Effectively all modern language model systems are based on *reasoning models*: language models that have been explicitly trained to generate a sequence of tokens *before* tey output their final answers. These tokens are variously called a *thinking trace*, or *reasoning tokens*, or a *chain-of-thought*, though the first two are sometimes used specifically to indicate that the model has been trained via reinforcment learning to perform said reasoning.

In this note we'll learn how to train a reasoning model, as well as how to think about the capabilities and limitations thereof.

Motivation: scaling test-time computation

So far in the class we've discussed the benefits of scaling the amount of computation we can put into training our models. To this end, we discussed pretraining and architectural parallelism, to spend floating-point operations to learn from text.

Another axis along which we can spend computation in order to achieve better performance is at test-time, or deployment time. That is, when a user asks a query, we want to be able to have a model (or system) "think harder" in order to better tackle a problem.

A simple way to implement this is to sample multiple answers in parallel, and then aggregate somehow. Concretely, consider a model we've trained to sample $\hat{y} \sim p_\theta(y \mid x)$, where x is a mathematical lemma and $\hat{y}$ is an attempted proof of that lemma. What we've done so far is just sample once:

$$\hat{y} \sim p_\theta(y \mid x), \tag{1}$$

and we return the one sample $\hat{y}$. Instead, we could sample k times, with k different seeds, to generate independent samples:

$$\hat{y}_i \sim p_\theta(y \mid x) \tag{2}$$
$$i \in \{1, \ldots, k\} \tag{3}$$

Now we have k possible responses, and we can't return all k, so we have to aggregate—use the set of k to choose a single response. The easiest case here is if we have a verifier—like how sometimes in mathematics, we can automatically check the correctness of a proof (say, if it's in Lean,) or maybe it's a set of test cases. A verifier is a function $v(x, \hat{y})$ such that:

$$v(x, \hat{y}) = \begin{cases} 1 & \hat{y} \text{ is a correct answer to } x \\ 0 & \text{o.w} \end{cases} \tag{4}$$

So, our aggregation is simply to walk over all k samples and return the first one that passes our verifier. Something like

$$\text{return } \hat{y}_i \tag{5}$$
$$\text{s.t. } r(\hat{y}_i) = 1 \tag{6}$$

though, it doesn't matter in this case which correct answer we return. If we don't have a verifier, we might instead use some heuristic for deciding betewen the k. For example, we might try to find a consensus candidate answer: the $\hat{y}_i$ most like the *others* that we generated (Wang et al., 2023). If we let $\phi(\hat{y})$ be a representation of $\hat{y}$ under the Transformer language model that generated it (say, some representation somewhere in

the network), then we might use that for our aggregation:

$$\text{return } y_i \tag{7}$$

$$\text{s.t. } i = \max_{i=1}^{k} \left(\sum_{j=1}^{k} \phi(\hat{y}_i)^\top \phi(\hat{y}_j) \right), \tag{8}$$

So here, we're using representation similarity to compute which y_i is most like the others. Why might this work? The intuition here is that if the model is good but sometimes makes mistakes, sampling multiple times and then choosing a consensus prediction may reduce the chance we return one of the bad samples (since, say, most samples are good.)

Chain-of-thought prompting

One problem with sampling many possible answers to a question is that none of them may be right, in which case no aggregation strategy will help. The problems we're trying to solve may be too difficult to solve directly; much like any hard problem, it may be necessary to work through intermediate problems, think it through, etc. A natural way to do this in a language model is to first tell it to do so (instruction following!) and then allow it to first generate tokens before its answer (Wei et al., 2022).

Consider the following input transformation: for an input sequence $x \in \mathcal{V}^*$, we'll simply prepend the sequence of tokens corresponding to our command:

$$x_{\text{cot}} = \text{“Think about this problem step by step:”} + x \tag{9}$$

With that now defined, we'll split the generation into two portoins: the chain of thought, and the answer. (This can be done, e.g., by adding special delimiter tokens, or just XML tags in plain text, like `<think>` and `<\think>`:

From here on we'll denote model outputs as follows:

$$\hat{c} \sim p_\theta(c \mid x) \qquad \text{(thinking trace)} \tag{10}$$

$$\hat{y} \sim p_\theta(c \mid x, \hat{c}) \qquad \text{(predicted answer)}, \tag{11}$$

which, recall, we can write equivalently as $\hat{y}, \hat{c} \sim p_\theta(y, c \mid x)$.

