CARL: Cloud Assisted Reinforcement Learning

Abdullah Bin Faisal

Abstract

Emerging AI applications like self-driving cars and warehouse robots continuously interact with their environment to learn policies in order to meet meta-objectives. These applications impose processing and storage requirements. In this work, we posit the use of the cloud to host performance critical learning algorithms for such agents. We propose CARL, a framework, that decouples learning policies from an agent's interaction with the environment; delegating the latter to the cloud while retaining the former at the agent. To kickstart the discussion, we propose a first cut programming interface that shows support for a variety of reinforcement learning algorithms. We place key emphasis on algorithmic durability to network impairments and show that by making hyper-parameters (e.g., greediness) of an agent adaptive to network conditions, CARL can work well over challenging scenarios where the network is a bottleneck (e.g., lossy, slow). Our preliminary results show that CARL can maintain good performance in face of network impairments.

1 Introduction

For the past several years, the extensive use of data coupled with supervised and unsupervised Machine Learning (ML) has become ubiquitous. This interest in using big-data to aid decision making has catalysed innovation across multiple fields: i) Programming abstractions ([3, 8, 14, 6, 2]) and ii) Hardware ([12, 13]). There also has been prior work ([4]) aimed at harnessing the power of the cloud to support complex ML systems, with a strong focus on the performance requirements imposed by these systems (e.g., low latency, high computational prowess).

Even more recently, the incapability of typical ML approaches in handling stochastic and unpredictable environments has been highlighted - where in order to reach a target, an agent must perform a long sequence of actions and there is no notion of a "label" as in ML [9]. Towards this end, Reinforcement Learning (RL) has shown promise and 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.

RL applications typically have three main components: i) interaction with the environment, ii) following a policy, and iii) policy evaluation and improvement. Interacting with the environment requires agents to often make mechanical movements to navigate in the state-space and gather sensory data such as a video stream of its surroundings. Following a policy enables the agent to decide what actions to take based on the state of the environment. Lastly, policy evaluation and improvement is a cyclical process of running extensive simulations to generate a variety of trajectories and using them to learn and improve the policy.

These components translate to different system level requirements. For instance, interacting with the environment imposes a need for mechanical robustness and elongated battery life. Following a policy is the equivalent of "inference serving" in the ML world, and often requires low latency while policy improvement and evaluation requires storage and processing prowess.

With the cloud now offering scalable and inexpensive compute and storage power ([1, 11]), there is hope for applications dependent on RL to see deployment success. However, existing cloud-based learning frameworks ([6, 14, 7, 4]) are optimized for big-data workloads or for supervised learning problems where there is little communication between an agent and the cloud. On the contrary, RL agents will need to periodically communicate with the cloud and thus the network becomes a *critical* resource.

We argue that offloading computation and storage to the cloud does not come for free: the fidelity of the network in between the RL agent and the backend RL framework running at the cloud becomes the key. If the network is poor, then the performance of an RL agent can degrade. This is especially true for thin RL agents designed to rely almost completely on the cloud and expend most of their efforts in collecting and reporting sensory data.

In this work, we look at the performance of a specific RL agent with a computationally intensive code-path offloaded to the cloud - we call this system CARL. In particular, our focus will be on learning and planning methods, with the planning phase offloaded to the cloud.

In total, we make the following contributions:

- We propose a first-cut programming interface that shows promise to support a variety of RL algorithms in a cloud assisted framework (§4). We show a specific instance of this interface in the context of planning and learning methods.
- We show that by adapting the level of greediness of an RL agent based on hints from the network (loss rate, delay), CARL can work well over a challenged network (§6.3).

