Professor Alex Olshevseky

EC 418 - Reinforcement Learning

Team 7

EC-418 Project: Developing an End to End Double Deep Q-Network for a Reinforced Learning PySuperTuxKart Agent

Rashid B. Kolaghassi

Layth Amra

Max Malamut

Introduction:

PySuperTuxKart is a racing game library that allows for sensorimotor control experiments. Our objective is to teach the agent, the kart, to drive around the tracks as fast as possible through deep Q reinforcement learning. The objective will be assessed by recording how fast the kart is able to traverse the track.

Approach:

In order to develop a reinforced learning agent that can run on the track, we first outline our approach and explain in depth how we implement each step to develop a reinforced learning agent. Please refer to figure 1 that highlights our approach:

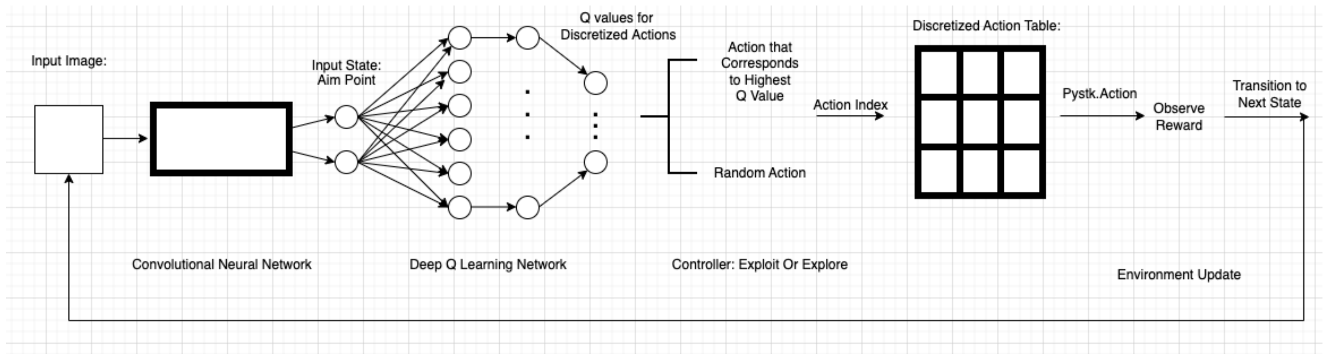

Input Image:
Input State: Aim Point
Q values for Discretized Actions
Action that Corresponds to Highest Q Value
Random Action
Action Index
Discretized Action Table:
Pystk.Action
Observe Reward
Transition to Next State
Convolutional Neural Network
Deep Q Learning Network
Controller: Exploit Or Explore
Environment Update

Figure 1: Flow chart of approach

The main idea is as follows: the kart is on the track and observes an RGB image of the track. Our first goal is to convert this image of the track to a state. This is done by using a convolutional neural network that is able to take this RGB image and output an X and Y coordinate on the track defined as the aim point. The aim point highlights where the kart should direct itself. Given its current state, the kart should take an action that directs it towards the aim point as motivated by our reward function. The kart can do this by taking the action that corresponds to the optimal

Q value outputted by the deep Q learning network. The input to the neural network is the X and Y coordinate of the aim point (i.e its current state) and the output of the neural network is the Q-values of all possible discretized actions that the kart can take. As the action space is continuous, we discretize it so that the number of actions the kart can take is finite. In the beginning of training, the kart will explore the track by taking random actions. Once the kart takes the action, it is now in a new state. We will observe the reward the kart has taken by completing this action through a defined reward function and cache the state the kart was in, the next state it observed after performing the action, the action itself and the reward. Each cached set of variables is named an "experience". We will recall a sample of these experiences chosen at random every few steps the kart takes to update our deep Q learning network through temporal difference and back propagation.

As the kart learns the track through exploring (i.e taking random actions), it will aggregate its rewards over multiple time steps to change its future actions and learn from experience. As the kart develops a better idea of which actions give it the better reward through exploring, it will reduce its tendency to act randomly (decay of the exploration rate) and choose the action that gives its largest Q-value as outputted by the deep Q learning network.

I. Creating the Environment

The first step in developing our reinforced learning agent is to develop the environment in which the agent can interact with and observe a reward from after completing an action. We decided to rewrite the environment code provided in order to better understand the possible variables in the environment that we can use to calculate rewards. We implemented the environment so that the environment receives a pystk.Action(), updates the environment based on the action taken and returns the observed reward of the action taken, the next state the kart is in, and whether the kart has completed the track.

II. Preprocessing the Environment

Our goal here is to use the environment data, which is an RGB image, to predict the aim point of where the kart should direct itself to. To do so we used a convolutional neural network that takes the RGB image as input and returns the aim point as output. Training data was collected previously by labeling frames of the race with the aimpoint at each frame as a function of the position of the kart and track. This training data was then fed into the CNN we designed and was trained over 100 epochs.

