CARL: Cloud Assisted Reinforcement Learning

Abdullah Bin Faisal

1 Project Overview

Reinforcement Learning (RL) is becoming increasingly important for next generation applications (e.g., self-driving cars [5]). While its benefits in dealing with problems requiring dynamic control under stochastic settings has been shown, there exists a lot of ground to cover before RL can become the defacto tool for the industry. It has been a center of criticism from leading machine learning experts due to its hunger for data while computational requirements of RL algorithms stifle their chances of deployment.

With the cloud now offering scalable and inexpensive compute and storage power ([1]), there is hope for applications dependent on RL to see deployment success. Additionally, there has been a wave of new computer architecture, custom tailored to the requirements of machine learning applications (e.e., TPUs [2]). Other efforts include creating cloud based systems that can provide useful abstractions to programmers interested in using machine learning for their applications (e.g., Tensorflow [4], Keras [3]). However, offloading computation and storage to the cloud does not come for free: the fedility of the network in between the RL agent and the backend system running at the cloud becomes the key. If the network is poor, then the performance of an RL agent may degrade. This is especially true for thin RL agents designed to rely on the cloud and expend most of their efforts in collecting and reporting sensory data.

In this project, we aim to look at the performance of specific RL algorithms with computationally intensive code-paths offloaded to the cloud. In particular, our focus will be on learning and planning methods, with the planning phase offloaded to the cloud. By simulating varying level of network fidelity, we aim to understand:

- How does the computation proefficiency of the cloud speed-up performance?
- How does an impaired (slow/lossy) network impact inversely affect performance?
- What algorithmic modifications can be made to make RL agents more robust to such impairments?

Algorithm 1 Dyna-Q

```
1: procedure ComputeThresholds(FlowList $F$)
2:     Do Forever:
3:     $S \leftarrow currentstate$
4:     $A \leftarrow \epsilon$-greedy$(S, Q)$
5:     Execute $A$; observe reward $R$ and state $S'$
6:     $Q \leftarrow Q + \alpha \times [error]$
7:     Update model $M$ with updates $U$
8:     $send(U, cloud)$
9:     $get(Q, cloud)$
10: end procedure
```

2 Background and Related Work

Cloud based applications have been around for years now. A broad category of current and next-generation applications are deemed to be reliant on inference (ML) and sequential decision making (RL) systems. With the cloud, the performance bottleneck has shifted from computation and memory to the network which connects the end-points (RL agents) with the cloud. While the power of the cloud is well understood, the question of building algorithmic robustness within RL agents to work well over a variety of networks has been largely unexplored.

While cross-layer optimizations (enriching the interface between appications and the network stack) have existed for a while in the mobile community for traditional applications such as web search and audio/video streaming, no such parallel exists for applications reliant on ML and RL. These techniques work on adapting application logic based on network feedback. For instance, graceful degradation of a mobile application on a clumsy network. This is analogous to the agenda of this project; adapting RL algorithms based on network feedback.

Complementary to the agenda of this project, RL has been extensively used in network engineering. However, these efforts have largely focused on how RL can be used to solve network resource management problems. Instead, we take a look at the problem from the other end; how can we use network feedback to intelligently manage RL algorithms.

3 Problem Formulation and Technical Approach

Chapter 8 of the book focused on RL in the wild - paying heed to some of the real world constraints. It also discussed augmenting real experience that an agent goes through with planning, which is taking what has been experienced so far and running simulations on that model. This has been shown to speed up convergence to the optimal. The chapter discusses that planning can be a computationally intensive process, especially if the state-space is large.
Given the computation requirements of the planning step, one could offload

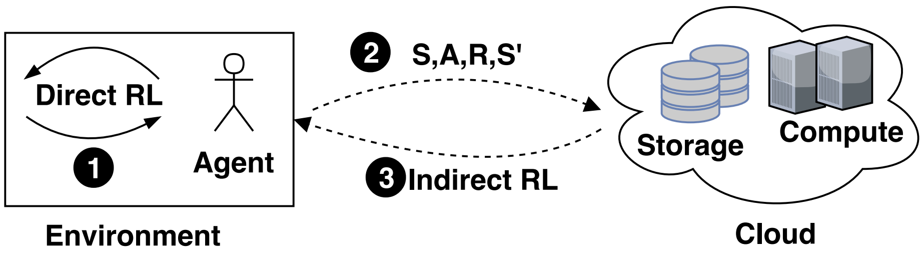

Figure 1: Harnessing the power of the cloud, by outsourcing planning.

it to the cloud. Figure 1 shows the high level setup of such a system. The agent interacts with the environment and maintains the necessary state to do direct reinforcement learning, possibly via TD or MC learning. It then sends the relevent observations (Current state, Action taken, Reward obtained, and Next state) to a cloud setup over the network. The cloud concatenates the new information with the previous model already maintained and then runs computation on it (simulated experience). After this, it responds with relevent updates (potentially an updated Q-function) to the agent, enabling it to do indirect reinforcement learning.

While this strategy allows to harness the power of the cloud, it fundamentally hinges on the fidelity of the network that connects the agent to the cloud. If the network is corrupt/lossy/slow, it can affect the convergence (to the optimal) time of the agent - which may not be desirable in certain cases.

3.1 Potential solution

In my project, I would like to delve deeper into planning and learning methods over such a framework and focus on algorithmic durability; how can we make learning methods robust in such settings?

Algorithm 1 shows the Dyna-Q process implemented using the cloud. Line 8, sends the model updates to the cloud while Line 9 receives the updates Q-function. Notice that these two steps require the assistance of the cloud and jointly make the planning phase of the Dyna-Q process.

I have identified two potential scenarios where this algorithm can be adapted based on network feedback:

Adapting greediness. Suppose the agent identifies that the network is causing large delays in transmitting new experience to the cloud. Instead of relying on planning to improve convergence, it could try exploring more states. Therefore once the network is less-busy again, more of the state-space would have been explored and the resumption of the planning module running at the cloud could provide the most useful updates.

Prioritized exploration. If the network bandwidth to the cloud is limited, the agent could perform some sort of prioritized exploration so that only the most useful observations are transmitted over the network.

Hypothesis. The hypothesis is that under normal settings, being less greedy or prioritizing would infact hurt performance (lowering cumulative reward). However, under constrained settings, the agent would willingly make this compromise for overall better performance (faster convergence).

4 Evaluation and Expected Outcomes

The evaluation and experimental setup consists of the following steps:

1. Setting up the distributed Dyna-Q algorithm (Alg.1)
2. Simulating a delayed network. This can be done by probabilistically skipping the planning step. This models that the network is busy and updates cannot to pushed to/pulled from it.
3. Simulating a lossy network. This can be done by capping the maximum updates that can be pushed to the cloud.
4. Adapting greediness and prioritizing updates based on the level of network loss and delay. The intuition for this step has been described in §3.

We will test the cloud assisted network adaptive version of Dyna-Q versus standard Dyna-Q on a variety of small and large grids. Very similar to how it has been done in chapted 8 for various versions of the Dyna-Q algorithm.

References

[1] Amazon ec2.

[2] Cloud tpus - ml accelerators for tensorflow — cloud tpu — google cloud.

[3] Keras: The python deep learning library.

[4] Tensorflow.

[5] Pranav Dar and Analytics Vidhya. An autonomous car learned how to drive itself in 20 minutes using reinforcement learning, Jul 2018.