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

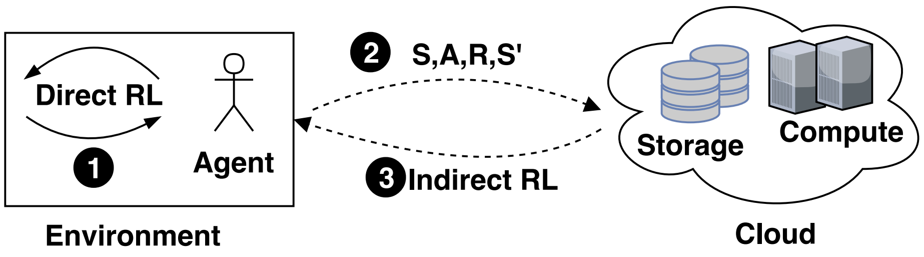


Figure 1: Harnessing the power of the cloud, by outsourcing planning.

(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 applications 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 Motivation

We begin by considering the basic requirements of an RL algorithm. There are three main steps involved. An agent repeatedly *interacts* with the environment, and makes important observations about the state and reward. It uses these to *update* its version of the model of the environment. Extensive *simulations* can then be run on the model to learn a policy and improve it.

Thus, CARL must provide efficient infrastructure for *interactions*, *updates*, and *simulations*; next we describe these.

Interaction typically requires the agent to navigate through the environment. As part of this interaction, the agent collects observations about the environment. For instance, its GPS location. Additionally, this interaction also collects the rewards present in the environment. This step often requires mechanical movements and the cost per interaction is high.

Updates transform the observations to a model. In RL, this is typically represented as a Markov-chain. Additionally, depending on the exact learning algorithm, the manifestation of the model maybe a table containing state-action values or a neural network that approximates such a table.

Simulations are then run on the model to learn a policy or improve it further. This is critical as most RL applications are not sample efficient and their hunger for data requires them to reuse the modest set of observations in an intelligent fashion.

We take the specific case of learning and planning methods and show how these requirements are orchestrated under CARL. We see in Figure 1, that the first step is the *interaction* step where an agent interfaces with the environment by exploring and exploiting it (direct RL). In the second step, the agent offloads the environment observations (State, Action, Next state, Reward) to the cloud, allowing the cloud to *update* the existing model. The third step involves the cloud running multiple planning passes (*simulations* in general) to improve and evaluate the policy. The policy is then pushed to the agent so it can perform indirect RL.

Notice that the agent maintains very little state and the local policy generated by doing direct RL is not sufficient alone and is very basic. The bulk of the computation and storage is offloaded to the cloud. The cloud then runs the computationally intensive planning step using its processing prowess and returns updates to the agent which it can use to improve its local policy (indirect RL).

Since the network conditions between the agent and the cloud play a crucial role in the performance of the RL task, we induce network awareness in the modules running on the agent. In particular, when the network reports to be degraded (lossy, slow), the agent adapts its greediness in exploring the environment and prioritizing the updates that need to be sent to the cloud. We elaborate more on this in §5.

Summary. By decoupling decisions: *interactions* from *simulations* and designing a suitable interface to support *updates*, we can satisfy the requirements of each module. By creating network awareness in RL applications running on agnets, we can also maintain good performance on heterogeneous networks.

4 Programming interface

To kick start a discussion on a cloud assisted reinforcement learning framework, as in CARL, we present the following simple and flexible API that can be used by RL agents to "talk" to the cloud running RL algorithms (e.g., Q-learning, SARSA):

```
updateModel(State,Action,Next State,Reward)->(context)
syncPolicy()->(Policy)
egreedy(State,Epsilon)->(Action)
batchObservations(State,Action,Next State,Reward,criteria)->(context)
```

Algorithm 1 CARL enabled Dyna-Q

```
1: Do Forever:
2: $S \leftarrow currentstate$
3: $A \leftarrow$ cloud.egreedy$(S, E)$
4: Execute $A$; Observe reward $R$ and State $S'$
5: $Q \leftarrow Q + \alpha \times [error]$                      ▷ direct RL
6: cloud.updateModel$(S, A, S', R)$
7: if S == Terminal State then
8:     $Q \leftarrow$ cloud.syncPolicy()     ▷ Download Q values from cloud
```

