Reinforcement learning in a symmetric multi-agent team-based environment.

Note: All code is available in my public 533V github repository, under the 'project' folder:
https://github.com/padster/CPSC533V/tree/master/project
Similarly, all videos are available in public Google Drive folder:
https://drive.google.com/drive/folders/1DOPRF4nJia_EckekpUNma4cO5F-P_rqC

1: Background

Rocket League is a soccer/hockey-like game where two teams of cars drive around and try to hit a ball into a goal, by driving into the ball and bouncing it off you. Players control the throttle and steering of their car, and can use a finite supply of boost to gain additional speed. It has a few properties that make it a good testbed for reinforcement learning techniques, being/having:

(a) a physics-based environment with simple movement logic
(b) a team-based sport with sparse rewards (~under 10 goals per 5 minute game)
(c) code for accessing replays of games and running a scripted agent in a game
(d) pre-existing in-game AI implementations that are not considered good.

The aim of this project is to use reinforcement learning techniques covered throughout the semester to see how well a robot agent's behaviour (an example can be seen here) can learn to behave like an expert human player (as seen here).

2: Environment

2.1: Data

To reduce the problem of learning a Rocket League bot AI into a tractable space, many simplifications were made. The state space was lowered by limiting the number of players on the field to the minimal 1v1 format (i.e. 2 players, one on each team). This results in 13 independent state variables per player, as well as 6 for the ball, for a total of 32. For more detail:

2.1.1: State: 32-dimensional vector of floats

Variable	Description	Data type
`b_pos_(x/y/z)`	Ball position (3D)	Float, in [-1, 1]
`b_vel_(x/y/z)`	Ball velocity (3D)	Float, in [-1, 1]
`me_pos_(x/y/z)`	Bot position (3D)	Float, in [-1, 1]
`me_vel_(x/y/z)`	Bot velocity (3D)	Float, in [-1, 1]
`me_rot_(x/y/z)`	Bot rotation (3D)	Float, in [-1, 1]
`me_ang_vel_(x/y/z)`	Bot angular velocity (3D)	Float, in [-1, 1]
`me_boost`	Percent of boost remaining	Float, in [0, 1]

`e_pos_(x/y/z)`,
`e_vel_(x/y/z)`
`e_rot_(x/y/z)`
`e_ang_vel_(x/y/z)`
`e_boost`
	Same as above, but for opposition player rather than agent.	Same as above

The X dimension represents the short axis (left/right) of the pitch, Y is the long axis (forwards/backwards), with the center of the pitch at (0, 0). Z represents height, with 0 at the ground position. For simplicity, Y values can be mirrored if required, to ensure the 'me' agent is always aiming towards the goal at positive Y (and defending the goal at negative Y).

2.1.2: Action: 3 dimensional, (float, float, boolean)

As with the state, to make the problem more tractable, only three of the full range of actions were made available to the agent. Care was taken to ensure that even with only this subset of actions, a skilled human player can still easily defeat the game's existing in-game AI implementations. The exact actions available are:

Variable	Description	Data type
`throttle`	Continuous, from full brake (-1) to full acceleration (1)	Float, in [-1, 1]
`steer`	Continuous, from hard left (-1) to hard right (1)	Float, in [-1, 1]
`boost`	Whether to spend boost to gain extra forward momentum.	Boolean

2.1.3: Rewards

In its simplest form, the game has one reward: victory (or loss) after 5 minutes of play. As a victory goes to the team with the most goals, and play is reset after each goal, in reality the true rewards are when a goal is scored (and a negative reward for goal conceded). This is extremely sparse however: Only in the order of 10 or fewer goals are scored during 5 minutes of gameplay (or roughly ~12000 frames of potential actions), so rewards are provided at most for only around 0.08% of actions. This also applies only to competent agents: random agents are likely to never score a goal, so will never encounter rewards.

Luckily, it is possible to augment these per-goal rewards with alternatives that can be applied each time an action is taken to encourage good behaviour. For more on this see section 4.

2.1.4: Termination

The game ends after 5 minutes of play, about 12000 action frames with variances explained by variable in-game delays after each goal. If scores are equal at that point, the game enters a sudden-death phase where the winner is whichever agent scores the next goal.

