[torch.Tensor, Any]],
    return_outputs=False,
) -> Union[torch.Tensor, Tuple[torch.Tensor, Dict[str, torch.Tensor]]]:
    if not self.use_reward_data_collator:
        warnings.warn(
            "The current compute_loss is implemented for RewardDataCollatorWithPadding,"
            " if you are using a custom data collator make sure you know what you are doing or"
            " implement your own compute_loss method."
        )
    rewards_chosen = model(
        input_ids=inputs["input_ids_chosen"],
        attention_mask=inputs["attention_mask_chosen"],
        return_dict=True,
    )["logits"]
    rewards_rejected = model(
        input_ids=inputs["input_ids_rejected"],
        attention_mask=inputs["attention_mask_rejected"],
        return_dict=True,
    )["logits"]
    # calculate loss, optionally modulate with margin
    if "margin" in inputs:
        loss = -nn.functional.logsigmoid(rewards_chosen - rewards_rejected - inputs["margin"]).mean()
    else:
        loss = -nn.functional.logsigmoid(rewards_chosen - rewards_rejected).mean()

    if return_outputs:
        return loss, {
            "rewards_chosen": rewards_chosen,
            "rewards_rejected": rewards_rejected,
        }
    return loss
```

Reward (single floating-point number) given to the chosen response

Reward (single floating-point number) given to the rejected response

Calculate the loss

$$Loss = -\log \sigma\left(r(x, y_w) - r(x, y_l)\right)$$

Umar Jamil – https://github.com/hkproj/rlhf-ppo

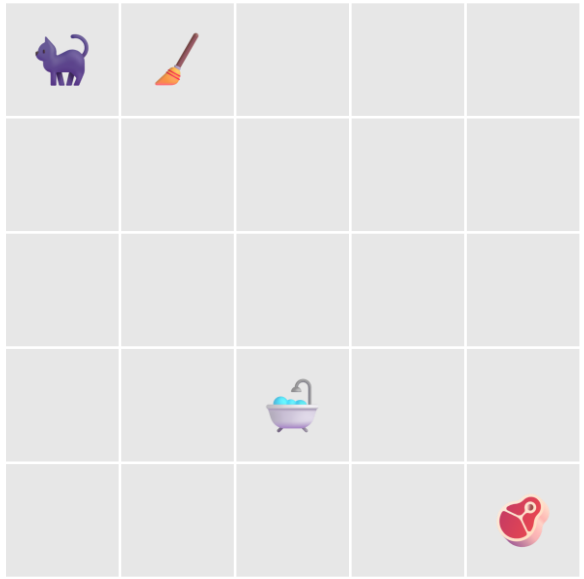
Let's talk about trajectories…

As said previously, the goal in RL is to select a policy which maximizes the expected return when the agent acts according to it. More formally:

$$\pi^* = \arg\max_{\pi} J(\pi),$$

The expected return of a policy is the expected return over all possible **trajectories**.

$$J(\pi) = \int_{\tau} P(\tau|\pi) R(\tau) = \mathop{\mathrm{E}}_{\tau\sim\pi}\left[R(\tau)\right].$$



A trajectory is a series of (action, state), starting from an initial state

$$\tau = (s_0, a_0, s_1, a_1, ...).$$

We will model the next state as being stochastic (suppose that the cat is drunk and doesn't always succeed in moving correctly)

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

We can thus define the probability of a trajectory as follows:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

We will always work with discounted rewards (we prefer immediate rewards instead of future):

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

Umar Jamil – https://github.com/hkproj/rlhf-ppo

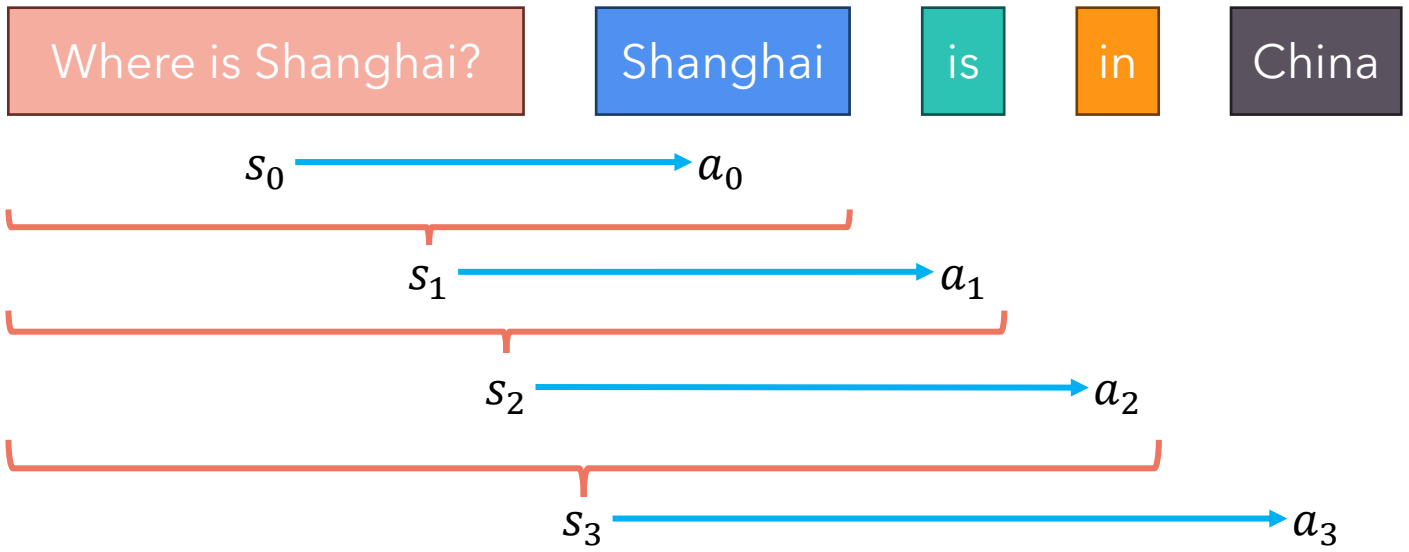
Trajectories in language models

When dealing with language models, we want to fine-tune the language model so that it selects the next token in such a way as to maximize the reward it gets.

