Project Log

Phil Saad

September 2025

A place for me to log what I've been doing for my project.

1 Setup (model training)

- picked gsm8k as my main dataset to train on because it seemed simple enough that I could get good results out of a small model + lora, but nontrivial enough that RL on COT reasoning made sense.
- Picked qwen2.5 1.5b because it had $\sim$ 50 percent pass rate, with pretty high pass@8, indicating that there was a lot of room for it to learn.
- Formatted dataset uisng the deepseek r1 template, including instructions and tag structure. Worked well so why not.
- Finetuned on the dataset until format adherence went up.
- Set up some comprehensive eval code For training, started with a batch size and learning rate from the dr grpo paper github. Also used only on-policy updates for simplicity.
- Set up multi gpu training to speed things up. mostly used 2x H100.
- Tried estimating the GNS (simple) during training (average over the past N = 8, 16 steps, so not a true estimate but the measured value ended up stabilizing so i think it was good enough). Probably due to drift noise from the averaging, I didn't see the expected 1/B scaling - though I only tried 3 batch sizes, B = 64, 128, 256 - rather I saw a decrease by a factor of 2 over that range.
- I was mostly measuring the GNS for practice. Since I was only using 2 GPUs max, to even fit the modest batch sizes I was using I had to do many grad accum steps. So increasing the batch size came at the cost of wall clock time. But I wanted to get a sense of whether my batch sizes are large enough for stable training, and if they were maybe I'd try bumping up the learning rate (I was being cheap so I never did learning rate sweeps). So I thought if I've decreased the GNS by a factor of two, to where it was about 4, I can probably safely double the learning rate. Tried this to great success.

- Over 64 steps of training with B = 256, lr = 2e-6 on 2x H100 I had an increase from I think 52 percent to 65 percent pass@1.
- I would have liked to experiment with the GNS stuff more and see if I could find a better way to average.

2 Trying to find something to study

- While setting up the training I had decided I wanted to study something that got at the idea of "the dynamics of RL"
- I had seen some time ago this paper: https://arxiv.org/abs/2505.22617 on the "entropy mechanism of reinforcement learning" being discussed a bit on twitter. This seemed like an interesting topic to understand more so I started trying to understand their results
- The first part of their paper, about this empirically observed relationship between entropy and performance, didn't seem worth trying to investigate, since It would involve doing lots of long training runs. The second part of their paper, about trying to identify why the entropy decreases and giving evidence that some outlier tokens are responsible for much of the decrease, seemed like a better problem to focus on.
- I did some investigating of their logprob - advantage covariance criterion. They don't say this in the paper, but all the tokens with high logprob-advantage covariance have negative logprobs and negative advantage: unlikely tokens that appeared in wrong answers. Intuitively that makes sense - very unlikely tokens have an outsized contribution to the entropy, and punishing those tokens if they give the wrong answer would have an outsized contribution on the change in entropy.
- When I tried to understand their derivation I was pretty disappointed that it was basically just "inspiration", since it assumes a trivial "tabular softmax" policy.
- However it was pretty easy to see that a proper derivation suggested that their "logprob-advantage" covariance is probably part of a more interesting and richer story. Their nice empirical results from clipping these outlier tokens suggests that the tokens they identify are probably still a major source of the entropy collapse, but it seemed worthwhile investigating the proper derivation mor thoroughly
- If you're in the regime where the entropy change can safely be linearized in learning rate (later I confirm that holds for learning rates of interest) and you use SGD (or RMSprop - somethign with no momemntum so updates are coming from local gradients) then the entropy change has a conceptually simple form.

Let's define the Fisher Kernel for a pair of sequences t, t' as $K(t,t') = \sum_{params\ \alpha} \partial_\alpha \log \pi(t) \partial_\alpha \log \pi(t')$ (we can include preconditioning factors if we want)

This object is basically the effect of one sequence t' on another t. For a policy gradient update with SGD and batch B we have

$$\delta(\log \pi(t)) = \eta \sum_{t' \in B} K(t,t') A(t')$$

If we use Adam or another optimizer with momentum, the RHS will include a sum over contributions from past updates as well.

If we then use $\delta H = \delta E[-\log \pi] = -E[\log \pi \ \ \delta \log \pi]$ we get a simple formula for δH:

$$\delta H_U = -\sum_{t' \in U} E_{t \sim \pi}[(\log \pi(t) - b) K(t,t') A(t')]$$

where we can subtract a baseline.

If we focus on the $t = t'$ contributions, we get something like the results of https://arxiv.org/abs/2505.22617. This makes sense - sequences t used in an update will have a big effect on the probability for that same sequence. But clearly there's a lot of room for more than just those contributions.

3 Measuring entropy change

Even though the goal for this part of the project was pretty simple, it took me at least a couple weeks because there were so many things (theoretical and coding) that I had to learn in order to be able to do a proper precision measurement.