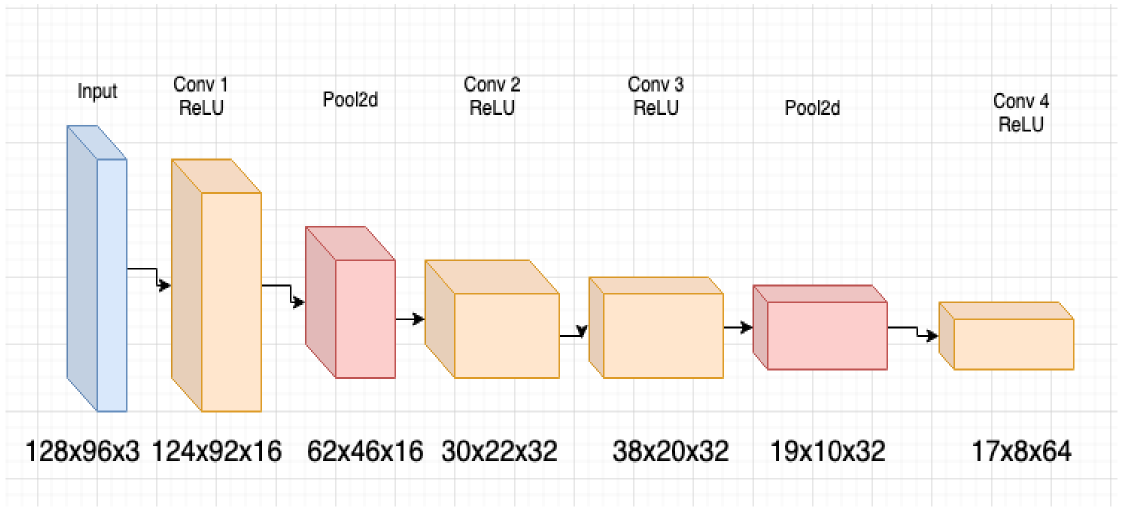


Figure 2: Network Architecture for determining aimpoint from RGB image

III. Creating a Deep Q learning Agent

First, we define a fully connected neural network architecture with a prescribed set of parameters. We make two versions of the neural network, one dubbed “online” and the other named “target”. The Q-agent will perform actions every frame based on an epsilon greedy-policy and it will store the state, action, reward, nextstate and done into its memory array. After a certain number of steps, which is a hyperparameter we define, then the Q-agent will recall a batch of its experiences and calculate the loss between

A. Act

Given an input state consisting of an X coordinate of the aimpoint and Y coordinate of the aim point, our agent should choose an action from the discretized action list by either “exploiting” or “exploring”

B. Cache and Recall

Each time our agent performs an action, he experiences a reward and a new state. We store these experiences (the state,action, new state, reward) in memory using the cache function. In our recall function, we sample a number of experiences (defined as a batch size) selected at random so that our agent can learn from them. This is known as experience replay. There are several advantages of using experience replay to teach our agent as compared to learning from consecutive experiences as done in standard Q learning. Learning from sequential experiences is inefficient as there is a strong correlation between the current state and the next state. Further, if we learn using our on policy network directly, we risk getting stuck in local minimas due to feedback loops. For example, if our policy tells us to take a right, our next states will be biased to the right hand states. Experience replay averages the behavior distribution over several previous states as the past experiences are selected at random, leading to a more stable learning experience and reduces the probability of parameters diverging or oscillating.

C. Learn

We use temporal difference learning to update our online neural network parameters.We first calculate our temporal difference estimate as follows:

$$TD_e = Q^*_{online}(s, a)$$

We do not actually know what the next action a' will be that gives us the optimum reward, so we use the action that maximizes our Q value from the online model as follows:

$$a' = argmax_a Q_{online}(s', a)$$

Next we calculate our target temporal difference by adding the reward observed to the discounted Q value of the target network in the next state as follows:

$$TD_t = r + \gamma Q^*_{target}(s', a')$$

Finally, we back propagate the loss as follows to Q online as follows:

$$\theta_{online} \leftarrow \theta_{online} + \alpha \nabla (TD_e - TD_t)$$

(Feng, Yuansong)

IV. Optimization of deep Q Learning Network

Parameter	Sub parameters:
Architecture of Deep Q learning Network	# of layers, size of layers, activation functions
Architecture of Aim Point Network	Aim-point taken 15 units down kart path, X-coordinate rounded to two decimals, Y-coordinate exchanged for the distance of the kart down the track
Exploration Rate	max($1*0.9965^x$, 0.1); where x is ticks
Decay Rate of Exploration Rate	0.9965
Reward Function	Size of rewards (avoiding local minimums)
Batch Size of Replay Learning	# of experiences used to learn from
# of Episodes to Train On	336
# of Tracks to Train	Why: Our implementation is suitable for generalization learning, as the state space of the aim point is the same regardless of track
Increments to Discretize Actions By	Five actions spread across -1 to 1, resulting in two degrees of turns and the option to drive straight