How do we train better chains of thought?

What makes a good chain of thought? Well, to a first approximation, whatever makes the model come to a correct answer. When we prompt the model to think step by step, we certainly hope the model's chain of thought will be helpful (and it is!) but if we want even better chains of thought, we might try training the model to do so. We begin this with simple supervised finetuning, on a dataset of inputs, chains of thought, and corresponding outputs $\mathcal{D} = \{(x_i, c_i, y_i)\}_{i=1}^n$:

$$\min_\theta - \sum_{i=1}^{n} \log p_\theta(y_i, c_i \mid x), \tag{12}$$

where note that we are maximizing the likelihood of both the traces and the corresponding answers (so, our loss mask masks out only each x_i.)

Note as well here that we are training the model to generate chains of thought even when not prompted to! That is, in our SFT data, our x are not x_{cot}; they don't have a prompt to think step by step. Instead, after this SFT, the default behavior of the model will be to think first and then answer.

Toolformer and agents

tbd

On-policy vs. off-policy behaviors

So, how can our SFT fail to get us the behaviors we want? A few ways. For very hard problems, it's possible that we don't even know a useful chain of reasoning that gets us to the problem. But for now, let's assume this isn't an issue.

Another issue is that the model is being trained to perform behaviors during SFT that it will find difficult to perform on new problems. Here's an intuition. To solve difficult problems, you have to follow reasoning steps you understand, so you can apply then in new scenarios. If a tutor teaches you a few examples that use methods that are unfamiliar to you, it may be difficult for you generalize. If instead a tutor gives you examples that use methods that you yourself are already familiar with, you're much more likely to be able to apply them elsewhere. Not only this, but as you improve, you want the examples you're learning from to improve with you — if they don't that may limit your growth.

The language we use to discuss these things we often call **on-policy vs. off-policy** behaviors. We won't necessarily use this language, but you'll see it around so you need to know it. A sequence is on-policy simply if is drawn from the distribution defined by the model. So, if $\hat{y}, \hat{c} \sim p_\theta(y, c \mid x)$, then $\hat{y}$ and $\hat{c}$ are on-policy for p_θ. If not, then they're off-policy. Now, if some gave you a single sequence, say c', and a model p_θ, could you tell if c' is on-policy for p_θ? Not without a bunch of work; it's not obvious from just, say the likelihood $p_\theta(c')$. But if, say, you have some dataset you constructed somehow other than sampling from the model (like in the reasoning SFT we talked about just now,) then it's off-policy.

So, we want a way to train models to reason better in an on-policy way, to better hope that they generalize. For this, we'll come back to verifiers.

Verifiers

When we constructed our SFT dataset of reasoning, we had some method, maybe human evaluation, maybe some expensive process, of determining that indeed for input x, the reasoning trace c and the answer y are good or expected. But if we want on-policy learning of reasoning, all we have are maybe an input x, possibly an expected output y, and no notion of what the right reasoning trace c is. This brings us back to the intuition that the right reasoning trace is whatever gets the model to the right answer, which we'll check with a verifier.

It's an old adage in computer science that verifying the correctness of an answer is (probably) easier than generating a candidate answer. A simple example of a verifier might just check that the answer is correct, so

$$\hat{y}, \hat{c} \sim p_\theta(y, c \mid x) \tag{13}$$

$$r(x, \hat{y}) = \mathbf{1}\{y = \hat{y}\}. \tag{14}$$

This works when there is roughly one potential correct answer to the given question. For example, in simple maths word problems, like *Iroh has ten cabbages and gives half to Zuko...*, there's a single integer answer, leading to easy verification. In something more complex like code, we might instead have a verifier that checks all the test cases;

for tests $t_1, \ldots, t_q$, we might have:

$$r(x, \hat{y}) = \forall_{j=1}^{q} t(x, \hat{y}), \tag{15}$$

which we'd run in a container computing environment.

