Analysis

How to turn SupAmp into ReAmp?

Richard Möhn

2019-09-12

Contents

1 Overview

(For a project overview and a glossary, see the home page of the Farlamp repository.) This is a design document that I mostly wrote for myself in order to clarify where I have to go. I will update it on the way to implementation.

Before I can experiment with overseer failure, I need to adapt the supervised learning system from (Christiano et al., 2018) to reinforcement learning and evaluate the result on the original tasks from the article. I see three major areas of work: overhaul the learning mechanism, find a good way to determine rewards, and update *H* to return evaluations instead of answers.

In the following I assume that you are familiar with (Christiano et al., 2018). OQ stands for *open question*. I also recommend you read Overseer failures in SupAmp and

ReAmp first, even though it's about the second part of the project. It's more polished and has a proper introduction to the problem. And it uses more consistent notation, distinguishing between problems, solutions, questions and answers.

2 Major adaptations

2.1 Adapting the mechanism

In the following three points I paraphrase (Christiano et al., 2018, 2.2), with adaptations for RL. Note that here we run only three processes in parallel, while in the original there are four. This is because the original trains X with supervised learning (SL), where we can feed samples to the learner in one direction and asynchronously. Here we need two-way synchronous communication between the learner and the overseer, as X generates a sample answer, gives it to $\text{Amplify}^{H'}(X)$ for evaluation and gets back a reward that it learns from.

1. Sample a question $q \sim D$ and pose it to X. X returns answer a. Use $\text{Amplify}^{H}(X)$ to evaluate a and calculate a reward r for X. Record the whole interaction, ie. $(q, a, \langle\text{sub-questions and -answers}\rangle, r)$. The number of sub-questions H asks is fixed.
2. Using a recording from process 1, train H' to predict the outputs of H.
3. Sample a question $q \sim D$ and pose it to X. X returns answer a. Feed q and a to $\text{Amplify}^{H'}(X)$ to generate a reward r. Train X on r using reinforcement learning.

In SupAmp the learning process builds on questions that H can answer without asking X sub-questions. Here it builds on question/answer pairs that H can *evaluate* without asking X sub-questions. (See (Stuhlmüller, 2019).) H can always ask primitive questions, because they don't depend on X.

- If the number l of sub-questions is fixed, does H ask blank sub-questions if it has fewer than l actual sub-questions? OQ
- Does H always decompose a question? Or does it sometimes evaluate higher-level questions directly as well? Eg. 'What is $\sigma^2(5)$?' OQ
- How does pretraining work? How do I have to adapt it? OQ
- In process 3, is it useful to take one question and do multiple rounds of answer-evaluation? OQ
- Read what Paul has written about RL-based IDA. TODO
- X doesn't have a way to feed sub-questions to itself. This is because the authors of (Christiano et al., 2018) want it to come up with structures more efficient than recursive questions. But this limits capability, doesn't it? I suspect that at some point feeding questions to oneself is necessary to become more intelligent. Then the question becomes: when and when not to decompose? See (Christiano et al., 2018, 6 'Recursive model architectures') for going deeper into this. OQ

2.2 Determining the rewards

This is not clear to me yet. It looks like in some tasks, such as permutation powering, we can only give reward 1 for a right answer and reward 0 for a wrong answer. On other tasks the reward might be between 1 and 0, depending on how close X's response was to the correct answer. For example, for shortest path. In these cases the evaluation scripts might become more complicated, though. OQ

- Choosing between two answers as in (Stuhlmüller, 2019) wouldn't work well here. Or would it? Why do they do it? OQ
- Get a general idea about how rewards are determined in RL. TODO
- Read again about 'reward engineering' (Christiano, 2016). TODO

2.3 Adapting the H scripts and questions

This is almost the same as in SupAmp, except that X now suggests an answer to the top-level question and H gives it a reward depending on how close the suggested answer is to the truth. In order to find the truth, H asks sub-questions.

