Section 7: Inverse Reinforcement Learning and LLM RL

March 10th, March 11th

Author: Kevin Black

1 Inverse Reinforcement Learning

The standard RL problem is to learn an optimal policy given a reward function. Sometimes, we might want to do the opposite – learn a reward function given demonstrations that we assume to be optimal. We can start with the same graphical model we learned about last week, with a binary variable indicating optimality, and a probability of optimality that is proportional to the exponentiated reward:

$$p(\mathcal{O}_t|\mathbf{s}_t, \mathbf{a}_t, \psi) \propto \exp(r_\psi(\mathbf{s}_t, \mathbf{a}_t))$$

Except now, we are trying to learn the reward function, and we indicate this with a set of parameters ψ.

The first approach that might come to mind is maximum likelihood learning: make the observed trajectories look as likely as possible under our reward function. Given a dataset of observed trajectories $\{\tau_i\}$ that come from an (assumed) optimal policy $\pi^\star$, we can directly write down the maximum likelihood objective and simplify.

$$\begin{aligned}\max_\psi \mathbb{E}_{\tau_i \sim \pi^\star} \left[\log p(\tau_i|\mathcal{O}_{1:T}, \psi)\right] &= \max_\psi \mathbb{E}_{\tau_i \sim \pi^\star} \left[\log \left(\frac{1}{Z} \cdot p(\tau_i) \exp(r_\psi(\tau_i))\right)\right] \\ &= \max_\psi \mathbb{E}_{\tau_i \sim \pi^\star} \left[r_\psi(\tau_i)\right] - \log Z\end{aligned}$$

In the second step, we drop $p(\tau_i)$ because it does not depend on ψ. Z is the normalization constant that ensures that $p(\tau_i|\mathcal{O}_{1:T}, \psi)$ is a valid probability distribution, and it is the same for all i because it always integrates over all possible trajectories. It is also sometimes called the "partition function," and it is equal to

$$Z = \int p(\tau) \exp(r_\psi(\tau)) d\tau$$

To optimize the maximum likelihood objective, we can simply take the gradient with respect to ψ, and keep simplifying:

$$\begin{aligned}\nabla_\psi \left[\mathbb{E}_{\tau_i \sim \pi^\star} \left[r_\psi(\tau_i)\right] - \log Z\right] &= \mathbb{E}_{\tau_i \sim \pi^\star} \left[\nabla_\psi r_\psi(\tau_i)\right] - \frac{1}{Z}\frac{dZ}{d\psi} \\ &= \mathbb{E}_{\tau_i \sim \pi^\star} \left[\nabla_\psi r_\psi(\tau_i)\right] - \frac{1}{Z}\int p(\tau) \exp(r_\psi(\tau)) \nabla_\psi r_\psi(\tau) d\tau \\ &= \mathbb{E}_{\tau_i \sim \pi^\star} \left[\nabla_\psi r_\psi(\tau_i)\right] - \int p(\tau|\mathcal{O}_{1:T}, \psi) \nabla_\psi r_\psi(\tau) d\tau \\ &= \mathbb{E}_{\tau_i \sim \pi^\star} \left[\nabla_\psi r_\psi(\tau_i)\right] - \mathbb{E}_{\tau \sim p(\tau|\mathcal{O}_{1:T}, \psi)} [\nabla_\psi r_\psi(\tau)] \end{aligned} \tag{1}$$

Now we have two expectations: one over the observed trajectories, and one over $p(\tau|\mathcal{O}_{1:T}, \psi)$, which is the distribution over trajectories induced by the "soft optimal policy" for our *current* reward function r_ψ. We say "soft optimal policy" rather than "optimal policy" because we know from last week that this is what the control-as-inference framework gives us when we condition on optimality.

1.1 Estimating the Gradient

The first term is easy – we can just sample trajectories from our dataset and compute the gradient of our reward function, as in any supervised learning algorithm. The second term is harder, because we first need the "soft optimal policy". Lucky for us, we've already learned about an entire class of algorithms – max-ent RL – that happen to learn the exact same soft optimal policy as defined in the control-as-inference framework. That gives us a sketch for a practical algorithm:

1. Run a max-ent RL algorithm to convergence using our current reward function, r_ψ.

2. Use the resulting policy to sample trajectories from $p(\tau|\mathcal{O}_{1:T}, \psi)$.
3. Sample some trajectories from our expert policy $\pi^\star$ (i.e., our demonstration dataset).
4. Perform one step of gradient ascent on ψ using Equation (1).
5. Repeat until convergence.