$$\pi^* = \arg\max_{\pi} J(\pi),$$

What is a trajectory for a language model? It is a series of prompts (state) and their next tokens (actions).

$$\tau = (s_0, a_0, s_1, a_1, \ldots).$$



As we can see, when a language model is used to generate a response for a question (or in general to generate a text given a prompt), we can see a series of states and actions, which define a trajectory.

Policy Gradient Optimization

Imagine that we have a policy π_θ, parameterized by parameters θ. We want to change the parameters of the policy such that we maximize the expected return when using the policy. That is, we want to maximize the following:

$$J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} [R(\tau)]$$

When we have a deep neural network, our goal is to change the parameters of the network iteratively such that we minimize a loss function: this is a typical use case of Stochastic Gradient Descent. In our case, we want to maximize a function, so we can use Stochastic Gradient Ascent:

$$\theta_{k+1} = \theta_k + \alpha \, \nabla_\theta J(\pi_\theta)|_{\theta_k} .$$

The gradient of the policy is known as **policy gradient** and the algorithms that optimize the policy this way are called **policy gradient algorithms**.

The problem is that to calculate the gradient, we would need to evaluate it over all the possible trajectories, which is **computationally intractable** unless we have a very small state space.

Policy Gradient Optimization

Let's try to derive an expression for the policy gradient that we can compute in a reasonable time.

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) && \text{Expand expectation} \\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) && \text{Bring gradient under integral} \\
&= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) && \text{Log-derivative trick} \\
&= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)R(\tau)] && \text{Return to expectation form} \\
\therefore \nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right] && \text{Expression for grad-log-prob}
\end{aligned}
$$

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta).$$

$$P(\tau|\pi) = \rho_0(s_0)\prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

$$
\begin{aligned}
\nabla_\theta \log P(\tau|\theta) &= \cancel{\nabla_\theta \log \rho_0(s_0)} + \sum_{t=0}^{T}\left(\cancel{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log \pi_\theta(a_t|s_t)\right) \\
&= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t).
\end{aligned}
$$

This is an expectation, which means we can approximate it with a sample mean by collecting a set *D* of trajectories.

$$\hat{g} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau),$$

Recap of what we've done so far

We first found an expression of the gradient of the expected reward with respect to the parameters θ and then we approximated it using a sample mean.

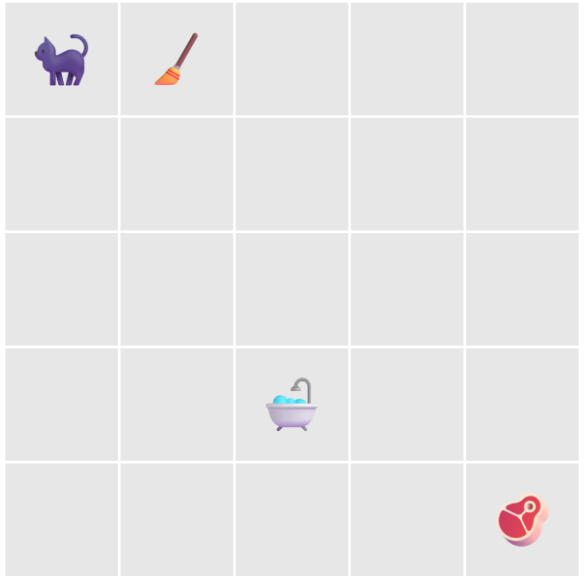
This is how we should proceed in practice:

1. Create a neural network that defines a policy (takes an input the current state of the agent and outputs the probability over the action space)
2. Use the network to sample trajectories and their corresponding rewards (we can run each trajectory for example for 100 steps or until the cat faints).
3. Use the sample to calculate the gradient
4. Run stochastic gradient ascent to update the parameters of the policy/network
5. Go back to 2

This is known as the REINFORCE algorithm in literature.

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),$$

$$\theta_{k+1} = \theta_k + \alpha \, \nabla_\theta J(\pi_\theta)|_{\theta_k} .$$



Umar Jamil – https://github.com/hkproj/rlhf-ppo

Generating trajectories for LMs

Remember the dataset of preferences that we built for the reward model? We can use questions from the data set and ask our model to generate the answer. We then calculate the reward for the generated answer and train the model according to the approximated gradient of the policy, as described in the REINFORCE algorithm.

Question (Prompt)
Where is Shanghai?
Explain gravity like I'm 5
What is 2+2?