- My next goal was to test this linear order in learning rate formula for the step change in entropy. I had no idea whether it would be a good approximation for learning rates of interest, so I wanted to either confirm or deny that (maybe I'd have to use the quadratic term, or maybe we are just unlucky and the good learning rates for learning aren't good for these sort of approximations)

- I started by writing some code that would, during training, compute the linearized approximation for the entropy change (δH_1) at each step, and then compare it to the "true" change. But the "true change" that I was already measuring during training wasn't a super precise measurement - I was using different batches at each step to measure the entropy, so it wasnt an apples to apples comparison

- Another issue was that using the Adam optimizer that I used during training made computing the δH_1 a fair bit more complicated, unless I directly

computed the change in parameters of the model before and after an optimizer step. That would be fine for just measuring H_1 but later I wanted to study the Fisher kernel, which would require me to be able to pick apart the optimizer state. So it seemed like good practice to use the existing optimizer state and the current loss gradient to compute the parameter changes "by hand".

- I decided that the best way to proceed was to instead do my experiment "offline" rather than "online": I'd take a model checkpoint, saved along with its optimizer state, and test the entropy change formulas by using that checkpoint to get multiple data points, and then do that with multiple checkpoints.

- At this point I didn't know much about estimators besides "the variance of a monte carlo estimator goes like 1/N", and the fact that I could subtract a baseline sometimes. So my first approach to measuring the entropy changes, true and approximate, was not very sophisticated. My most naive first approach used a single batch B to compute the parameter changes and estimate the entropy before and after an update, as well as the entropy gradient. But of course this didn't make sense - in the double sum over t, t' the $t = t'$ terms add a bias and I was getting crazy looking results.

- The only "sophisticated" thing that I was doing at that point was using importance sampling to compute the entropy after the update, since I was using the same batch to measure the entropy before and after the update.

- I then learned about "U-statistics" and tried to use U statistic estimators for the entropy changes. Still got some pretty nonsense looking results though

- I then spent some time learning about more sophisticated techniques for variance reduction, like control variates and Rao-Blackwellization. The basic estimator for the entropy, $\sum_t -\log \pi(t)$ was maybe not the best because the $-\log \pi$ have a heavy right tail. At the batch sizes I was using, i was barely sampling the tail and getting some really high variance estimates of the entropy. One issue is that because I was insisting on using the same batch B for updates and estimating the entropy/entropy gradient, which had $G = 8$ responses per prompt, I wasn't getting as much bang for my buck out of my batch size, since the responses in the group were correlated.

- So going forward I decided that

  a) I would use separate batches for gradient updates and entropy/entropy gradient estimation, with the E (entropy) batch being sampled by uniformly sampling prompts with replacement from the test set (small enough that maybe the with-replacement mattered) and getting only one response per sampled prompt.

b) I'd use a fancier "Rao-Blackwellized" estimator for the entropy - $H = E[\sum_{tok\ k} H_k(toks_{<k}, prompt)]$ and the corresponding estimator for the gradient of the entropy

- I also decided to take this opportunity to completely redo my code for this portion of the project. Part of the motivation for this is that I was worried (due to my poor results so far) that the learning rates I was using were to high. To test what in what range of learning rates the linear approximation was valid I could either do a learning rate sweep or try and compute the ratio of the second order change in entropy to the first order. The second order term would require the hessian of the entropy. I learned about pytorch's JVP and functional call, and decided that functional call would be pretty useful. For example, if I compute the "update vector" (parameter change divided by learning rate), I could compute the entropy for the model at various learnign rates by calling it with paramters $\theta + \eta * v$.

- I wrote code to, given the model and optimizer states and an update batch, compute the update vector (for Adamw). I compared it with the true change in parameters, and found very good agreement - magnitudes of the update vectors agreed within a fraction of a percent, and cosine similarity was like 0.75 ... given that these are vectors in a 90 million dimensional space, that seemed pretty good.

- I then used the functional call to compute the entropy for a set of learning rates. However, I got some pretty nonsensical results. It took me a while to debug this. I think the biggest issues were:

  a) numerical precision

  b) not turning off nondeterministic things in the model

  c) some issues with not restricting to lora parameters only

  The numerical precision thing ended up being somewhat frustrating because for the learning rates I was sweeping over I was running into numerical precision issues and could not do better - I couldn't get the model to run in fp64. BUT I was using unnecessarily small learning rates, like 1e-10 - 1e-8. For 1e-6 - 1e-4 (the lower end of this is what I used for training), then not only was the numerical precision I could run seemingly sufficient, but the entropy change was very close to linear in the learning rate! At around 1e-4, the ESS for my importance sampling started to collapse.

- So it seems I got a bit lucky: With the learning rates of most interest, I am just above the numerical precision noise threshold and comfortably in the linear-in-learning-rate regime.