CS7610 Final Project: Decentralized Reinforcement Learning

David Klee and Luke Roberto

December 5, 2019

1 Introduction

Reinforcement learning (RL) can be a powerful machine learning tool for problems where supervised training data is unavailable. However, like supervised learning, a deep RL system requires a multitude of data points in order to refine its model. For deep reinforcement learning, there are two main computational burdens that arise when training on complex problems. First, in order to generate a reliable policy, the model must be trained on a set of experience that has high coverage of the state space. However, for expansive or continuous state spaces, this can mean the costly simulation of billions of transitions in complicated, high-fidelity physics engines. Second, complex domains require deep neural networks to estimate the value of states in the environment (up to 100 convolutional layers for some visually demanding tasks). Thus, the back-propagation step to optimize the network is time-consuming even for specialized hardware.

To accelerate learning rates, distributing computational load has proved necessary for difficult problems. While most existing approaches use a centralized approach for learning, in this work we explore a decentralized method. Decentralized learning has applications in cases with large latency between training agents and the central server, or when the experience for training is private and cannot be sent to a central server. In this paper, we explore a diffusion-based method from the paper, "Diff-DAC: Distributed Actor-Critic for Average Multitask Deep Reinforcement Learning (2017)" [1]. We compare the decentralized approach to a centralized method and investigate the fault tolerance of the method to fail stops and network partitions.

2 Methods

In this section, we will discuss what we implemented in order to evaluate the performance of diffusion-based decentralized deep reinforcement learning from [1]. First, we created an environment with which to evaluate reinforcement learning rates. Then, we formulated a deep q-network that was compatible with the state

and action space of the environment and had sufficient parameters to effectively estimate the q-function. We discuss how we enabled distributed communications for parallel learning then present a centralized RL approach to be used as a comparison. Finally, we detail the implementation of the decentralized, diffusion-based learning.

2.1 Environment

We created a grid world environment for testing. This is a common environment for our own research as a test bed for potential learning algorithms or neural architectures. The grid world is a 2D space where the agent observes its x, y-position and can move horizontally, vertically, or diagonally (8 movements total). Having a discrete action-space simplifies the neural network models needed to learn an effective policy. The agent receives a reward of 1 when it is within a threshold of the goal position. For learning, the agent is spawned at a random position for the start of each episode. One added benefit of this environment is that the difficulty of the problem can be easily modified by changing the size of the grid. For our work, we set the size of the grid such that the agent must take 30 movements in order to transverse the grid.

2.2 Deep Q-Network

As stated above, we choose the grid world environment so that we could use a simpler RL method called deep q-learning. In deep q-learning there is a neural network (called a deep q-network or DQN) that takes observations from the environment as input and produces an estimate for each action available to the agent in that state. In other words, the DQN can be viewed as a function of state, s, and action, a: $q_\theta(s, a)$, where θ represents the network weights. Given a set of experience $\{s, a, s', r\}$, we can optimize the neural network with the following loss function:

$$l(\theta) = \|q_\theta(s, a) - (r + \gamma \max_{a'} q_\theta(s', a'))\| \tag{1}$$

To stabilize learning, we use a replay buffer of 100k transitions to make the sampled batches for gradient descent more independently distributed. Additionally, we use a target network to estimate the value of the next state in Eg. 1, which is lagged behind the main network by 30 episodes. A value of 0.98 was used for γ in all tests.

2.3 Distributed Communications

For distributed processing and communication, we chose to use a python library called Ray [2]. Ray is a "high-performance distributed execution framework targeted at large-scale machine learning and reinforcement learning applications". This provided a simple, abstract API for use to easily distributed out DQN code to many cores/machines. The example code given in Fig. 1.

```python
@ray.remote
class Agent():
    def run(self):
        return 1

# create remote agent
agent = Agent.remote()

# run method
p_id = agent.run.remote()

# wait till run is complete (BLOCKING)
ready = ray.wait(p_id, timeout=1)

# access the output
result = ray.get(ready)
```

Figure 1: Running methods in parallel with Ray

In ray there is a notion of an "actor" which is a remote class object. Creating an instance of a class remotely generates a parallel process that includes its own block of memory. Any calls of functions to this parallel process will automatically queue them to be run sequentially, in a non-blocking manner. You can then use the wait/get functions to actually block the program to receive a list of futures given to you before. We use this to train all of our agents in parallel and also for diffusion in the decentralized agent system.

2.4 Centralized

A common method for increasing learning rates in RL is to use a centralized replay buffer and model. A collection of runners, each with a copy of the environment, gather experience to send to a central agent. The central agent collects the experience in a replay buffer and samples from the buffer to optimize its deep q-network. This approach is especially useful in cases where the state-space takes a long to fully explore. We used this approach as a comparison to evaluate the learning rate of the decentralized approach.