Reinforcement Learning from Verifiable Rewards

So, if we have a verifier, we have a way of telling whether our model's reasoning traces are good—if they lead to answers that are right. Here's a simple way to train the model to then generate more such reasoning traces:

We take our dataset $\mathcal{D} = \{(x_i)\}_{i=1}^{n}$ of just input problems, we sample a reasoning trace and answer for each, and then just train on the ones we got right:

$$\min_{\theta} -\sum_{i=1}^{n} \log p_\theta(\hat{c}, \hat{y} \mid x) r(x, \hat{y}). \tag{16}$$

So, is this on-policy learning? Well, for the first gradient step it is, but then afterwards, it's not. Let's make this clearer. We sampled all of our reasoning traces and answers *first*, before we did made any updates to our parameters. Let's call that initial model p_{θ_0}, indicating our initial model, while p_θ will refer to the "current" model throughout the optimization. So, we're working with:

$$\min_{\theta} \mathbb{E}_{\hat{y}, \hat{c} \sim p_{\theta_0}(y, c \mid x)} \left[-\log p_\theta(y, c \mid x) r(x, \hat{y}) \right] \tag{17}$$

Note how the expectation we're taking is with respect to the initial model, not with respect to the current parameters, so as we keep finetuning the model, we stray further from the distribution from which we sampled the sequences.

We want to instead maximize the likelihood of generating high-reward responses:

$$\max_{\theta} \ \mathbb{E}_{x \sim \mathcal{D}, \hat{y}, \hat{c} \sim p_\theta(y, c \mid x)} \left[r(x, \hat{y}) \right], \tag{18}$$

a traditional reinforcement learning goal.

Simple Grouped RL

Let's start with a simple way to try to solve our reinforcement learning problem: for each problem x, we just sample a batch of k guesses (like we did when we were doing parallel sampling before!), compute the reward of each of them, and **normalize** the reward by the average reward of the batch. That is, we compute:

$$\hat{c}_i, \hat{y}_i \sim p_\theta(y_i, c_i \mid x) \qquad i = 1, \ldots, k \tag{19}$$

$$\bar{r}(x, \hat{y}_i) = r(x, \hat{y}_i) - \frac{1}{k} \sum_{j=1}^{k} r(x, \hat{y}_j) \tag{20}$$

$$\max_{\theta} \mathop{\mathbb{E}}_{x \sim \mathcal{D},\, \hat{c}_i, \hat{y}_i \sim p_\theta(y, c \mid x),\, i=1,\ldots,k} [r(x, \hat{y}_i)] \tag{21}$$

but we use $\bar{r}$, the normalized rewards for optimization, and then our gradient step is of the form:

$$\nabla_\theta \left(-\sum_{i=1}^{k} \log p_\theta(\hat{c}, \hat{y} \mid x) \bar{r}(x, \hat{y}_i) \right) \tag{22}$$

Intuitively, look how we're (1) always sampling from the up-to-date model, and (2) basically performing SFT weighted by how much better than average the reward of each guess is.

GRPO

GRPO (Shao et al., 2024) is a lot like our simple grouped reward algorithm. The first change is that our normalized reward will also correct for the standard deviation across a batch:

$$\bar{r}(x, \hat{y}_i) = \frac{r(x, \hat{y}) - \mu_r}{\sigma_r} \tag{23}$$

Importance sampling. The second change is the introduction of the following importance sampling term. We consider taking multiple gradient steps on a single batch of data (or having our log-likilhoods at backpropagation time being different from those at sampling time, which can happen due to prefill-vs-sampling differences in numerical realities on the GPU,)

$$q_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t}) = \frac{p_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})}{p_{\theta_{\text{old}}}(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})} \tag{24}$$