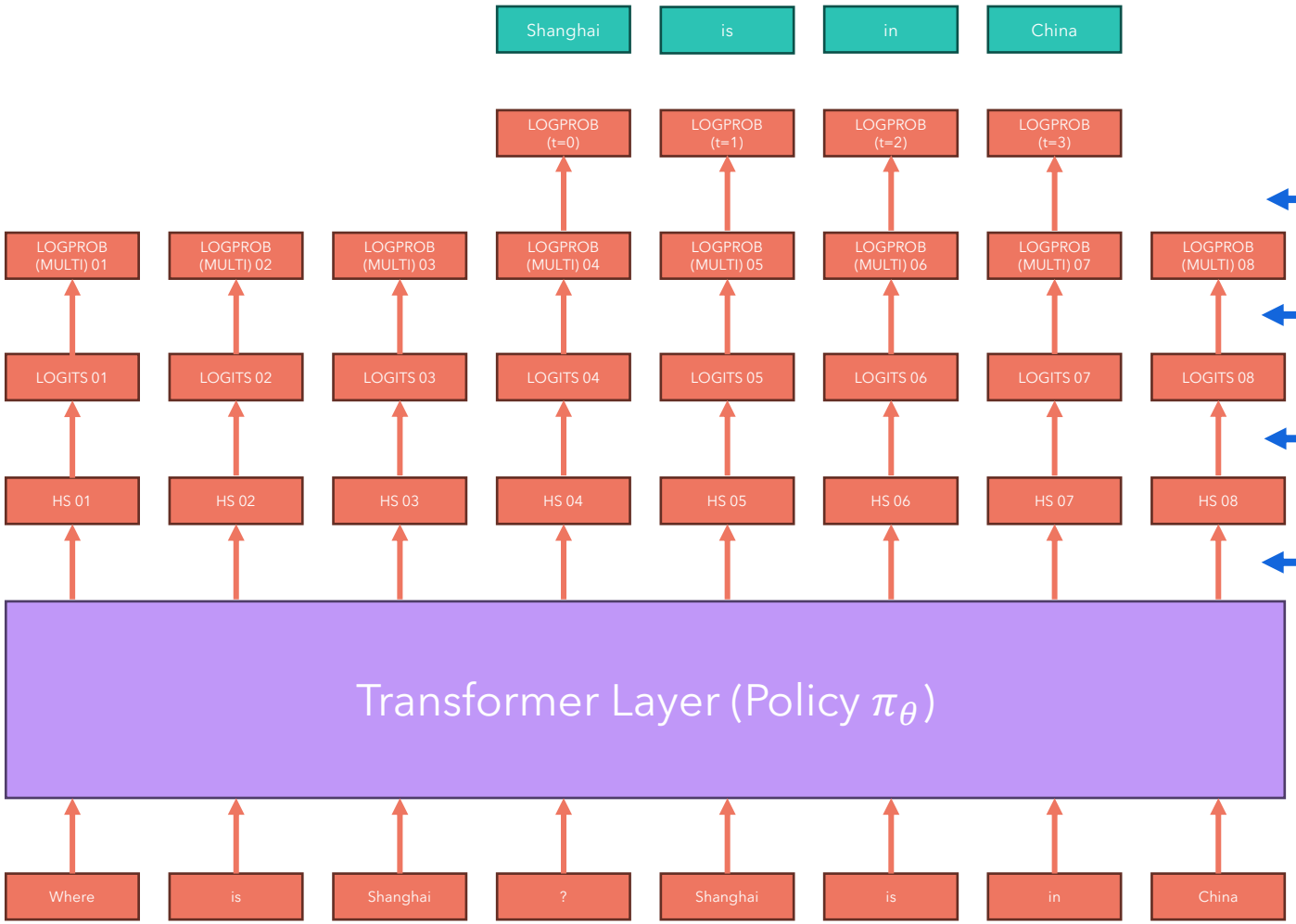
Since the text generation process results in a series of states (prompts) and actions (next tokens), we obtain a series of trajectories!

OK, but how do we do this in practice?

Calculating log probabilities of our policy (LM)

How can we calculate the log probabilities of a trajectory for a Language Model? Imagine our language model generated the following response for the given question. Let's see how we can leverage the generated response to calculate log probabilities over the single (state, action) pairs.

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),$$



Select the log probably corresponding to the action actually taken by the policy (the next token!)

Apply the $\log\_softmax$ function to generate log probabilities. $\log\_softmax(x) = \log(softmax(x))$

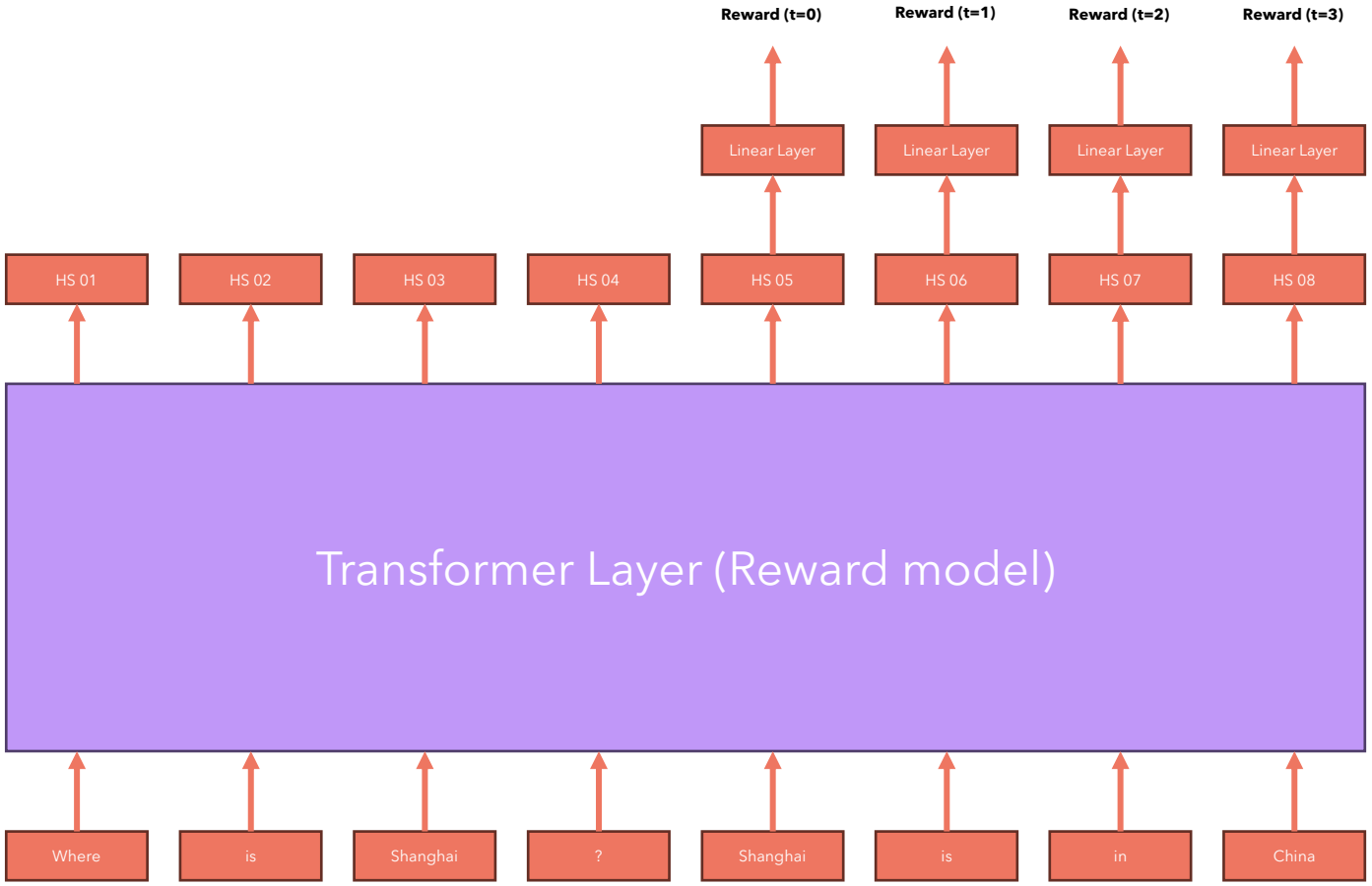
Apply the Linear projection to **all** the hidden states to generate logits for all the positions

Each hidden state encoder information only about previous tokens because we apply a **causal mask during self-attention**!

Calculating rewards for each trajectory

We can do the same to generate the reward for all the (state, action) pairs! This is because the reward model is usually a Language Model with a linear layer on top!

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),$$



Linear layer with only one output feature