2.5 Decentralized

The decentralized approach we tested is based on the Diff-DAC paper [1] (the main difference is we use a DQN instead of actor-critic). In this method, there are agents each with a set of neighbors, arranged in a network. Every agent iterates through the following three steps: (1) gather experience in the environment, (2) use experience to optimize deep q-network, (3) exchange network weights with neighbors and update to the average. The final step is called diffusion, which is where the method gets its name.

3 Results

There were several tests we ran on the system in order to benchmark performance and robustness to different types of network stresses. We detail our results below:

3.1 Comparison of Distributed Approaches

We first wanted to generate a comparison on the convergence properties of the single, centralized, and decentralized agent systems. In this test we run a single trial with 6 agents for the multi agent systems (centralized and decentralized).

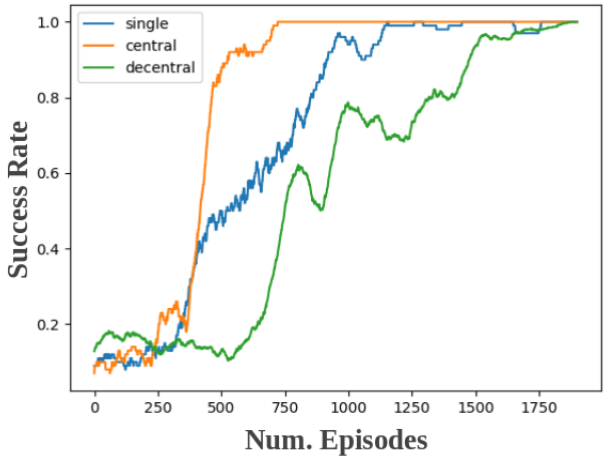

Figure 2: Comparison of single agent, centralized parameter server, and decentralized agent reinforcement learning algorithms.

Fig. 2 shows the relative convergence of the 3 learning agents over roughly 2000 episodes of training. The centralized agent learns the quickest, achieving optimal performance in roughly 750 episodes. The single agent performs an intermediate amount relative to the centralized and decentralized agents, and the decentralized agent learns the slowest. We also note that the stability of learning is also ranked the same amount as the learning rate of each agent, with the centralized agent having the most stable learning progress and the decentralized with the most unstable learning progress.

Initially it was unexpected to see the relatively poor performance of the decentralized agents, given that roughly 6x the amount of data is being generated relative to the single agent. Fig. 3 provides an intuitive explanation as to why the diffusion in the decentralized system can have a destabilizing effect on learning.

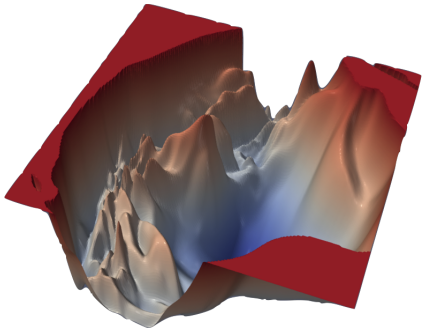

Figure 3: Visualization of loss landscape of a neural network [3]

The loss landscape of a neural network is quite nonconvex (in this context convex functions are those that are of the form $\alpha f(x)+(1-\alpha)f(y) \leq f(\alpha x+(1-\alpha)y)$). This means that averaging functions in this space will not results in a value that is in between the values of the two functions. Translating to the RL domain, this means that mixing network weights of the policies will not result in a policy that performs in between relative to the original two policies. This means that in the diffusion process, there is actually a lot of instability generated by the mixing of policies over time. In the steady state, this process is proven to settle to the optimal policy of the environment, but the transitory learning process is quite unstable.

3.2 Effect of Network Topology

The next test we performed was the performance benchmark against different network topologies. We tested 3 different network topologies: linear chain, fully connected, and spoke-and-hub. This provided a good spread of different network structures from complete communication (fully connected), communication with only a single node (spoke-and-hub), to minimal network communication (linear chain). Fig. 4 shows a variance plot of 5 rollouts with each network having a total of 6 agents.

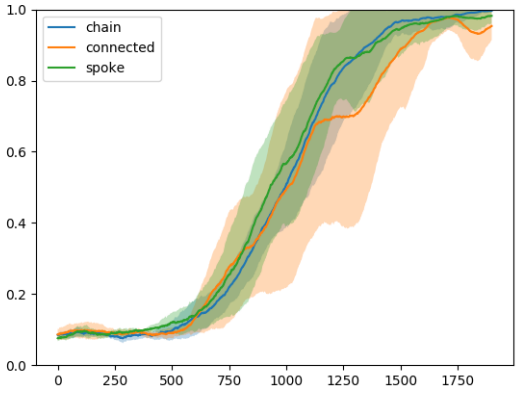

Figure 4: Network topology comparison.

