Artificial Intelligence Explainability Accountability Lab Research Report

Richard Dao
rqdao@ucsc.edu
University of California, Santa Cruz

1 KUBERNETES & NAUTILUS

For week one, our tasks were to read through the Nautilus guide, get onboarded, and create our first job/deployment. After setting up an account with Nautilus and getting added to the namespace, I followed Coen's demo to get a simple job running.

1.1 Running the Job on Nautilus

1.1.1 Sources of Error

When following the guide and copying the given yaml files, I ran into an error. Aftering copying the yaml files, I ran these commands in order:

```
$ kubectl create -f
testpvc1.yaml
$ kubectl create -f
testjobpod.yaml
$ kubectl cp decisiontree.py
test-pod1:/pvcvolume/
$ kubectl create -f
testjob.yaml
$ kubectl delete job testjob
$ kubectl cp
test-pod1:/pvcvolume/tree.png
tree.png
```

I ran into trouble with the last command, where tree.png didn't exist. Upon debugging by calling command

```
$ kubectl exec -it test-pod1
-- ls /pvcvolume
```

I noticed that there was no tree.png in /pvcvolume. Upon further inspection using:

```
$ kubectl get pods
```

I found that the job had created 5 pods that all had status of Error (in other words they failed) before it reached its bakoffLimit (set to 5).

Using command:

```
$ kubectl logs <pod_name>
```

I found that the error all the pods were getting was Errno 13, Permission Denied.

1.1.2 Solution

I modified the yaml file for testjob such that it employs an init container to set the permissions before the main container starts.

```
template:
  spec:
    initContainers:
    - name: init-permissions
      image: busybox
      command: ["sh", "-c", "chmod
      -R 777 /persistentvol"]
      volumeMounts:
      - name: persistentvol
        mountPath: /persistentvol
    containers:
        # Rest of demo here
```

1.2 Results

After adding this code, the job was able to finish successfully and produce tree.png.

2 Running AV Simulators Locally

The goal for week 2 was to get an autonomous vehicle simulator running on our local computers. I'm using a Windows 11 environment, so I opted to use CARLA.

2.1 CARLA Installation

Following the CARLA quickstart guide found here, installing the CARLA package was straightforward. I opted to use option B. Package Installation as it was simplest. I ran into trouble when I tried to install the CARLA library from pip3.

```
pip3 install carla
```

What I realized after hours of debugging was that I was simply using the wrong python version. I was on Python 3.12 and the CARLA library was only installable using Python 3.8 or earlier.

To get around this, I employed the use of miniconda and created a conda environment that used Python version 3.8.

```
conda create --name carla python=3.8
```

with this issue resolved, I was able to run the CARLA library and begin playing around with the provided test scripts within the PythonAPI folder.

2.2 CARLA's Baseline Experiment

By navigating to the PythonAPI folder, I took a look at all of the different test scripts we have access to that show how to control different variables within the CARLA simulator environment.

While running these baseline scripts was easy and straightforward, reading them was a little tedious, and I ultimately found that it was more helpful to read the API documentation and write my own little scripts on how to set up a CARLA environment.

3 CARLA & Nautilus

Week 3's tasks saw a sudden increase in complexity mainly due to difficulties with Nautilus and an unfamiliarity with Kubernetes in general. Being thrown into the deep end, a lot of my fellow auditors seemed to get stuck at this stage as many found themselves simultaneously preparing for midterms and dealing with increased workloads.

While there were challenges, it is in this week's tasks that I found myself learning the most about Kubernetes, Nautilus, and CARLA.

3.1 Oliver's Repository

At the auditor meeting, it seemed that I was a week ahead of the given pace, so I got a head start on trying to get CARLA working on Nautilus. Initially, I searched Youtube and Google to see if there were any tutorials on the matter, but I quickly realized that Nautilus was a highly specific resource to this lab.

Looking into old messages on the Discord channel: *nautilus-support*, I stumbled upon Oliver's repository for setting up a Nautilus GUI Desktop and instructions on how to install CARLA on a Nautilus *Deployment*.

3.2 Nautilus GUI Desktop

Following the instructions from Oliver's setup README.md here, the process to get a GUI Desktop of Nautilus running was pretty straightforward. The key challenge was retrieving a turn-shared secret from Nautilus Support on Matrix.

In addition, initially, the xgl.yaml file that was given by Oliver was a deployment. However, when I asked in the nautilus channel whether or not I could run the Deployment, I was denied and told to convert Oliver's Deployment into a Pod.

Seeing as how I had little prior experience with Kubernetes, figuring out what I needed to change in the file was tough, but ultimately deleting just 4 lines of the code resolved the issue.

3.3 Installing CARLA on Nautilus

With the GUI Desktop ready, we were ready to install CARLA on Nautilus. Luckily, Oliver also had instructions on how to do this already in his README file with helpful instructions on debugging the common errors he saw.