Problems with Gradient Policy Optimization

First problem: the gradient approximation that we have found is unbiased (meaning that on average it will converge to the true gradient), but it exhibits high variance.

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),$$

Reducing variance: you can't alter the past

The estimator of the gradient that we have found, is multiplying the gradient of the log probabilities of each action in the current trajectory with the rewards obtained in the entire trajectory

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]$$

$$\nabla_\theta\big(J(\theta)\big) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\big(a_{i,t} \mid s_{i,t}\big) \right) \left(\sum_{t=0}^{T} r(s_{i,t}, a_{i,t}) \right)$$

For each action, we are also multiplying rewards that came before the action was taken.
It has been proven that past terms cancel out in expectation, so we can remove them.

$$\nabla_\theta\big(J(\theta)\big) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\big(a_{i,t} \mid s_{i,t}\big) \right) \left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}) \right)$$

Commonly known as "**rewards to go**"

Umar Jamil – https://github.com/hkproj/rlhf-ppo

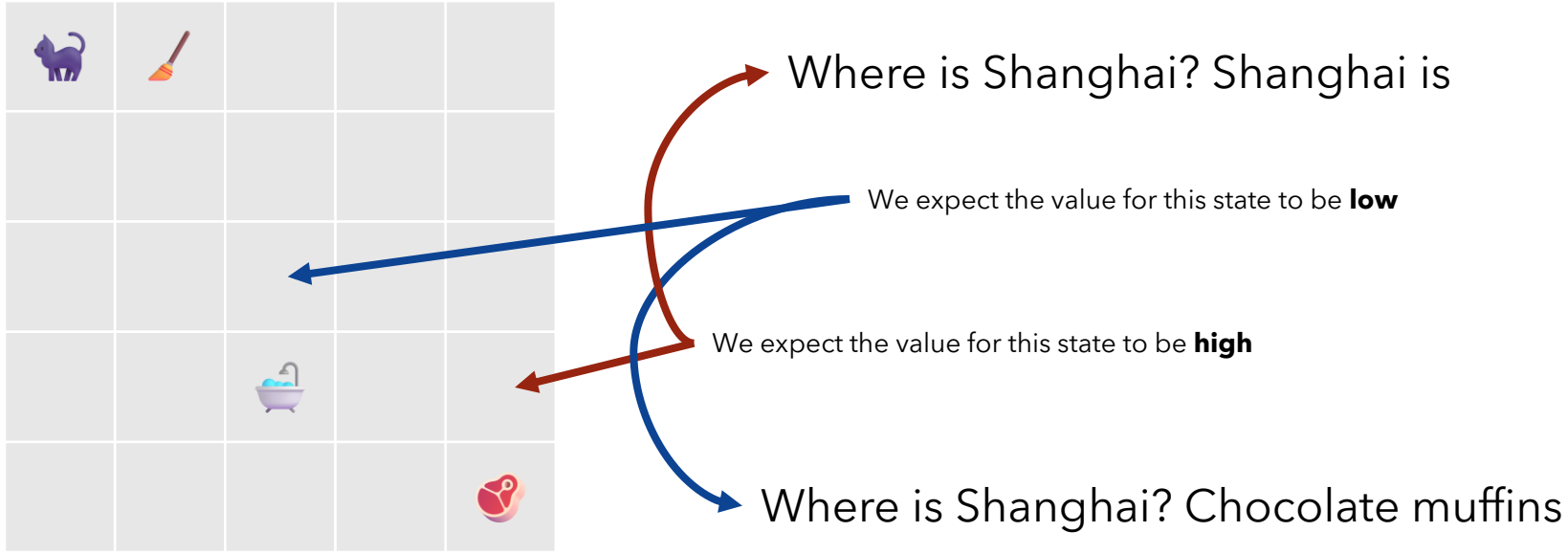
Reducing variance: baseline

It can be proven that subtracting a baseline from the rewards to go, still results in an unbiased estimator of the gradient.