- Is this really so? – Sketch how to adapt all the scripts to evaluation. TODO

2.3.1 Permutation powering

To the task description in (Christiano et al., 2018, app. C) add: 'Evaluation: If the suggested root answer equals the answer to the second sub-question, return $r = 1$, otherwise $r = 0$.'

What is the greatest power that we expect H to figure without asking non-primitive sub-questions? This goes back to an open question in section 2.1. OQ

Example 2.1. Given this permutation:

n	0	1	2	3	4	5	6	7
$\sigma(n)$	1	0	6	2	3	5	4	7

q_1: What is $\sigma^{28}(4)$?
a_1: 7 (X's suggested answer)

H's sub-questions and X's sub-answers:

$q_{1.1}$: What is $\sigma^{14}(4)$?
$a_{1.1}$: 2

$q_{1.2}$: What is $\sigma^{14}(2)$?
$a_{1.2}$: 4

Following the evaluation prescription above, H should give reward 0. But how does H know whether a sub-answer is correct? If I understand (Christiano et al., 2018) correctly, the system could receive a question like the one above in the very first run. Yes, the training starts with easy tasks, but in permutation powering this limits the size N of the domain, not the power k. OQ

Is it that X returns gibberish in the beginning and not numbers and thus H knows that X hasn't been trained enough yet? It might work with SL, because when the learner returns gibberish, we give it some labels that are numbers and eventually it starts to return numbers. However, with RL, H would have to reward gibberish that looks like it's getting closer to a number until it reliably gets numbers. I doubt that this is what we want. Instead, we have to restrict X to only returning numbers. Which is why the suggestion at the top of this paragraph wouldn't work with RL.

Or does H blindly rely on the sub-answers and the occasional $k < 4$ ensures that the training slowly moves in the right direction? I don't think this would work, either.

Maybe when X is in execution mode and unsure about the correct answer, it can return 'don't know'. Only in learning mode does it propose numbers, because its exploration parameter is turned up. Looks like I have to brush up on my reinforcement learning knowledge.

Question: Could we get a better training signal if I ask for all the intermediate results and make the reward the number of correct mappings? Answer: Maybe, but then I prevent the learning of more efficient representations.

This task would work well for early experiments: start with small N and limit the range of k.

3 Experiments

In this part of the project I'm not testing a hypothesis, but trying to turn SupAmp into an RL-based system (ReAmp) with the same performance. In order to judge success, I need to compare ReAmp with SupAmp by measuring accuracy over number of training data for X, as (Christiano et al., 2018) do. Then I might extend the comparison with another analogy to the article: an RL agent trained with ground truth data.

4 Efficiency

- Can I make sampling more effective by preferring questions that H has asked as sub-questions before? – I could try this with SupAmp. OQ
- Is it efficient to start training X and H' at the same time? Wouldn't it be better to wait until H' is reasonably accurate in predicting the initial behaviour of H? And pause training of X whenever the accuracy of H' drops below a threshold. OQ

These are points that came to my mind. Turning SupAmp into ReAmp doesn't depend on them.

5 Required learning

I will have to learn a lot in order to understand the architecture described in (Christiano et al., 2018, 2.5). TODO

- autoregressive models (not sure if this is required with RL)
- Transformer architecture (encoder-decoder, self-attention)
- embeddings and linear projection
- more about neural nets in general
- pointer networks
- more about reinforcement learning

References

Christiano, P. (2016). The reward engineering problem. Available at: https://www.alignmentforum.org/s/EmDuGeRw749sD3GKd/p/4nZRzoGTqg8xy5rr8.

Christiano, P., Shlegeris, B., and Amodei, D. (2018). Supervising strong learners by amplifying weak experts. Available at: http://arxiv.org/abs/1810.08575.

Stuhlmüller, A. (2019). Delegating open-ended cognitive work. Available at: https://ought.org/presentations/delegating-cognitive-work-2019-06.