The API has four functions. The first enables the agent to offload environment observations to the cloud and depending on the exact implementation, the returned context could denote whether the update was successful. The second provisions for the case where the agent wants to inherit the superior policy learned by the cloud. This is useful in episodic tasks, where the agent can expend some time "downloading" the policy from the cloud at the end of each episode and the beginning of the next. The third function is to invoke a cloud hosted epsilon-greedy policy, where the greediness can be specified by the agent and the cloud will return the appropriate action based on the policy it has currently learned. The fourth is to enable batched updates to be sent to the cloud. The inputs are the observations and a criteria which denotes some condition that the agent desires should be met before the updates are pushed to the cloud. For example, the criteria could be a timer which when expires, triggers the updates to be automatically dispatched to the cloud.

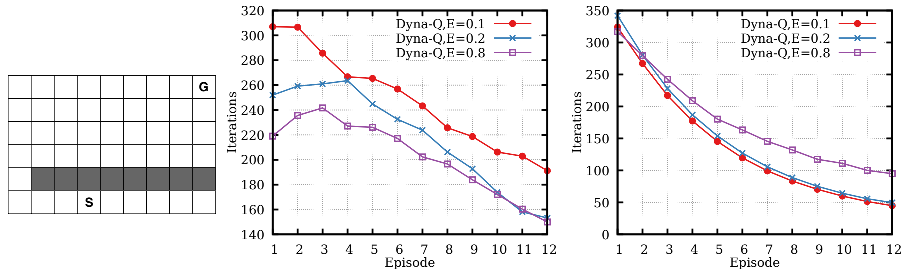
Our choice of starting with this simple interface is motivated by the consideration that often, the agent will not be able to host complicated state-action tables and powerful RL algorithms. Thus a suitable way of offloading observations to the cloud is a key requirement. Similarly, the need to have a flexible way to batch updates is important because the agent may not be able to send updates as they become available, perhaps because it doesn't temporarily have internet connectivity.

We now show how this interface can be used by an RL application. Algorithm 1 shows the Dyna-Q algorithm running at the agent connected to the cloud. We assume that the agent is capable of direct RL however it doesn't execute the planning step which is computationally intensive. The agent offloads environment observations using the `cloud.updateModel()` function call. It also updates its local Q table, as is necessary to do direct RL. The agent however doesn't use the local Q table and in this code snippet, it always invokes the `cloud.egreedy()` method which returns the best action to take. Lastly, once an episode is completed, the agent "downloads" the clouds version of the Q table using `cloud.syncPolicy()`. Here, policy loosely refers to the information base (Q-table) used to come up with actions.

5 A case for network awareness in CARL

Given the emphasis of harnessing the power of the cloud to improve the performance of RL applications, special attention needs to be given to the network that connects the agent to the cloud. Network impairments, like outages and loss can have performance implications and if not handled, may fizzle out the benefits of using CARL. For example if the network between the agent and the cloud is temporarily congested, then the `cloud.egreedy()` call may take a long time to be executed, increasing the latency of completing the learning task.

Inspired by cross-layer optimizations, especially in the mobile community, we investigate the possibility of instilling network awareness in RL algorithms; making them more durable to impairments like outages, delays and losses. More concretely, we see if the choice of exploration-exploitation can be viewed as a function of network hints. Additionally, we see if prioritized sweeping - a technique used to prune state updates that have low utility - can be extended to judiciously select updates that need to be sent to the cloud if the network cannot handle all of them. We now elaborate both these strategies in the context of planning and learning methods.



(a) Dyna Maze (b) No planning (c) 50 planning steps

Figure 2: Motivational experiment showing that exploration can be made a function of network conditions. The agent runs Dyna-Q on the maze with and without planning (indirect RL). Learning curves are plotted as a function of the first 12 steps. Number of iterations to reach the goal state are on the vertical axis, while episodes elapsed are on the horizontal axis. Three curves of different exploration rates (E) are plotted. We see that E=0.8 is best when planning is unavailable while it is poor when the agent can perform planning.

Adapting greediness. Suppose the agent receives the feedback 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 which the cloud can use to enhance the realism of the model it hosts.

