Reinforcement Learning in SuperTuxKart

Final Project for CS342: Neural Networks with Dr. Philipp Krähenbühl

Yian Wong, Grace Liu, Cecilia Vu

1 SuperTuxKart

Our project is about a sub-game of PySuperTuxKart (PyStk): a 2v2 ice hockey game where each player is a kart. The PyStk game poses many different challenges from a reinforcement learning standpoint: it is a partially observable, multi-agent environment with very sparse rewards and continuous action space. Additionally, since it is an adversarial game, rewards are a function of not only the environment, but the opposing agents as well.

With this in mind, we approached PyStk from a reinforcement learning (RL) context, rather than creating a hand-tuned controller to play the game. We hence model PyStk as a Markov Decision Process (MDP) with transition function $T(s_{t+1}|s_t, a_t)$ and reward function $R(s_t, a_t)$ where s_t is the state of the karts and puck at time step t and a_t is the set of all actions taken by the karts in the model.

2 Reinforcement Learning

2.1 Method

With our environment modeled as an MDP, we are equipped to apply common RL algorithms to PyStk that have had success in other RL environments.

A big challenge we had to tackle was that PyStk as an environment was very expensive to query. The overhead mostly came from reinitializing the game in between episodes. Hence, sample efficiency was an important concern while planning out a RL-based approach.

We initially considered value-based reinforcement learning algorithms like Deep Q-Learning: they offer good sample efficiency and fast convergence compared to other methods. However, we decided that learning the Q-values of the environment would not be effective and likely very difficult in PySkt, since Q-values aren't necessarily fixed, and can often change based on the other agents in the environment.

Hence, we looked to a policy iteration method. We focused our work mostly on the traditional REINFORCE policy gradient learning rule:

$$\Delta\theta = \alpha(R - b)\nabla_\theta \log \pi_\theta(s, a)$$

In this update, α is the learning rate, R is the future discounted cumulative reward, and b is a baseline reward function. Intuitively, this learning rule makes sense because it increases the probabilities of actions that give good rewards, and decreases the probabilities of actions that give worse results. It has been shown that in expectation, $\Delta\theta$ will optimize a policy π_θ that maximizes the expected discounted reward.

We implement a variation of REINFORCE, which is Advantage Actor Critic (A2C). The A2C algorithm extends REINFORCE by replacing the baseline function b with the value function $V_\beta(s)$ of the current state, giving the advantage function, $A(s, a)$:

$$\Delta\theta = \alpha A(s, a)\nabla_\theta \log \pi_\theta(s, a)$$

$$A(s, a) = V_\beta(s) - V(s)$$

A big disadvantage of these policy gradient algorithms is that they are Monte-Carlo algorithms, and hence are on-policy and are extremely sample inefficient.

With the sample inefficiency of PyStk in mind, we decided to extend our implementation to be able to use previous trajectory samples of our PyStk games during training. To do this, we introduced an off-policy correction to our policy gradient algorithm by introducing an importance sampling term, which corrects for samples generated by a seperate behavior policy $\mu(s, a)$ [1]. We also bootstrapped our value network targets to one-step temporal difference targets, which we believed will help our model converge faster and be a better target given the off-policy setting [2].

With these off policy corrections the gradients for the value network ($V_\beta(s)$) and policy network become:

$$\Delta\theta = \alpha \frac{\pi_\theta(s, a)}{\mu(s, a)} A(s, a)\nabla_\theta \log \pi_\theta(s, a)$$

$$\Delta\beta = (v_{t_s} - V_\beta(s))\nabla_\beta V_\beta(s)$$

$$v_{t_s} = r_t + V_\beta(s + 1) - \gamma V_\beta(s)$$

2.2 Experiments

For all experiments, we discretized the action space to make the RL task simpler to learn. The native action space includes two continuous values (acceleration and steering) and 5 boolean values. We break the acceleration into 3 values (0, 0.5, 1) and steering into 7 (-1, -0.67, -0.33, 0, 0.33, 0.67, 1). We also assume that only at most one of the 5 boolean values can be selected at a time. This gives us 3 * 7 * 6 = 126 discrete actions.

The native state space included absolute positions of the karts and puck and vectors representing their direction. We created a normalizing transformation matrix that we apply on the coordinate system such that the bottom left corner of the map is (-1, -1) and the top right corner is (1, 1).

We initially ran a simple task with one kart being the only actor in the game, and the reward function was defined to be the negative of the kart's distance from the puck. After many trials, we learned a policy that went close to the puck.