2.2: Data Cleaning

The game system itself provides ways to store full-game replays after a game is complete, and open-source python library `carball` can be used to access all state and action values for each timepoint within the match, as well as many other values that may be useful. As such, the first stage of preprocessing involved removing each timepoint's data that is not needed (e.g. car colours, extraneous actions, in-game camera options, ...).

Next, data normalization is used to convert all values into the ranges specified above. For physical data (_x/y/z columns), these are already symmetric around the centre of the field, and are normalized to [-1, 1] by dividing by the maximal found value across all replays, rounded up to a clean number. Finally, imputation is done separately for state and actions. For state, values are missing when players have crashed (i.e. another car hit them at speed), in which case we impute with the most recent valid values. Additionally, some data is missing at the start of the game, but again here it is safe to impute with the first valid value, as these all occur during game countdown before any actions can take place. Next, missing actions can be filled in by lack of action; i.e. throttle = 0, steer = 0, and boost = False.

2.3: Evaluation

Due to being a digital environment, to evaluate a trained agent the 'simulation' environment is the same as a 'real world' environment, and as the game is not open-source, execution within the simulation is only available at real-time speeds of ~30 (state, action) frames per second. Agent results can be obtained through integration with an RLBot python framework which receives a packet of current game state, and sends back the desired action. Full code is available in the botSrc folder, with the main section of interest being bot.py. Here we load the agent's net, convert the current state to pytorch tensors, use it to get pytorch action from the our agent, and finally the resulting tensor is converted back into actions that move the bot in-game. Evaluation can be seen through performance in-game, and videos are provided with this submission.

3: Attempt #1 - Behavioural cloning ('BC')

As replays are simple to obtain, and contain an entire 5 minutes worth of state and action trajectories, the first approach was to attempt behavioural cloning. For this, three different games were played with myself as the expert, and an opponent representing existing in-game AI implementations. Together, these total 37619 (state, action) pairs.

3.1: Network

A deep artificial network $f : S \rightarrow A$ was trained to predict expert actions given the expert's current state. As the network values do not contain many spatially or temporally repeating structures, a simple fully connected network was selected. Experimenting with different setups found best results with two hidden layers: the first with 32 nodes and the second with 8, with batch normalization and ReLU activation between each. Code for the model is available in models/StoAModel.py.

3.2: Training

To train the network, multiple replays' worth of data was loaded, split into (s, a) pairs, and parceled into batches of size 100. To evaluate the loss of each predicted action, it is first split into two parts: the continuous actions (throttle and steer) are mapped onto [-1, 1] using the sigmoid function, and compared to the expert action using mean squared error loss. The discrete action (boost) is boolean True/False, so is compared to the expert action using Binary Cross Entropy with logits loss. These two are weighted and summed for the final loss, which is minimized using the Adam optimizer. The trained loss can be seen in Figure 1a.

3.3: In-game results

After multiple iterations of optimizing the network structure, a video of the final trained behaviour can be seen in 3.3_BC_Result. The trained agent visually drives towards the right side of the ball at kickoff, and hits it to the left. This is a set position and behaviour encountered many times in the training data, and it appears to have learned this section. Very quickly however, the agent strays far away from the ball, and does not behave how an expert would. It is expected that the observed state is too far from what was experienced during training - that is, due to the large state space, the agent has quickly diverged, and is unable to get back to return to previously observed states. It is interesting to see the agent also recover out of net by boosting, as can be seen in the human player's example too, however it continues to drive from the ball for too long after.

3.4: Observations/Suggested changes

Rather than cloning behaviour from a human, which requires a lot of human intervention, a better proof-of-concept would be to clone behaviour from opponent AI implementations. These are much simpler to generate in bulk overnight, and would provide a lot more data.

Another improvement would be to gather expert behaviours in more unusual states. Similar to the DART technique by Laskey et al (2017), by swapping between random and expert control every 5 seconds of gameplay, this forces the expert into unusual situations, providing examples of expert recovery for the next five seconds. These would generate many samples of expert actions from non-expert locations, and the network would have better performance in sections of the state space with few examples normally.