$$\nabla_\theta(J(\theta)) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)\left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}) - b\right)$$

Baseline
Can also be dependent upon the state

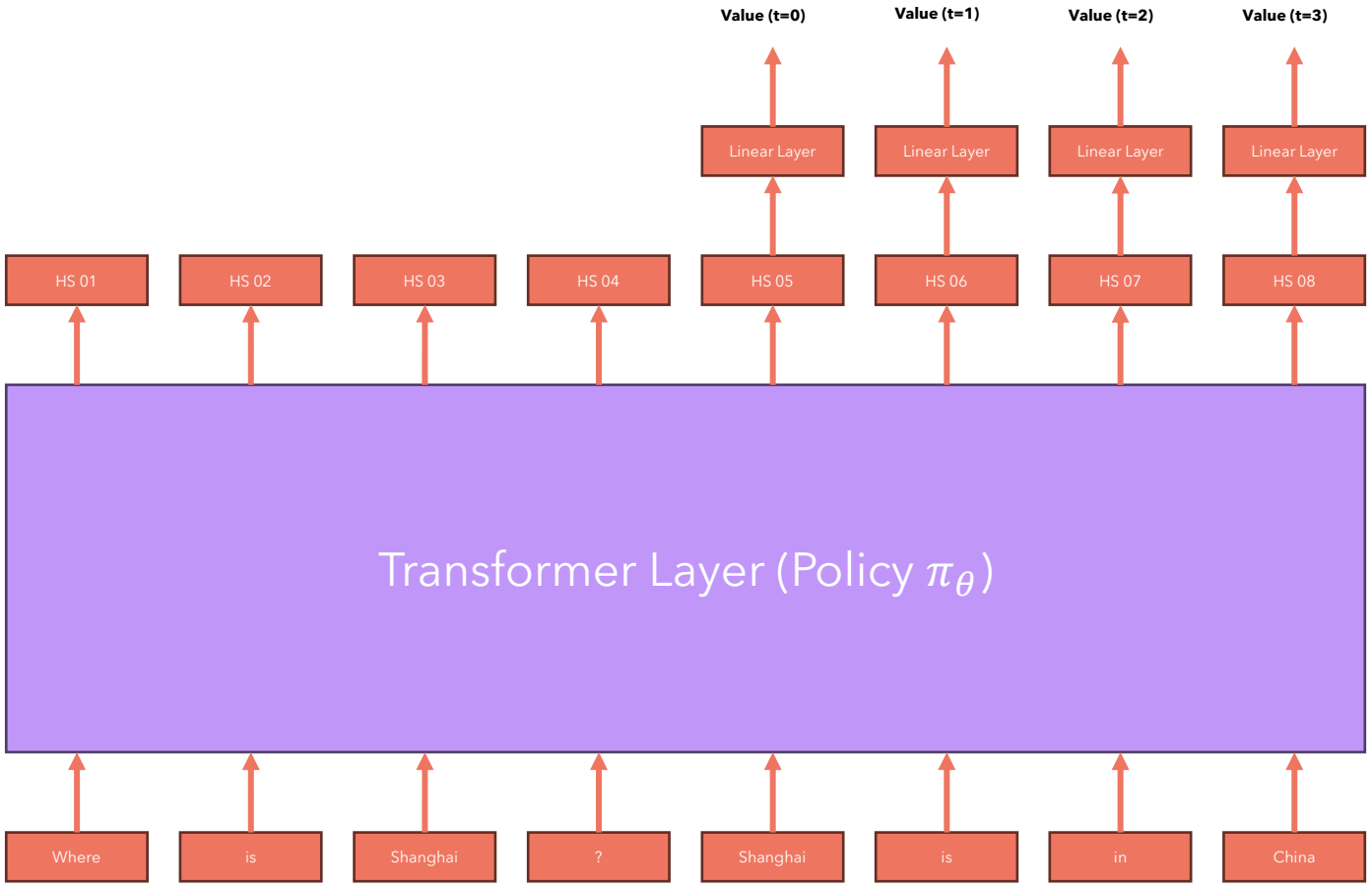
As baseline we will choose the value function $V^\pi(s)$ which indicates what is the future expected reward if the agent is in state ***s*** and then acts according to the policy π.



How do we estimate $V^{\pi}(s)$?

We an **additional** Linear layer on top of our Language Model (the policy π_θ) that estimates the value of the state at a particulate time step.

Please note that the Language Model (the policy π_θ) already has a Linear layer to transform hidden states into logits. The one shown below is another Layer that is added to the model.



Linear layer with only one output feature

Reducing variance: introducing Q and V

In the previous slide, we have found an expression for the "rewards to go" that can reduce the variance of our estimator. The "rewards to go" indicates how much reward our agent will receive if it starts from state ***s*** and takes action ***a***, and then acts according to the policy itself. This in RL is known as the Q function, which indicates what's the expected return if the agent starts at state ***s***, takes action ***a*** and then acts according to the policy.