Table 1: Parameters of network

For the model to properly learn from its actions, it is important to have a direct correlation between input states, action, and then the reward received. During early iterations of the race environment, the rewards were given based on time spent on track(negative), needing to be rescued(negative), finishing the track(positive), and distance from aimpoint (positive if within a threshold, negative otherwise). The issue with this, is that the state-space only included the aim-pointer, and as a result, the model was unable to properly find correlation between its actions and the reward received. Continuously receiving a negative reward for spending time on the track is meaningless, as is finishing the track.

In later iterations, the reward function was primarily based around the state-space. For the state-space of the aimpoint, the rewards were based on deviation from the aimpoint. When the state-space included distance down the track, a reward for finishing was added, as well as a reward for staying on the track.

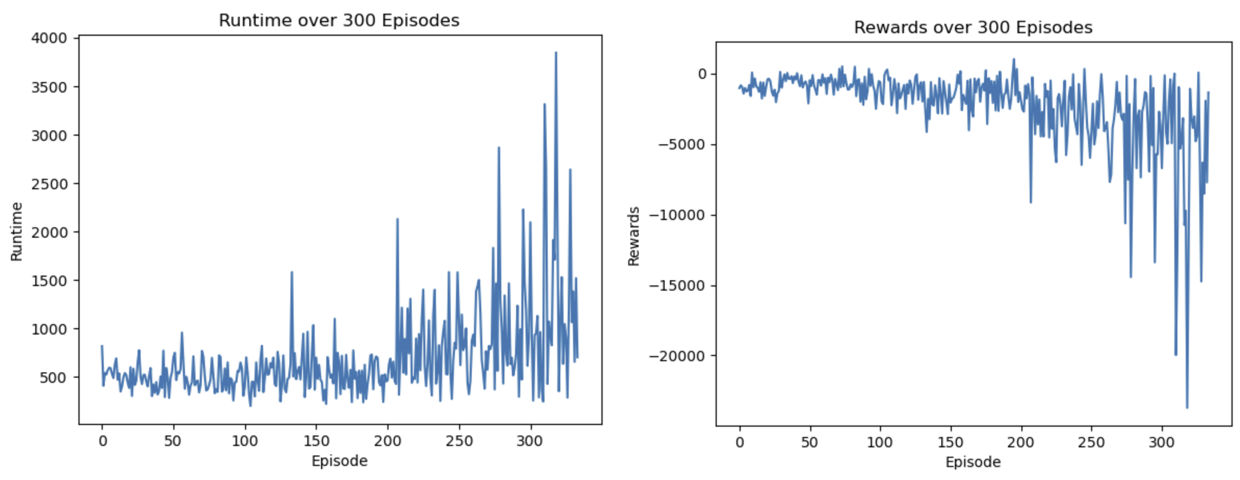
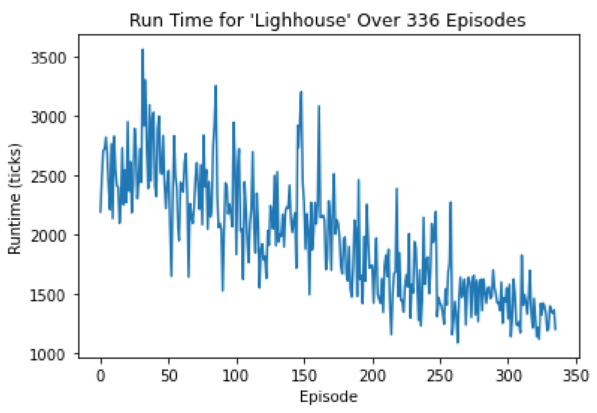


Figure 3: Runtime and rewards for first turn of "Lighthouse" over 300 episodes

Though the rewards and runtime are correlated, as shown above, for 300 episodes run over the first turn of the track "Lighthouse", they both seem to be noise, rather than any indication that the agent is actually learning and performing learned actions.

The second iteration of the Q learning agent, under the same training parameters, performed significantly better. It is clear from the results of training on the newest iteration, that not only is the agent learning, but that the agent is also converging.