3.3.1 Limitations of Pods

A problem that I quickly realized and ran into was that given auditors were now limited to only Pods, we would have to reinstall CARLA every single time we created or recreated our Pod.

Additionally, one thing that was confusing was how to take advantage of our persistent volumes. Originally, I thought it would be a good idea to try and install CARLA in the persistent directory so that I wouldn't have to re-install it every single time I re-created my Pod, but this resulted in significantly slow performance and installation issues.

3.3.2 A Helpful Bash Script

My solution to this was to make a bash script that I could store in the persistent path that automatically:

```
- installs miniconda
- creates conda env in python 3.8
- activates env in ~/.bashrc
- installs the carla package
- installs recommended libraries
```

I made a pull request to Oliver's repository to add my bash script to help others with this similar issue. The script was accepted and can be found here. My first small contribution to the lab!

Coen proposed that another way to automate this process would be to do it directly in the .yaml file. While I tried to integrate the script with the xgl.yaml files, I found that there were too many permission errors to debug and was ultimately not worth the extra time to resolve when the script version was sufficient.

4 Stable Baselines with CARLA

Week 4's tasks also saw another sudden increase in difficulty, with this week's tasks spilling over into the remaining weeks of the quarter. The main challenge was figuring out environments and how to set up a CARLA gym environment to train on one of the Stable Baseline algorithms.

While the process was told to be straightforward, I found that there was little support for it on the internet and on the Discord. At status meeting, I asked to see if there were any guides from previous auditors and lab members that could help with setting up Stable Baselines to CARLA or explain the process better, but ultimately there weren't any available. (So here's how my group partner Vinh and I did it!)

4.1 Figuring Out Environments

Since both Vinh and I were unfamiliar with CARLA, we opted to spend a couple of working sessions to just play around with the simulator and see what we could figure out. One of the more difficult aspects of the CARLA environments were tick-rates and ports.

4.1.1 Common Errors

For example, if you navigate to the PythonAPI/example folders and tried running the manual_control.py file, you'll always get an error that looks something like this:

```
RuntimeError: trying to create rpc
server for traffic manager; but the
system failed to create because of
bind error.
```

This error was due to the fact that by default, the traffic manager binds to port 8000, which was the port we were already port-forwarding and hosting our Kubernetes Pod on. We were able to fix this by editing the manual_control.py file to hardcode the traffic manager port to something not 8000. This error shows up a couple more times in different situations and was ultimately pretty tedious to handle. A better solution would have been to simply change the .yaml file so that our Pod isn't hosted and port-forwarded on a conflicting port to a CARLA default.

The other common error was tick-rates. With the modified manual_control.py file, we

tried to run multiple scripts at the same time. For example, we ran generate_traffic.py simultaneously with manual_control.py and found that the simulation was incredibly sped-up. What we found was that the tick-rates defined by each file were conflicting with each other, and setting one of the scripts to asynchronous (which ultimately meant it based its tick rate on the other files) helped maintain stability visually and mechanically.

After a lot of just playing around with CARLA, we finally had enough of an understanding of the environment and how the API was structured to create our own environments.

4.2 Figuring Out Stable Baselines

Prior to this, my experience with Reinforcement Learning algorithms was limited to the Pacman PacAI assignment from CSE 140. Reading up on how Stable Baselines worked was confusing as a lot of the prerequisite knowledge that was assumed was how fundamentally these algorithms worked. However, the tasks for the following Week 5 where we were asked to research one of the core RL algorithms and present on it helped push me to look into their functionality and ultimately taught me how they could be used for training autonomous vehicles.

4.2.1 Action Spaces

One of the pressing difficulties of using some of the Stable Baseline algorithms was figuring out compatible Action Spaces. The first algorithm that I tried to run on the simulator was DQN, which failed with the error:

```
AssertionError: The algorithm only
supports (<class
'gymnasium.spaces.discrete.Discrete
'>,) as action spaces but
Box(-1.0, 1.0, (2,), float32)
was provided
```

PPO, SAC, and DDPG were all able to run fine on the environment, but DQN was always failing. After a quick look into the Stable Baselines documentation, it shows that DQN only supports Discrete action spaces, which means in order to get it working with CARLA, we would need to create a custom Wrapper to convert continuous action spaces to discrete ones, which felt too tedious.

4.3 MACAD Gym

While I understood how to make my own environments, I was having difficulty setting up a gym that was compatible with Stable Baselines. While looking through Oliver's CERIAD repository, I found that he was using a public repository that handled creating a CARLA-StableBaselines compatible environment.

The repository, named MACAD Gym, provides a Gym-compatible learning environment that is built on top of CARLA. With this interface, all we have to do is install the repository and accompanying python package, import it and specify some of the environment parameters. Our repository with this setup can be found here.