$$\nabla_\theta(J(\theta)) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)\underbrace{\left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})\right)}$$

Should remind you of the **Q** function.

$$Q^\pi(s,a) = \underset{s'}{E}[r(s,a) + \gamma \underset{a'}{E}[Q^\pi(s',a')]]$$

Remember we said we can also use a baseline to reduce variance? Well let's use another function, called value function, as baseline, to further reduce the variance.

$$\nabla_\theta(J(\theta)) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)\underbrace{\left(Q^\pi(s_{i,t}, a_{i,t}) - V^\pi(s_{i,t})\right)}$$

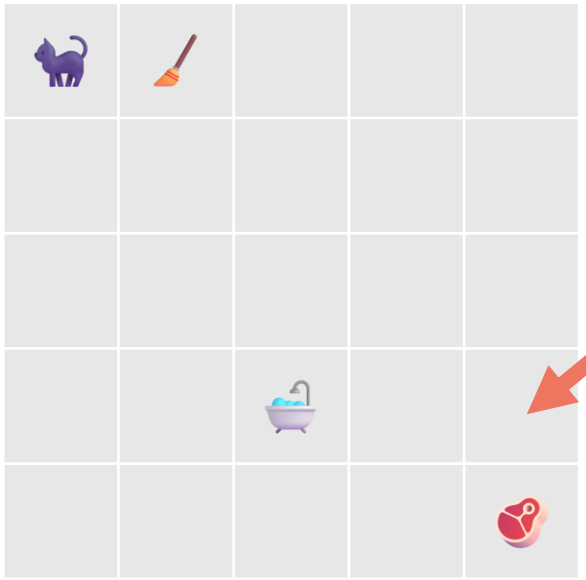
Known as the **advantage** function

Reducing variance: interpreting the advantage

Let's try to interpret the formula we have obtained

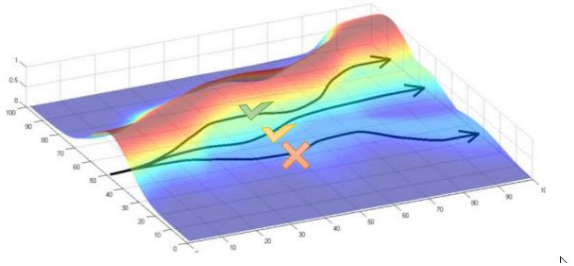
$$\nabla_\theta(J(\theta)) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)\left(Q^\pi(s_{i,t}, a_{i,t}) - V^\pi(s_{i,t})\right)$$

$$\nabla_\theta(J(\theta)) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)A^\pi(s_{i,t}, a_{i,t})$$



The advantage function tells us how better is to choose a particular action ***a*** in a state ***s*** over the average expectation we get by choosing randomly an action in the same state ***s***.

In this state, it gives the agent more reward to choose the action "go down" instead of randomly choosing an action. **It means the action "go down" is better than the average action in this state**.



Umar Jamil – https://github.com/hkproj/rlhf-ppo

Estimating the advantage term

We can estimate the advantage term in many ways, as follows:

$$\hat{A}^{\pi}(s_t, a_t) = [r(s_t, a_t) + \gamma V^{\pi}(s_{t+1})] - V^{\pi}(s_t)$$

$$\hat{A}^{\pi}(s_t, a_t) = [r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 V^{\pi}(s_{t+2})] - V^{\pi}(s_t)$$

$$\hat{A}^{\pi}(s_t, a_t) = [r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \gamma^3 V^{\pi}(s_{t+3})] - V^{\pi}(s_t)$$

If we stop too early, we will get a very **high bias** (because we are approximating the value function, and only using one "real" reward from our trajectory).
If we stop after many terms, we will get **high variance**. In order to solve this **bias-variance problem**, we can take the weighted sum of the terms to obtain the **Generalized Advantage Estimation**:

$$\delta_t = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$$

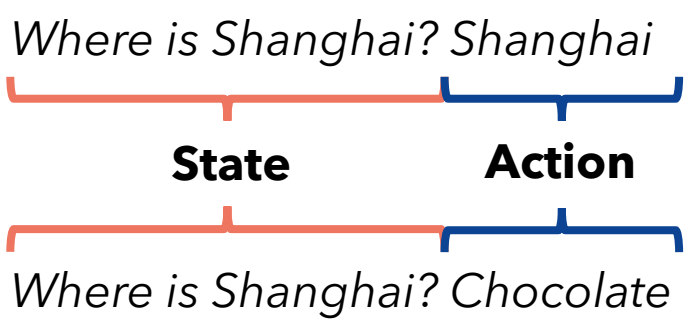
$$\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1}$$

This is a **recursive formula** that for the last term equals the first expansion
For the second-last term equals to the second expansion (weighted by λ)
Etc.

Advantage term for language models

In the case of language models, this result tells the policy (the language model) to increase the likelihood of choosing the next token, given a prompt (state) that in expectation results in "better than average" rewards. Which means that the language model will be forced to choose tokens that will more likely to lead to future tokens that comply with its reward model (that are more aligned with our training data set).

$$\nabla_\theta\big(J(\theta)\big) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta\big(a_{i,t} \mid s_{i,t}\big)\right) A^\pi\big(s_{i,t}, a_{i,t}\big)$$



This action (the word "Shanghai") will result in a **good response** and so, a high reward. This will train the model to select the word "Shanghai" more often if it sees the same prompt.

This action (the word "Chocolate") will result in a **bad response** and so, a low reward. This will train the model to select the word "Chocolate" less often if it sees the same prompt.