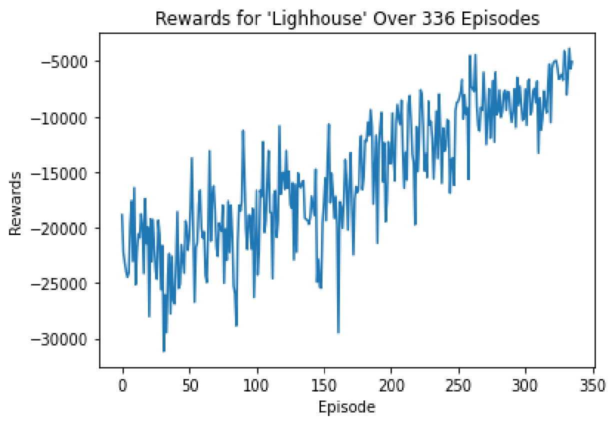


Figure 4: Runtime and rewards for "Lighthouse" over 336 episodes

This agent is conclusively learning, as both the rewards and runtime are decreasing. This is additionally for the entire track of 'Lighthouse', compared to the earlier data, displaying just the first bend. Unlike before, the appearance of significant random noise is not present. One pitfall of the newer learning agent is runtime; as for the 336 episodes, training took over 8 hours on a dedicated GPU. The end result was a total runtime of 63 seconds for the entire track.

Reward Parameter	Reward
Previous location is equal to current location[1]	-10
Aim-point X-coordinate within 0.15 of kart's direction vector, Else:	5, Else: -5
Finishing race	+500
Needing to be rescued	-30

Table 2: Reward function

[1] This serves two purposes. One, to make sure the kart has not stopped during the race. Also, the environment does not change the distance down the track if the kart is not currently on the track. By adding a penalty to not having the distance change, it also creates a penalty for going off the track.

Analytics:

Properly understanding the details about the agent were key in figuring out where changes needed to be made. Key information about each run was extracted, such as the reward from each action, the running reward, the action that the agent was performing, and if that action was random. In an early version of the agent, it was shown the agent was only performing two actions of the possible twenty. One was a left turn, and the other was a right turn. Because of this, the action space was reduced from twenty discretized values to five.

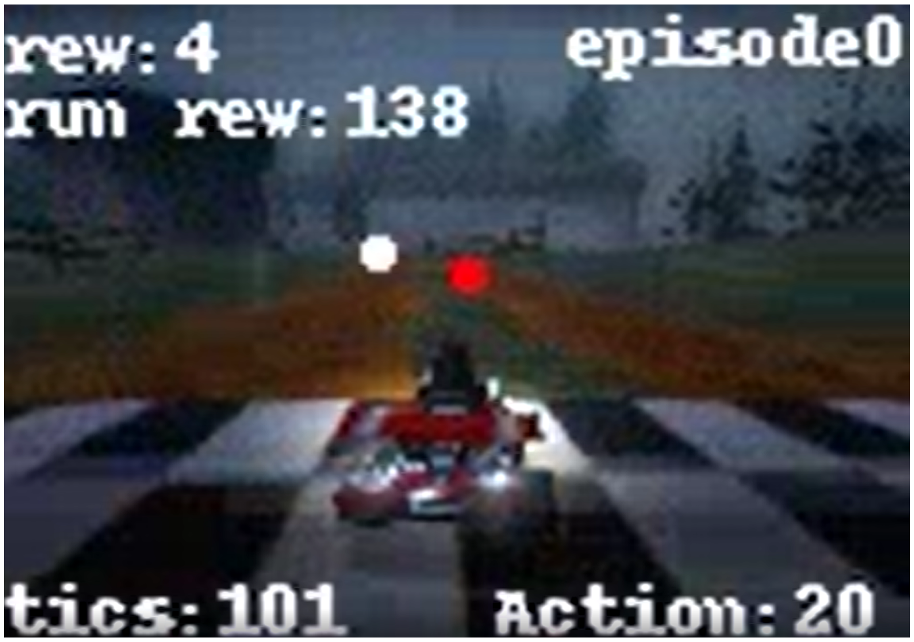


Figure 5: Example of live-run analytics

The above shows an example of how details from the agent can be helpful for gaining insights about how the kart is acting and learning. The top left shows the current reward for a given frame, below it, the running reward for the entire track. It also displays the current action the kart is taking; during training, it will also provide information about if the action is random or "greedy."

Improvements:

After realizing that the agent was not learning properly, we adjusted the Q learning agent's learn function to update Q values individually rather than changing all Q values for one learning iteration.

Next steps:

The agent has shown proof that it can learn with properly tuned reward functions as well as a better optimized network. The next step would be to create a true end-to-end agent that uses an image of the track instead of the aim-point. Additionally, the action-space should be expanded from just steering to also acceleration, drifting, and braking. Of course, this would require a new reward function, much longer training time, and perhaps an even more rigorous learning network. The initial agent had trouble learning twenty discretized steering actions over just a small section of one track. However, with proper scaling and rewards, it is highly likely that with enough training, the agent can be capable of learning multiple actions over multiple tracks.

Bibliography:

Feng, Yuansong. “Train a Mario-Playing RL Agent¶.” *Train a Mario-Playing RL Agent - PyTorch Tutorials 1.13.0+cu117 Documentation*, https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html.