4: Attempt #2 - Deep Q network ('DQN')

As done in class assignments, an attempt to rectify the cascading error was to switch from predicting A given S, to predicting a 'goodness' value Q from (S x A), using rewards and a temporal discount factor. As mentioned in 2.1.3, rewards are however extremely sparse, likely never to be experienced by random agents. Instead, we address this by adding expert-defined artificial rewards, calculated at every (state, action) timepoint, to encourage taking good actions and getting to positive environmental states.

4.1: Artificial rewards

Multiple artificial rewards are provided, with two classes: one being rewards based on a heuristic measure of the current state, and the second based on the action taken. Each reward is scaled independently, and the state and action rewards are combined to a global reward by taking a weighted sum (0.1 for state, 0.2 for action). These are manually defined and scaled using familiarity with the game. In detail:

Reward	Description	Scale
State rewards		
`NearBall`	Linear: 1 when agent is close to the ball, 0 when the agent is far.	3.0
`NearGoal`	Linear: 1 when the ball is near to the target goal, -1 when it is near our own goal.	0.3
`HasBoost`	Linear: Equal to our current boost level [0, 1]	0.1
`BehindBall`	1 if the agent is closer to our own goal than the ball, 0 otherwise	0.2
Action rewards		
`Forwards`	Linear: 1 if we're driving forwards, 0 if driving backwards	0.6
`Straight`	1 if we're driving straight, linear out to 0 if turning full left or right	0.6
`SaveBoost`	1 if we're not using boost, 0 otherwise	0.1

4.2: Network

A network similar to 3.1 was used, however now the shape is $f : S \times A \rightarrow \Re$ as the input is the current state and action combined, and the output is the scalar Q value for that pair.

Note that previously in the course, the Q network was still S->A, as its output defined a value for each action. As our action space is continuous, A is instead treated as an input. This does make it harder to find the maximal action for a given state (and its Q value) - which are required both during training (for $max_a\{Q(s',\ a)\}$) as well as inference (to know which action to take).

To solve this, we discretize the action space by trying only three values for throttle and steering: being [-1, 0, 1], at the extremes and no action. In combination with the two boost action options, this results in 18 actions to try. To find the best action for a given state, we try each 18 in turn, concatenate to the state, and find the optimal Q.

This discretization does not affect the ability of the agent too much: many human experts are limited to this discrete action space due to the controller hardware they use, and all data recorded also had this constraint. Note also that training occurred similar to the behavioural cloning network (see section 3.2), with the exception of the loss calculation. As the output is (a batch of) real numbers, a single mean-squared-error loss is used.

4.3: Validation

To validate this approach, first graphs were generated to ensure that in the saved replays, the expert was indeed gaining more artificial rewards than the lower-skilled in-game bots. The rewards collected over time, as well as a value for average per-frame reward, can be seen in Figure 2.

Secondly, the weights were changed to reward a very specific behaviour. By swapping the `Forwards` reward to -1.0, and everything else to zero, this penalizes driving forwards, instead rewarding reversing. Interestingly, the agent did not perfectly learn this reward despite it being simple (Figure 1b), however it did learn to drive backwards to maximise reward. Video of the results is available at 4.3_DQN_BackwardsRewarded.

4.4: Real world results

A video showing results when trained against artificial rewards from the entire collected dataset can be seen in the video 4.4_DQN_Result. As with the behavioural clone network, it starts well, however a clear difference can be seen in smoothness. Likely due to discretization of actions, it appears the network is constantly deciding to change action, as the reward tradeoff keeps switching (e.g. drive straight vs drive towards ball). In this case, it fails to hit the ball, instead getting rewards by picking up two of the small yellow boost pads, eventually reaching a state on the wall which it is unlikely to have experienced before.

4.5: Observations/Suggested changes

One interesting observation was that using the double-Q technique was slow to converge with large gamma and rewards. If the average reward is denoted $\bar{r}$, the first Q model was found to average zero as an output. After switching, the next Q model averaged $\bar{r} + \gamma * 0 = \bar{r}$, the next averaged $\bar{r} + \gamma\bar{r}$, the next averaged $\bar{r} + \gamma(\bar{r} + \gamma\bar{r})$ etc… The closer γ was to 1, the slower it took for the models to represent the true average Q value of $\bar{r}/(1-\gamma)$. For gamma too high (e.g. 0.999), network noise caused the errors to snowball towards infinity as the average value kept growing exponentially.