The problem is that step 1 is very expensive – we have to run an entire max-ent RL algorithm to convergence just to perform one gradient update on the objective we actually care about. If we get lazy and don't run step 1 until convergence, our estimate is now biased because we're not sampling from $p(\tau|\mathcal{O}_{1:T}, \psi)$, but rather from some arbitrary distribution $\pi(\tau)$, where π is some arbitrary policy that we got from step 1. However, we can correct for this bias using *importance sampling*:

$$\text{importance weight} : w_j = \frac{\text{target distribution}}{\text{actual distribution}} = \frac{p(\tau_j|\mathcal{O}_{1:T}, \psi)}{\pi(\tau_j)} = \frac{p(\tau_j)\exp(r_\psi(\tau_j))}{\pi(\tau_j)}$$

When computing importance weights, we can ignore the normalizing constant (partition function) because it cancels out when we later divide by the sum of the weights. If we continue simplifying the importance weight, we ultimately arrive at a very simple expression:

$$\frac{p(\tau_j)\exp(r_\psi(\tau_j))}{\pi(\tau_j)} = \frac{p(\mathbf{s}_1)\prod_t p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\exp(r_\psi(\mathbf{s}_t, \mathbf{a}_t))}{p(\mathbf{s}_1)\prod_t p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\pi(\mathbf{a}_t|\mathbf{s}_t)} = \frac{\exp(r_\psi(\tau_j))}{\prod_t \pi(\mathbf{a}_t|\mathbf{s}_t)}$$

The things we don't know (the transition dynamics) cancel out, and we are left with the things we do know – our current policy π and our current reward function r_ψ! This gives us the final algorithm:

1. Run one step of a max-ent RL algorithm using our current reward function, r_ψ, to update our policy π.
2. Sample some trajectories $\{\tau_j\}$ from π.
3. Sample some trajectories $\{\tau_i\}$ from our expert policy $\pi^\star$ (i.e., our demonstration dataset).
4. Perform one step of gradient ascent on ψ using an importance-weighted version of Equation (1):

$$\frac{1}{N}\sum_i \nabla_\psi r_\psi(\tau_i) - \frac{1}{\sum_j w_j}\sum_j w_j \nabla_\psi r_\psi(\tau_j)$$

5. Repeat.

Instead of restarting RL from scratch on step 1, we now persist π across iterations. This makes the algorithm look kind of like an aversarial game: step 1 tries to make the policy more optimal, and step 4 makes the expert demonstrations look more optimal while making the policy look less optimal in comparison.

2 LLM RL

Modern LLM training is broadly split into 3 phases: pre-training (next-token prediction on a huge corpus of data), supervised fine-tuning (SFT; next-token prediction on a smaller, more curated dataset), and RL. While there are innumerable variants of LLM RL, we an break it down into two more broad categories: RL from preferences (RLHF, RLAIF, etc.) and RL with verifiable rewards (often associated with "reasoning" or "thinking" models). The difference is in the reward function – the former uses a *learned* reward function, often from human preference data, whereas the latter uses a *verifiable* reward function, such as "does this code run" or "is this answer correct".

Let us first devise a simple formalism for the LLM RL problem. It will just be a one-step deterministic MDP (also called a contextual bandit). The LLM, π_θ, takes a prompt ($\mathbf{s}$) and generates a completion ($\mathbf{a}$), and then receives a reward $r(\mathbf{s}, \mathbf{a})$. We don't need to break it down further because an LLM is an autoregressive model, and we can get the probability of an entire completion by multiplying together the probability of each token. The only reason to use a multi-step MDP is to support intermediate rewards (sometimes called process rewards), but we won't cover those in this section.

2.1 RL from Preferences

Let us first talk about learning a reward function, $r_\psi(\mathbf{s}, \mathbf{a})$, from paired human preference data. This is the main breakthrough that first turned large pretrained language models like GPT-3 into useful assistants like ChatGPT (cf. Ouyang et al. (2022)).