Problems with Gradient Policy Optimization

Second problem: the expectation that we've found, forces us to sample trajectories from our neural network every time we update the parameters.

$$\nabla_\theta(J(\theta)) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right) A^\pi(s_{i,t}, a_{i,t})$$

This can be very inefficient because when training a neural network, we update the parameters many times by taking small steps (according to the learning rate).

$$\theta_{k+1} = \theta_k + \alpha \, \nabla_\theta J(\pi_\theta)|_{\theta_k} .$$

Importance sampling & off-policy learning

Importance sampling allows to evaluate an expectation over a distribution X using samples taken from another distribution Y.

$$
\begin{aligned}
E_{x\sim p(x)}[f(x)] &= \int p(x)f(x)dx \\
&= \int \frac{q(x)}{q(x)}p(x)f(x)dx \\
&= \int q(x)\frac{p(x)}{q(x)}f(x)dx \\
&= E_{x\sim q(x)}\left[\frac{p(x)}{q(x)}f(x)\right]
\end{aligned}
$$

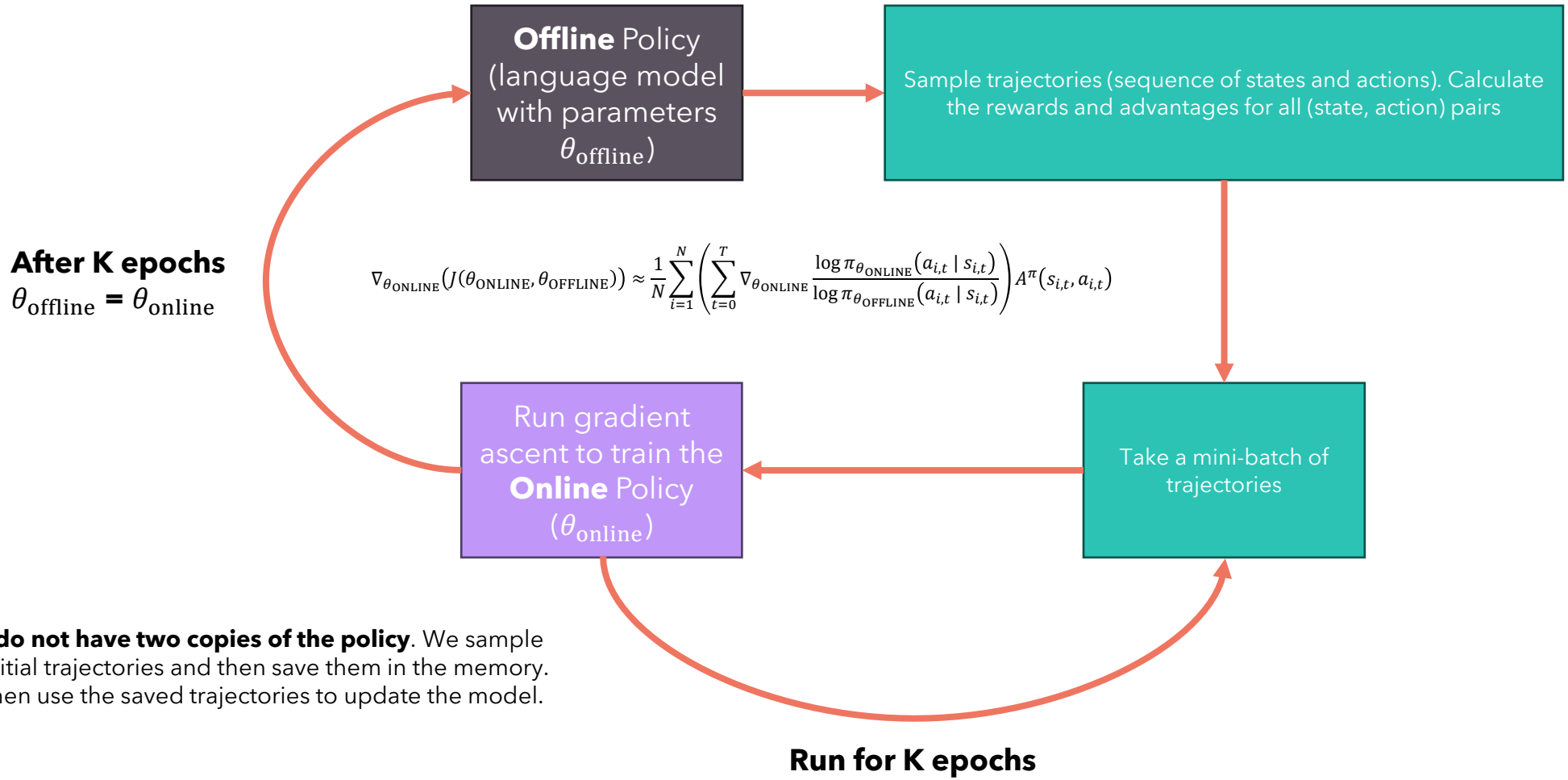
$$
\nabla_\theta(J(\theta)) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)A^\pi(s_{i,t}, a_{i,t})
$$

Trajectories sampled from π_θ

$$
\nabla_{\theta_{\text{ONLINE}}}(J(\theta_{\text{ONLINE}}, \theta_{\text{OFFLINE}})) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_{\theta_{\text{ONLINE}}}\frac{\log \pi_{\theta_{\text{ONLINE}}}(a_{i,t} \mid s_{i,t})}{\log \pi_{\theta_{\text{OFFLINE}}}(a_{i,t} \mid s_{i,t})}\right)A^\pi(s_{i,t}, a_{i,t})
$$

Trajectories sampled from $\pi_{\theta_{\text{OFFLINE}}}$

Off-Policy learning



*__We do not have two copies of the policy__. We sample the initial trajectories and then save them in the memory. We then use the saved trajectories to update the model.