Note how for any function $f(x, y)$ if I take the expectation over (that is, sample according to) $p_{\theta_{\text{old}}}$ and reweight by q, then it's as if I'm taking the expectation over my real p_θ:

$$\mathbb{E}_{\hat{y} \sim p_{\theta_{\text{old}}}(y|x)} \left[\frac{p_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})}{p_{\theta_{\text{old}}}(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})} f(x, y) \right] \tag{25}$$

$$= \sum_{x \in D, y_t \in \mathcal{V}} p_{\theta_{\text{old}}}(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t}) \frac{p_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})}{p_{\theta_{\text{old}}}(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})} f(x, y) \quad \text{(defn of expectation)} \tag{26}$$

$$= \sum_{x \in D, y_t \in \mathcal{V}} p_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t}) f(x, y) \quad \text{(cancellation)} \tag{27}$$

$$\mathbb{E}_{\hat{y} \sim p_\theta(y|x)} [f(x, y)] \quad \text{(defn of expecation)} \tag{28}$$

Note how this has been done at the per-token level. These are *importance sampling* weights.

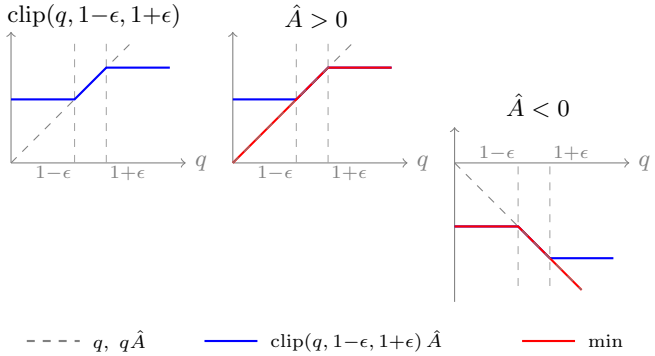
Clipping. But, if our probabilities are *too* different, we don't want our importance sampling weights to be crazy. We do this through **clipping**. For a specific token $\hat{y}_{i,t}$, without clipping we'd be maximizing, in expectation:

$$q_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t}) \bar{r}(\hat{y}_i, x), \tag{29}$$

where $\bar{r}(\hat{y}_i, x)$ is our normalized reward; often called $\hat{A}(\hat{y}_{i,t}, x)$ the *advantage* (and sometimes is dependent on the token; here our rewards are for the whole output, they don't differ per token.) The importance sampling ratio may be greater than or less than 1, so we clip the importance sampling ratio to a range between $1 - \epsilon$ and $1 + \epsilon$. Then the advantage may be greater than or less than zero, and so to be conservative, we take the minimum of the clipped and unclipped importance sampling-weighted advantage:

$$\min\Big(q_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t}) \hat{A}(\hat{y}_i, x), \text{clip}\big(q_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t}), 1 - \epsilon, 1 + \epsilon\big) \hat{A}(\hat{y}_i, x) \Big) \tag{30}$$

To get a visual intuition of the clipping, here are some LLM-generated nice plots of the different possibilities of advantage, importance sampling ratio, and clipping, which are a lot like Figure 1 in Schulman et al. (2017), which introduced the clipping for PPO, with some extra explanation:



Finally, here's GRPO, which again is a lot like simple grouped reward, but with the clipping and importance sampling ratios:

$$q_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t}) = \frac{p_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})}{p_{\theta_{\text{old}}}(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})} \tag{31}$$

$$\max_\theta \mathop{\mathbb{E}}_{\substack{x\sim\mathcal{D} \\ \hat{y}_i\sim p_{\theta_{\text{old}}}(y|x) \\ i=1,\ldots,k}} \left[\frac{1}{k} \sum_{i=1}^{k} \frac{1}{|\hat{y}_i|} \sum_{t=1}^{|\hat{y}_i|} \left\{ \begin{array}{l} \min\Big(q_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t})\, \hat{A}(\hat{y}_i, x), \\ \quad \text{clip}\big(q_\theta(\hat{y}_{i,t} \mid x, \hat{y}_{i,<t}),\, 1-\epsilon,\, 1+\epsilon\big) \hat{A}(\hat{y}_i, x) \Big) \\ \quad - \beta\, \text{KL}\big(p_\theta(\cdot \mid x, \hat{y}_{i,<t}) \,\|\, p_{\text{SFT}}(\cdot \mid x, \hat{y}_{i,<t})\big) \end{array} \right\} \right] \tag{32}$$

References

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. (2024). DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. (2023). Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q. V., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.