To rectify this, gamma was set to 0.7, to reward future values but also not take too long for the network to converge to stable Q values.

One approach that was planned but had to be omitted in the interest of time was using an approach from Goecks et al (2020), that uses the existing behavioural-clone network's predicted action as part of the artificial rewards. This would allow rewarding not only actions that lead to positive states, but also actions that 'look like' expert behaviour.

A second action reward worth investigating that would be simple to implement would be to smooth the changes in action, by rewarding similarity with the previous action taken.

5: Discussion

5.1: Issues encountered, and attempts at fixing/minimizing

Unfortunately, despite quite substantial simplification of the problem space, the results above show that a reasonable agent was still unable to be trained. There were a number of observed issues why this would be the case:

- Setting up a learning and testing environment for quick iteration took much longer than expected, even knowing that initial setup was going to take a long time. Thankfully it is a once-off cost, so this is no longer an issue for the project. One particular issue worth noting was that a lot of preprocessing/training code was copy-pasted between notebooks and the RLBot source, and these copies could easily drift apart. The unified `lib/` folder was not created until late in the project, creating it earlier would have saved substantial time when quickly testing different options/parameters.
- Using RLBot to test an agent's performance in the 'real-world' (i.e. simulator) was slow. Not only as it has to run in real-time, but often simple coding errors (e.g. typos, wrong index, …) would cause runtime errors that were only discovered a minute or two after starting the simulator. Access to a faster simulator is not possible (without help from the game's developers), so one possible way to mitigate this slowness is to use python's type annotations.
- Similarly, training a network went slowly - even for the three replays, running hundreds of epochs of the ~38000 training examples took in the order of 15 minutes, which is about as long as the data took to record, suggesting that online learning may be difficult. This should easily be able to be rectified by switching to using a GPU, however I was unable to get this working with pytorch. For future work on this project, either more time will be spent getting the pytorch cuda device working, or more likely, the code will be switched to tensorflow, whose GPU integration is already working on the computer used for training.
- A number of issues were found where state or action values were in the wrong order, or of the wrong scale. This occurred in a few places due to the mixed use of pandas dataframes and numpy arrays everywhere, and led to errors which were difficult to debug. Many were fixed by sharing mode code between different source files, but perhaps more uses of objects or keyed data-frames could help ensure that any array of numbers represents exactly what it should represent.

5.2: Future work

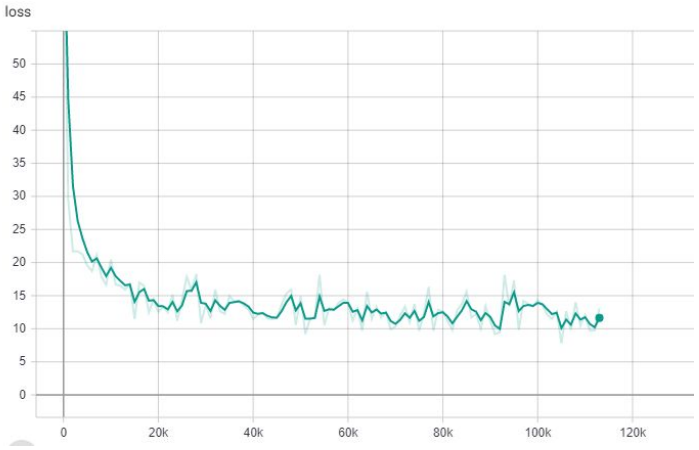
Now that the start-up cost of setting up the initial framework is complete, I intend to keep trying to improve the current setup, to get to the stage of sane agent behaviour. In order to progress, a few things must happen:

- I will begin by enforcing the action discretization mentioned in 4.2. This was only done for DQN when finding the best action given a state, however it does not limit the action space too badly, and will simplify a lot of future techniques.
- The task will be simplified even further, by ignoring the opponent completely. This reduces the state considerably, from 32 to 19 dimensions. Only once that version is solved, will a more advanced agent be trained that takes the opponent state into consideration.
- After these two are addressed, the next desired technique to implement is to enable using the DQN solution with online training. It still suffers from not seeing states that an expert is unlikely to see, which can only be addressed by letting the agent explore itself and learn from that. I was unable to get this performant enough to train the Q network during simulation when running on the CPU, however after GPU migration, this is likely possible. Despite the slow simulator execution, running this online overnight in conjunction with a novelty bonus (e.g. as seen in Burda et al, 2018) would allow exploring a large amount of the state space. This can be doubled by running two agents simultaneously within a single game.
- Techniques also exist that can be used to address the problem of reduced data. Chua et al (2018) model system dynamics and which allows trajectory sampling to generate more data. As the environment is a physics simulation with very simple dynamics, it is easy to predict the next state s' from the current state and action, so techniques to exploit this (i.e. imagined trajectories) would likely do well. Similarly, very many data-points from expert-only replays exist, which could be used to bootstrap any representational model using either behavioural cloning or unsupervised techniques such as autoencoders, extracting the smaller-dimensional feature-space that experts are using.
- Another very simple approach that may be worth trying is instead providing a history of past states and actions, rather than a single one. It feels as though having a single snapshot should be enough (especially as the physics is deterministic), but extending the dimensionality of the state and action spaces may provide more information the networks can use to learn how to act.

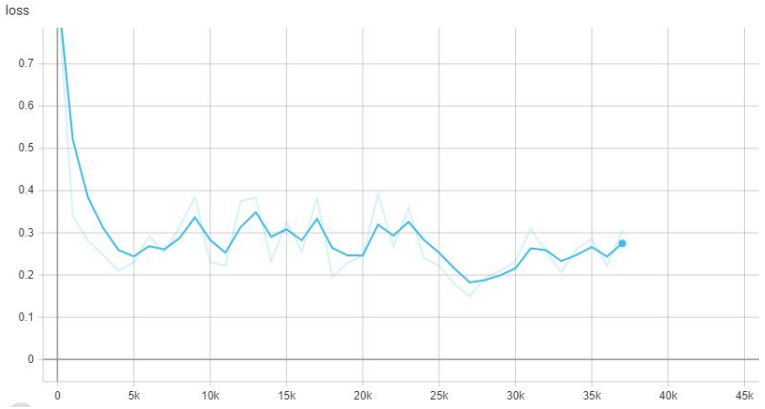
Appendix

Figure 1:

a) Training loss for the behavioural clone network over 300 epochs of all 37619 rows.



b) Training loss for the DQN network with rewards that penalize driving forwards.



c) Training loss for the DQN network with normal artificial rewards.