After this, we moved on to including multiple agents and introducing new rewards. Our reward function now also includes the ball's distance from the home goal, and the negative of the ball's distance from the enemy goal. We also add a constant large reward if a goal is scored.

Each iteration, we train for 50 episodes and store experiences into a replay buffer. The replay buffer stores at most 1,000,000 state, action, reward tuples, discarding the oldest ones first. Afterwards, we compute the corresponding policy gradients and value gradients and train our neural networks.

2.3 Performance

Our method initially was successful in the simpler reward function of minimizing the kart's distance from the ball. An optimal policy that we'd expect is the agent hugging the ball and perhaps pushing it. Our algorithm was able to produce such a policy.

However, in the overall setting with the true objectives of the game internalized into the reward function, we could not produce similar success. After running several different configurations of hyper parameters and training over many nights on a GTX 1080ti GPU, we concluded that our method was unable to learn a successful policy for this project.

3 Imitation Learning

3.1 Method

After failing to implement reinforcement learning, the TAs coincidentally released multiple agents that performed reasonably well in PyStk.

We decided to employ imitation learning to produce a policy that learns from these TA agents. We implemented DAgger, a simple imitation learning algorithm we studied in class. The DAgger algorithm is an iterative policy training algorithm. We create training data for our agent by allowing the agent to play the game itself, taking actions it thinks are best, then recording the states it encounters.

During training, we get the expert agent's predicted moves for all the stored states and do supervised learning in order to get our agent's actions to match the expert agent's outputs. We realised that Jurgen's agent was by far the best expert policy available to us, so we decided to use this in our DAgger implementation.

Our implementation of DAgger is a bit different from the original implementation. Mainly, we do not perform single trajectories per iteration, but rather perform many rollouts every interation. We do this because we randomly initialize the ball state every rollout, hence doing many rollouts avoids early overfitting to one ball configuration. The general procedure is as follows:

Step 1: Initialize empty dataset $D = \{\}$, initialize counter $i = 0$

Step 2: Get expert policy π^*, initialize random policy π

Step 3: Rollout n trajectories of maximum length $50(i+1)$, store all states in trajectories into D

Step 4: Perform supervised learning on D, where we optimize for a loss $L(\pi(a|s), \pi^*(a|s))$ for all s in D.

Step 5: Increment counter $i = i + 1$, go to step 3.

3.2 Experiments

We use the same action representation as the TA's agents, which is modeling every action as an independent policy. We model acceleration as a continuous bounded value between 0 and 1, steering as a categorical value from -1, 0, 1, and brake as a boolean value.

In this experiment, we rollout for $n = 50$ episodes every iteration, and train using either negative log likelihood loss for discretized actions and mean squared error loss for continuous for 40 epochs.

As a result, our agent was able to score consistently against the TA agents and give us impressive results that are comparable to the expert model's performance itself.

3.3 Performance

Our imitation learning agent performed significantly better than our reinforcement learning agent and was able to successfully mimic the behavior of the Jurgen TA agent. After running 4 games against each of the expert agents, we performed as follows:

Imitation Learning Performance

Expert Agent	Points Scored	Point Breakdown (Us vs. Expert)			
Geoffrey	7	1:2	1:1	2:1	3:0
Jurgen	1	1:0	0:0	0:3	0:0
Yann	4	2:1	1:0	0:1	1:0
Yoshua	7	1:0	3:0	1:0	2:0

Though we scored an average of 4.75 points per set of 4 games, we won or tied almost all the individual games played against the TA agents and had barely any points scored against our agent. Out of the 16 games played, we scored 19 points on our opponents and only had 9 points scored on us.

3.4 Discussion

Our algorithm in the expectation only converges to the behavior of the expert policy, Jurgen. We came up with two approaches to further improve our model and go beyond demonstrator performance. This method uses evolutionary algorithms and a modification of DAgger- Every iteration, we create mutations of the current model by adding noise to its weights. During rollouts, rather than having the model play itself, we randomly pick between one of the mutations or the expert model to play in the match. Then, whoever wins the match labels the data that is added to the dataset and effectively determines what is the target expert behavior policy for that entire game. We believe this will exceed performance compared to DAgger and the TA agents, however due to time constraints we were not able to fully implement this idea.

References

[1] Thomas Degris, Martha White, and Richard S. Sutton. Off-policy actor-critic, 2013.

[2] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.