We see that there is no significant difference in the convergence of any of the network topologies. We note that in the original Diff-DAC paper [], the authors also note that network topologies become a bottleneck in performance at around 100 agents. Due to the analysis in the previous section, we believe testing with a larger number of agents will show significant performance decrease for the fully connected agents. This is due to the polynomial increase in the number of connections and thus diffusion that will happen in the network, thus destabilizing much more than more loosely connected network structures.

3.3 Fail Stop Fault Tolerance

The next two experiments were in different types of fault tolerance. The first is to test the networks robustness to Fail Stops. In this situation we define a fail stop as an agent going offline for a certain number of episodes, and then recovering without memory of the policy it had before failing (initialize with random weights). Fig. 5 shows the results of this test in a spoke-and-hub network with 6 agents (5 agents only communicate with single hub agent).

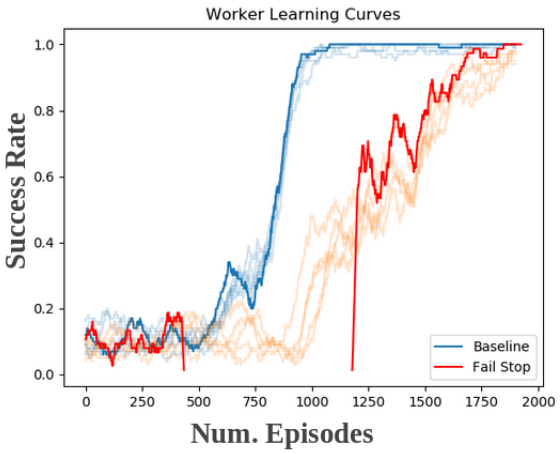

Figure 5: Fail stop test. The failure happens at episode 500 and continues until episode 1200.

The results of this experiment are very promising. They show that, compared to the baseline, the failed node is able to quickly recover due to the aggressive amount of diffusion from all the other agents that have been independently learning good policies. This resulted in only roughly a 1000 episode lag to optimal performance. This is only 300 episodes longer than the fault had occurred over!

3.4 Network Partition Fault Tolerance

The next fault test was the robustness to network partitions. The network partition in our experiment is defined as severing a connection between to portions of the network for a certain number of episodes and bringing them back online afterwards. The networks then diffuse information to recover.

Fig. 6 shows the results of the test for a network of 6 agents, each partition having 3 agents each, and a single edge connecting the two partitions.

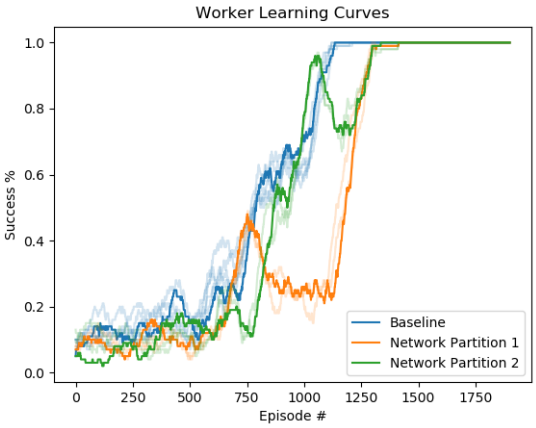

Figure 6: Network partition test.

Similarly to the previous fault test, we see that the network is quite robust to this type of network stress. We see that after the connection is severed at episode 500, the two network partitions diverge. The network denoted in green was quickly able to find a near optimal policy like the baseline, but the orange network still struggled to find a policy that was useful. Once the connection came back online, the green partition was able to diffuse useful policy information into the orange partition in order to quickly bring them back to a reasonable policy that become optimal in only a couple hundred episodes. This lag in convergence was even less significant than the fault stop tests.

4 Conclusion

We have showed that decentralized systems are quick robust to classical problems in a distributed systems setting. Sacrificing a some instability in the learning rate of the system, we are able to gain impressive robustness against many types of faults. In the future, it would be interesting to test what failure modes this system has to byzantine faults as well.

We believe that decentralized reinforcement learning systems will become more popular in the coming years due to a few main benefits. The reduced latency that comes with moving computations to the edge of the cloud allows situations where response is fast and learning occurs slowly behind the scenes. We also imagine that there is some degree of privacy that comes with these systems as well. Rather than sharing personal information to a centralized parameter server for processing, the devices will only need to share model weights within the local network. There are many interesting problems that are still left to solve with these types of systems and with reinforcement learning in general, but we be-

lieve that decentralized learning will provide a powerful framework in the future.

References

[1] Sergio Valcarcel Macua, Aleksi Tukiainen, Daniel García-Ocaña Hernández, David Baldazo, Enrique Munoz de Cote, and Santiago Zazo. Diff-dac: Distributed actor-critic for average multitask deep reinforcement learning. *arXiv preprint arXiv:1710.10363*, 2017.

[2] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[3] Tom Goldstein. *Visualizing the loss landscape of neural networks*, 2018.