Umar Jamil – https://github.com/hkproj/rlhf-ppo

Off-Policy learning (pseudo-code)

```
def off_policy_learning(frozen_model, model_to_train, reward_model, optimizer):
    for k in range(num_global_epochs):
        # The model is acting as the "offline policy"
        trajectories = sample_trajectories(model_to_train)

        # Compute the rewards, log probabilities, advantages, KL-divergences between the `frozen_model` and `the model_to_train`
        trajectories = update_trajectories_with_more_info(trajectories, reward_model, model_to_train, frozen_model)

        for j in range(num_ppo_epochs):
            mini_batch = get_random_mini_batch(trajectories)

            loss = ppo_algorithm(mini_batch)
            loss.backward() # Calculate the "loss"/objective
            optimizer.step() # Run gradient descent/ascent. Now the model is acting as the "online policy"
```

Umar Jamil – https://github.com/hkproj/rlhf-ppo

The PPO loss

$$L_{\text{POLICY}} = \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t, clip\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)$$

$$L_{\text{VF}} = \frac{1}{2}\left\| V_{\theta(s)} - \left(\sum_{t=0}^{T} \gamma^t r_t \mid s_0 = s\right)\right\|_2^2$$

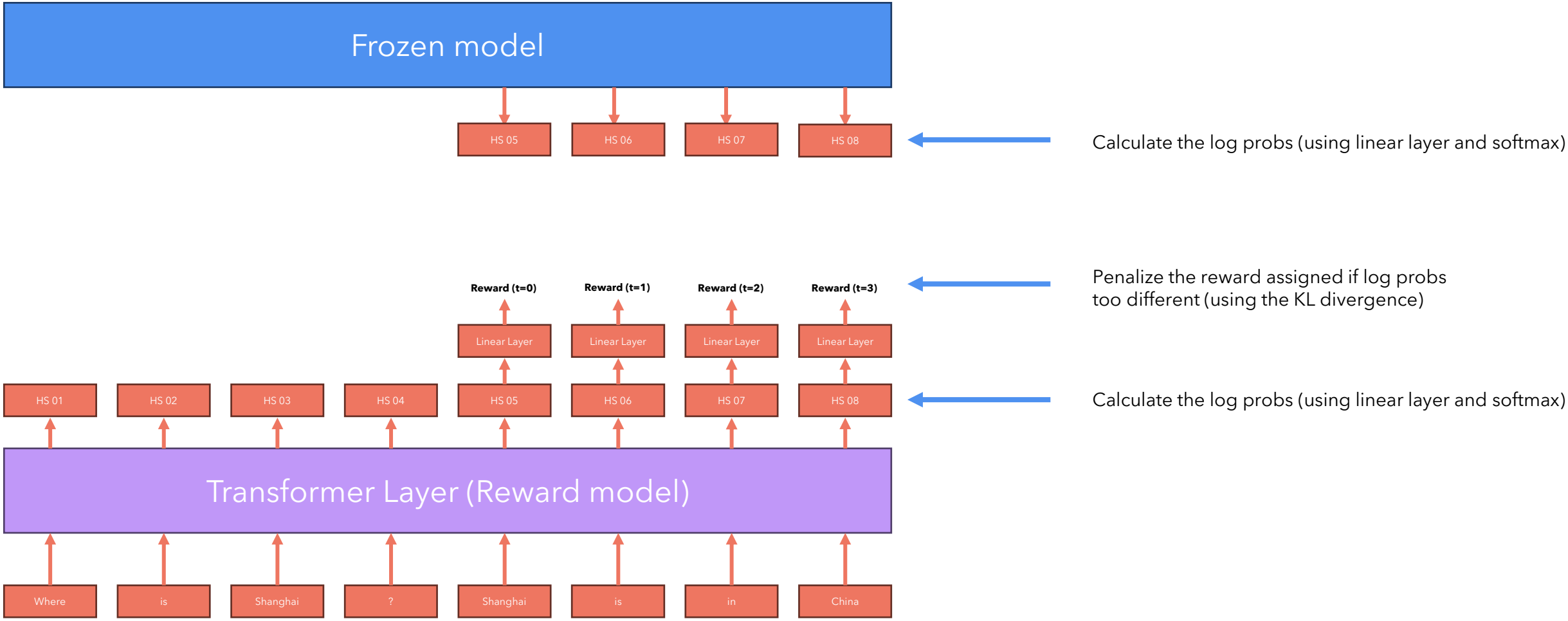
$$L_{\text{ENTROPY}} = -\sum_x p(x)\log p(x)$$

$$L_{PPO} = L_{\text{POLICY}} + c_1 L_{\text{VF}} + c_2 L_{\text{ENTROPY}}$$

Umar Jamil – https://github.com/hkproj/rlhf-ppo

Reward hacking

If we apply the "vanilla PPO" like described, the language model (our policy) may just learn to output whatever the reward model wants to see to maximize its return. We for sure want the language model to receive good rewards, but as the same time we want the language model to output something that still looks like the training data it was trained upon. For this reason, for every reward generated by the model, we penalize the reward by the KL-Divergence between the logits generated by the policy being optimized and a frozen version of the language model.



Umar Jamil – https://github.com/hkproj/rlhf-ppo

Talk is cheap. Show me the code.

- Linus Torvalds

Thanks for watching!
Don’t forget to subscribe for more amazing content on AI and Machine Learning!

Umar Jamil – https://github.com/hkproj/rlhf-ppo