5 Our Presentation

Given the circumstances of campus and our limited time in class, we were unable to fit a second presentation into the auditor meetings. Fortunately, we were one of the few who did present their first round of slides going over the Reinforcement Learning algorithm: DQN. A link to our slideshow presentation can be found here. We were unable to implement the algorithm for CARLA given difficulties with continuous versus discrete action space representations.

5.1 Reviewing What We Learned on DQN

5.1.1 Q-Learning

Deep Q-Networks (DQN) are a reinforcement learning algorithm that is based on the Q-learning concept. Q-Learning is a mathematical algorithm that predicts an expected sum of rewards in an environment. What is commonly done when employing the algorithm is to store outputs for actions based on game states in a table for quick and easy lookup. As the agent exposes itself to the environment, it receives an reward, an indicator of it's progress (or lack thereof).

5.1.2 Problem

The problem is that Q-learning becomes impractical when the number of states and actions is large (or even infinite). Consider a game with 1000 states and 1000 actions per state, the corresponding Q-table would contain 1 million cells. Q-learning also struggles in situations where the possible actions are and game states are infinite, so in a situation where it enters a new game state that it has never trained on before, we will have to arbitrarily tell it what to do instead of making a decision based on the table.

5.1.3 How DQNs Solve The Problem

The intuition behind DQNs is we can use deep neural networks to act as the Q-values *approximator* for each (state, action) pair. This way, we simply train a neural network instead of having a lookup table. This solves both problems mentioned before as it saves us from maintaining a potentially large lookup table **and** it allows for predictions in states we haven't seen before.

5.1.4 Architecture

The papers we read on DQN described the architecture as 3 entities. The Q-Network, the Target Network, and the Experience Relay.

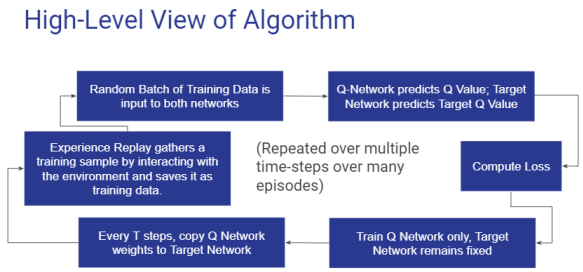

Fig. 1: Our diagram of the DQN training loop.
Source: *Our Slideshow*

The Q-Network, also known as the agent, is trained to produce the optimal state-action value. Originally, the architecutre used the singular Q-network to select actions and evaluate its own Q-values.

The Target-Network, which was added in the updated 2015 version of the paper by the same authors, is an identical replica to the Q-Network. The reasoning for its addition is because originally, the Q-Network by itself would provide both the predicted Q-value and the target Q-value. The weights get updated at each time step, which improve the predicted Q-value, but also our target Q-value. This results in a moving target for our network which might lead to difficulties in converging. Therefore, the target-network's purpose is to hold fixed parameters copied from the Q network with a delay on updating them so that our Q-Network has a stable target to compare to.

The Experience Relay is the mechanism in the architecture that gathers training samples by interacting with the environment and saves it as training data for the networks. It's addition to the architecture is to ensure training samples in batches to limit their variance and promote generalization across timesteps.

5.1.5 Results

The algorithm was tested on 7 Atari games and was able to outperform previous RL algorithms on 6/7 of the games. Additionally, it suprassed expert human players on 3 of the games and the episilon-greedy behavior policy annealed linearly from 1 to 0.1 over the first million frames.

6 What's Next?

Part of the auditing process was figuring out and understanding what tools, technologies, and procedures I found myself being comfortable using and sticking with them.

As for continuing research projects, I'm interested in using OpenAI's API to implement a training loop that creates generative explanations for decisions made by the autonomous vehicle algorithms in the CARLA simulator. I'm not sure if this would be too ambitious of a project, especially as an undergraduate. I could definitely use some guidance on what kinds of questions I should be asking myself in order to properly put myself on the right track!

References

[1] Jay Bailey. Deep q-networks explained. https://www.lesswrong.com/posts/kyvCNgx9oAwJCuevo/deep-q-networks-explained. Accessed: 2024-06-14.

[2] Oliver C. Ceriad: Custom repository for advanced development, 2024. Accessed: 2024-06-14.
[3] CARLA Team. *CARLA API Documentation*. Carla.org, 2024. Accessed: 2024-06-14.
[4] Ketan Doshi. Reinforcement learning explained visually (part 5): Deep q networks, step-by-step. *Medium, Towards Data Science*, February 2021. Accessed: 2024-06-14.
[5] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.
[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.
[8] Praveen Palanisamy. Macad-gym: Multi-agent car driving simulator for reinforcement learning research, 2024. Accessed: 2024-06-14.

[1] [4] [6] [7] [5] [3] [8] [2]