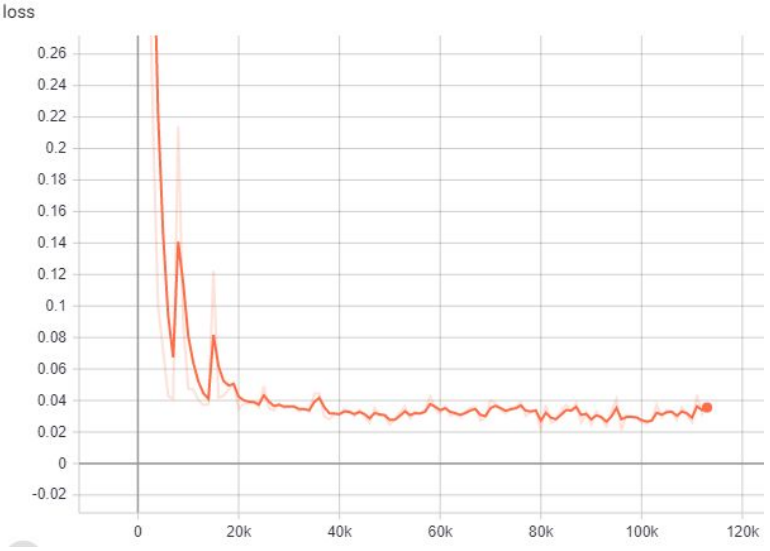
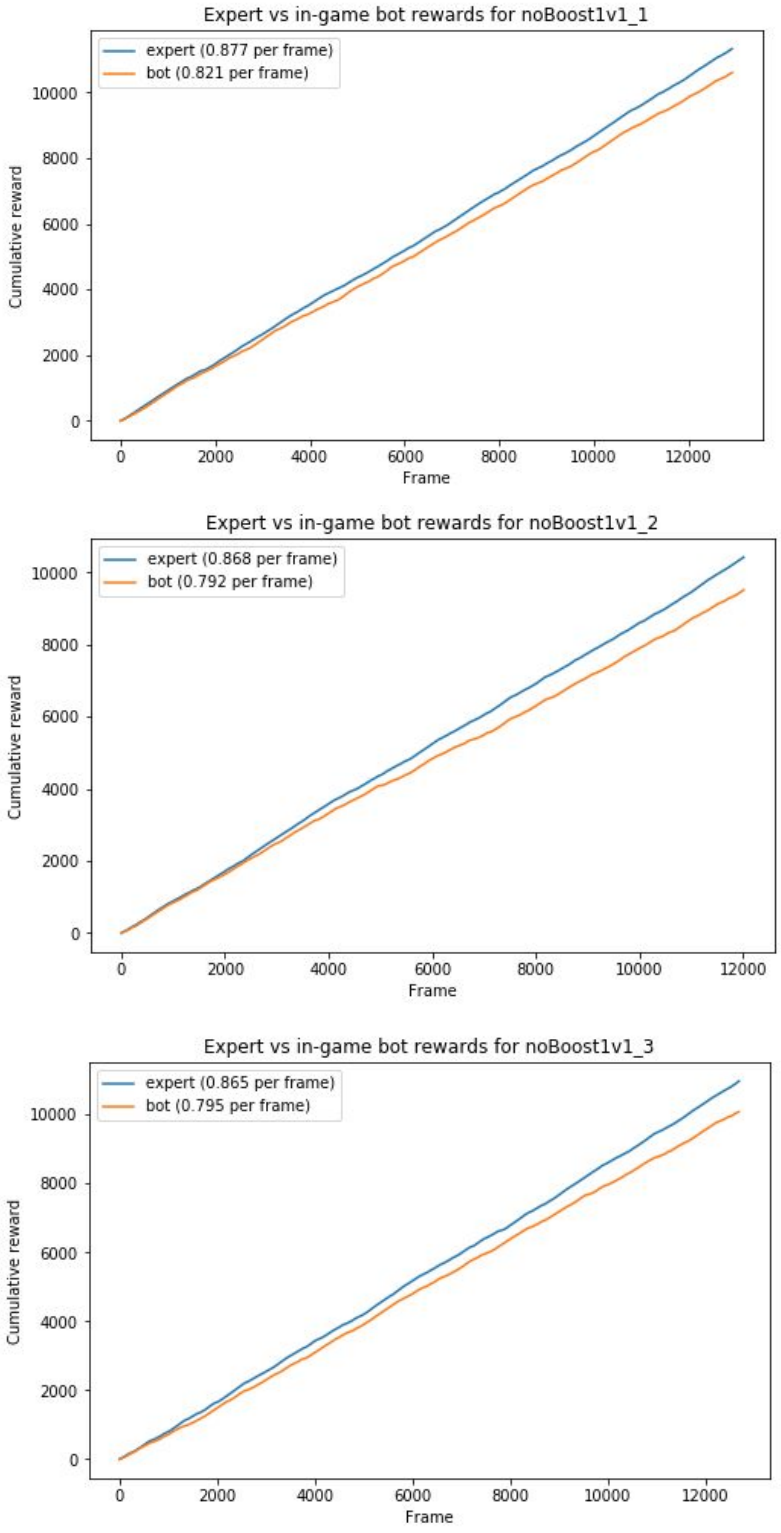


Figure 2:
Cumulative rewards for expert (blue) and in-game bot (orange) for the three replays, showing that expert rewards are higher, and roughly equivalent across games.



Videos: 4 videos are available in the folder linked at the start of this document. They are:
`1_ExpertExample`: Example mid-game behaviour from a human.
`3.3_BC_Result`: Final behaviour from a trained behavioural clone agent.
`4.3_DQN_BackwardsRewarded`: Behaviour from a deep-q-network agent, trained on artificial rewards that only rewarded driving backwards.
`4.4_DQN_Result`: Final behaviour from a trained deep-q-network agent with sane rewards.