We first collect some rollouts from our SFT (supervised fine-tuning) model, $\tau_i \sim \pi_{\text{SFT}}$. We then ask humans to compare pairs of completions and ask humans to choose which one they prefer, generating a preference dataset $\{\tau_i \succ \tau_j\}_{i,j}$. To learn a reward function from preferences, we turn to the *Bradley-Terry model*, a probabilistic model in which the probability of seeing $\tau_i \succ \tau_j$ is exponentially more likely the larger the difference in reward between τ_i and τ_j:

$$\begin{aligned} p(\tau_i \succ \tau_j) &= \frac{\exp(r_\psi(\tau_i))}{\exp(r_\psi(\tau_i)) + \exp(r_\psi(\tau_j))} \\ \log p(\tau_i \succ \tau_j) &= \log\left(\frac{\exp(r_\psi(\tau_i))}{\exp(r_\psi(\tau_i)) + \exp(r_\psi(\tau_j))}\right) \\ &= \log\left(\frac{\exp(r_\psi(\tau_i) - r_\psi(\tau_j))}{\exp(r_\psi(\tau_i) - r_\psi(\tau_j)) + 1}\right) \\ &= \log \sigma(r_\psi(\tau_i) - r_\psi(\tau_j)) \end{aligned}$$

where σ is the sigmoid function, $\sigma(x) = \frac{\exp(x)}{\exp(x)+1}$. This is a nice, easily differentiable objective, and thus we can do maximum likelihood on $\{\tau_i \succ \tau_j\}$ to learn our reward function.

$$\psi \leftarrow \arg\max_\psi \sum_{i,j} \log \sigma(r_\psi(\tau_i) - r_\psi(\tau_j))$$

2.2 Policy Gradient for LLMs

Once we have a reward function, we can maximize the reward using policy gradient methods. That can be the vanilla policy gradient, or REINFORCE estimator:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathbf{s},\mathbf{a}\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) r(\mathbf{s}, \mathbf{a})]$$

where $\mathbf{s}, \mathbf{a}$ are prompt-completion pairs coming from the current model π_θ. We can also use an importance-weighted estimator (e.g., PPO):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathbf{s},\mathbf{a}\sim\bar{\pi}}\left[\frac{\pi_\theta(\mathbf{a}|\mathbf{s})}{\bar{\pi}(\mathbf{a}|\mathbf{s})} \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) r(\mathbf{s}, \mathbf{a})\right]$$

which can improve sample efficiency by re-using the same $\mathbf{s}, \mathbf{a}$ pair for multiple gradient steps on θ. However, in practice we rarely do more than a few gradient steps before setting $\bar{\pi} \leftarrow \pi_\theta$ and collecting new samples.

In previous sections, we learned how to reduce the variance of policy gradient estimators by using a *baseline*; we proved that you can replace $r(\mathbf{s}, \mathbf{a})$ with $r(\mathbf{s}, \mathbf{a}) - b(\mathbf{s})$ for any arbitrary function b without biasing the estimator. Following non-LLM RL work, early LLM RL pipelines trained per-token value function baselines and additionally computed advantages using GAE. However, more recent work (Shao et al., 2024) has found that a very simple per-prompt averaging baseline can work just as well:

$$b(\mathbf{s}_i) = \frac{1}{K}\sum_{k=1}^{K} r(\mathbf{s}_i, \mathbf{a}_k)$$

where $\{\mathbf{a}_1, \ldots, \mathbf{a}_K\}$ are K different completions sampled for the *same* prompt $\mathbf{s}_i$. This is called GRPO, and intuitively, it ensures that the policy gradient considers only how good a completion is compared to other completions for the same prompt.

2.3 Reference Model Regularization

There is one more problem to solve, which is that learned reward functions $r_\psi(\mathbf{s}, \mathbf{a})$ are not perfect. If RL is run indefinitely, the model will eventually learn to "hack" the reward function, achieving high rewards while outputting nonsense text. The simplest and most common solution is to add a KL regularization term to the reward, which penalizes the model for deviating from a reference model π_{ref} (often the SFT model that the RL process starts from).

$$\bar{r}(\mathbf{s}, \mathbf{a}) = r_\psi(\mathbf{s}, \mathbf{a}) - \beta D_{\text{KL}}(\pi_\theta(\mathbf{a}, \mathbf{s}) \,\|\, \pi_{\text{ref}}(\mathbf{a}|\mathbf{s}))$$

References

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.