We show why this is a good idea on a simple maze solving task, using the Dyna-Q algorithm. For this experiment, the Dyna-Q algorithm runs on a single machine, and follows the exact semantics as described in chapter 8 of [9]. Figure 2 shows how the Dyna-Q learning agent performs on the maze solving task (2a) with no planning (2b) and 50 steps of planning in each iteration (2c). We can see that planning improves performance however the interesting observation is that a lower exploration rate ($\epsilon = 0.1$) performs best when the agent can perform planning steps but a higher exploration rate ($\epsilon = 0.8$) is best if the agent cannot perform planning. In CARL, we envision that planning will be offloaded to the cloud, hence in epochs where the network to the cloud is degraded, the agent can use a higher exploration rate on its local state.

Prioritized offloading. If the network to the cloud is congested and only a few updates can be pushed to the cloud, the agent could prioritize among the set of all updates that need to be sent so that only the most useful observations are transmitted over the network. The process of fine tuning the threshold that defines which updates are high priority and low priority can be made network aware.

6 Evaluation

Our preliminary evaluation aims to verify that indeed the programming interface works, can support CARL enabled Dyna-Q, and show promise of network aware RL algorithms. We first describe our setup then move onto the key results.

6.1 Setup

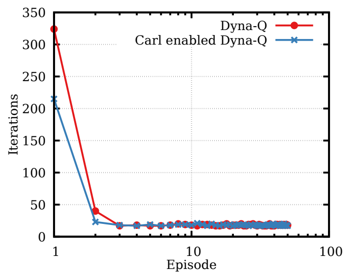
For our evaluation we use a python based prototype which uses RPC calls. We use two d430 nodes with 10Gbps network interfaces on the Emulab testbed [10]. One node runs the agent code while the other runs in server mode. We assume that network hints can be communicated to the agents running the thin RL application.

Network outage. To model an impaired network, we induce an emulated network outage at strategic points during task learning. During the network outage, the agent cannot use the programming interface except the `batchObservations()` call; the criteria is set such that updates are pushed to the server whenever the emulated outage ends (see §4).

Lossy network. To model a loss, we cap the maximum number of observations that can be offloaded to the cloud at a given time. This is configurable in the code script.

Task and metrics. The agent attempts to solve the Dyna-Q maze (§2a). It follows Algorithm 1. We are interested in the first few episodes of the task where most of the learning happens. Thus our primary metric is the learning curve.

Parameters. The agent running the CARL enabled Dyna-Q algorithm uses the following parameters: $\epsilon_{cloud} = 0.1, \alpha = 1, \gamma = 0.95$. Additionally, it uses a network dependent local exploration rate: ϵ_{local}.



(a) Baseline

Figure 3: Comparison of learning curves for the CARL enabled version of Dyna-Q (§1) and the traditional implementation. Both algorithms have similar learning curves.

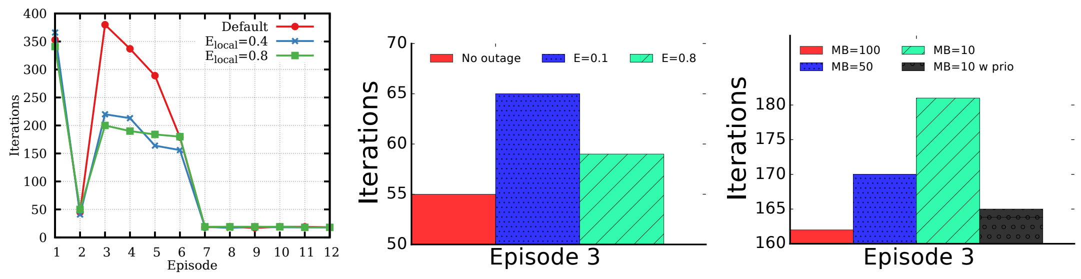
6.2 Comparison with baseline

We compare CARL enabled version of Dyna-Q versus the traditional implementation that runs on a single node. Figure 3 shows the learning curves for both implementations. While both implementations have similar learning curves and converge to the the optimal policy (17 iterations), we observe that CARL enabled Dyna-Q takes 2× more time. However, we posit that on bigger and complex tasks, computation will become the bottleneck and CARL will be able to offer lower mean task completion times as well.

6.3 CARL over challenged networks

We evaluate two network impairments: outages and losses. We induce outages by disconnecting the agent from the cloud for episodes, or within an episode for several iterations. For loss, we cap the maximum number of updates (Max Burst) that can be pushed to the cloud.

Outages lasting episodes. Figure 4a shows the learning curves when an outage is induced at the third training episode and lasts four episodes. During this time the agent can only rely on direct RL. We can see that having a higher exploration rate ($\epsilon_{local} = 0.4, 0.8$) during the network outage leads to the maze being solved in fewer iterations. We also see that as soon as the network outage



(a) Multi-episode outages (b) Intra episode outages (c) Intra episode loss

Figure 4: Performance of different strategies to cope with network impairments (outages, loss).

ends, the agent resumes using the `cloud.egreedy(S,0.1)`. Another insight is that if the length of the outage can be determined apriori, then modulating the exploration rate based on this could prove useful. As we can see that $\epsilon_{local} = 0.4$ slightly improves over $\epsilon_{local} = 0.8$ for the last two episodes of the network outage.

Intra-episode outages. Figure 4b shows the number of iterations it takes the agent to solve the maze when the cloud is not available for 30 iterations during the third episode. This is different from the previous case where the cloud is unavailable for 4 episodes. We can see that when there is no outage, the CARL enabled Dyna-Q agent can solve the maze in ≈ 55 iterations. However, for the case when the network outage is reported to the agent, a higher exploration rate, $\epsilon_{local} = 0.8$, can result in better performance (59 iterations) versus the case when nothing is done i.e. the agent continues using $\epsilon_{local} = \epsilon_{cloud} = 0.1$ (65 iterations).

Intra-episode loss. Figure 4c shows the number of iterations it takes the agent to solve the maze under the following induced impairments: a 200 iteration network outage during episode 3, followed by a limited number of updates that can be pushed to the cloud. While the network is unavailable, the agent uses $\epsilon_{local} = 0.1$. This is to rule out any gains due to the higher exploration as discussed in the previous case. Instead, the agent relies on the `batchObservations()` API and prioritizing updates. We see that if 100 observations are allowed to go through the network, the agent can solve the maze in 162 iterations at episode 3. As the Maximum Burst (MB) is reduced, the iterations required to solve the maze increases. With an MB limited to 10 (only 10 observations out of the 200 are allowed to be offloaded to the cloud), the agent takes 20 more iterations to solve the maze! However, by using prioritized sweeping even with MB=10, the agent can improve its performance and achieve comparable performance to the case when MB=100.

7 Conclusion

We presented a case for Cloud Assisted Reinforcement Learning to facilitate the deployment of mission critical AI applications like self-driving cars and warehouse robots. We distilled key requirements that such applications have: i) interacting with the environment, ii) updating model and iii) learning a policy. By delegating these to the agents and the cloud respectively, and designing a suitable programming interface to support model updates, we showed that a cloud assisted performance efficient framework could be designed, as in CARL. We addressed the issue when the network becomes the bottleneck and showed that RL algorithms could intelligently use feedback about the networks state to maintain good performance. Our evaluation results showed promise of our first-cut interface and network awareness in RL algorithms.

References

[1] Amazon ec2.

[2] Keras: The python deep learning library.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.

[4] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. 2017.

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

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. January 2008.

[7] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 2007.

[8] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 2012.

[9] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[10] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. pages 255–270, Boston, MA, December 2002.

[11] Wikipedia contributors. Google cloud platform — Wikipedia, the free encyclopedia.

[12] Wikipedia contributors. Graphics processing unit — Wikipedia, the free encyclopedia.

[13] Wikipedia contributors. Tensor processing unit — Wikipedia, the free encyclopedia.

[14] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 2016.