Automated Player using RL

Adriana Rivera

University of Texas at Austin

CS 394 Deep Learning

Instructor: Philipp Krähenbühl

December 2023

Author Note

UT ID: arr4933

MSDS Option III

Contact information: arivera007@utexas.edu

Abstract

The goal of this project was to design an AI agent to play Ice Hockey using the Super-TuxKart framework. The agent submitted was state-based. It was originally started as a Vision problem, image-based, but eventually was developed as a Reinforcement Learning problem, state-based. Developed using three consecutive phases: Imitation Learning, simple REINFORCE and REINFORCE with free-gradient optimization. The agent was trained using a simple Linear Layer that was spawned many times in the case of REINFORCE. THe first phase, Imitation Learning, only one layer was used and a good analysis of the selection of training data was imperative; this phase was moderately successful with only one goal achieved. At the time of this publication, the simple REINFORCE phase didn't score anything and the model actually got worse from the good initialization attained. For this phase, different initializations strategies were experimented including using a successful trained model by imitation learning from the previous phase but the Model still didn't improve and actually devolved. To make it perform better the following tuning were being made: inspect the sampling methods and distribution types, create a better Reward Function, increase the trajectories/games length. The last phase, REINFORCE with free-gradient Optimization was not even attempted due to the second phase still having problems.

Keywords: Imitation Learning, REINFORCE, gradient free optimization

Reinforcement Learning Automated Player

The agent type chosen was the state-based agent that uses the numerical state of the game at any given time during the interaction with Tux to predict the next action to take. The state-base agent was trained and experimented with using several methods using increased complexity.

Method

Reinforced Learning Phases

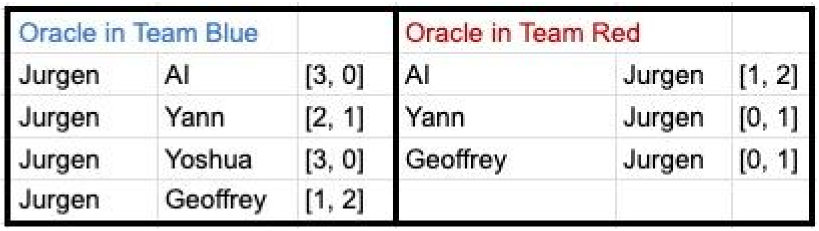
The chosen type of agent is state-based. The project was designed to be rolled out in 3 different phases as a mode of experimentation of the natural progression of more complex Reinforcement Learning algorithms. Using the same framework, it was desirable to compare the outcomes and the use of compute. The results of each of the following phases are embedded in each of the sections. These planned phases are:

1. Imitation Learning
2. REINFORCE simple (vanilla and model-free RL)
3. REINFORCE with gradient-free optimization
    a. Same NN as the previous phases for benchmarking
    b. Tune NN for performance against TA's agents.

1. Imitation Learning

1.1. Data Generation

For this algorithm, this step was critical. It started with using the provided agents and having them play against each other. Using the best scores for each team color category, the best player was chosen for each type of team, blue or red. This agent was named the Oracle or expert agent.



Oracle in Team Blue			Oracle in Team Red		
Jurgen	AI	[3, 0]	AI	Jurgen	[1, 2]
Jurgen	Yann	[2, 1]	Yann	Jurgen	[0, 1]
Jurgen	Yoshua	[3, 0]	Geoffrey	Jurgen	[0, 1]
Jurgen	Geoffrey	[1, 2]			

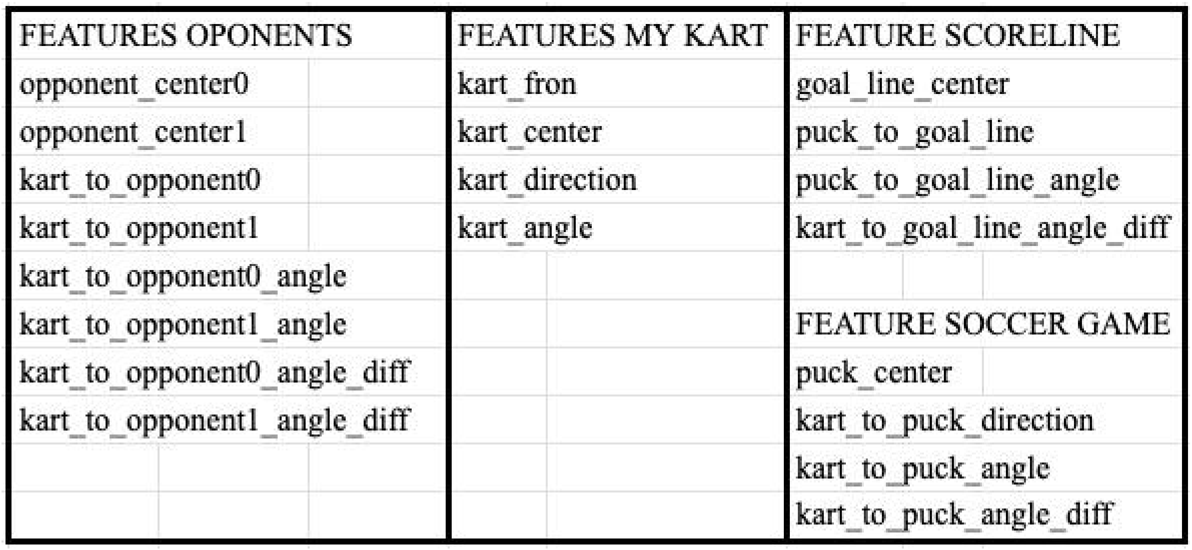
Figure 1. Analysis with Oracle agent to create training data for Imitation Learning. Oracle agent selected after the resulting scores for several agents playing against each other.

After choosing the Oracle, several games were played using it to generate the data for training. Being state-based, the resulting pkl files were accumulated and later transformed for the model consumption. The best agent played a total of 8 games playing either as a blue and a red team, each game event using 2400 frames, or steps. This resulted in a dataset of 19,200 entries. Another similar dataset was generated but using 28,800 frames per game and even a couple of games with 4,800 but surprisingly the games scores didn't improve that much and also didn't quite help the result model either. On the other hand, increased compute needs by a good measure.

1.2. Learning

Architecture. The network used for Imitation Learning was a simple Linear Layer, same as presented during lectures. More complex networks were not experimented because the goal wasn't to stay with Imitation Learning but to compare it with the next phase in simple REINFORCE. The later phases algorithms needed several instances of the neural network and a more complex network would have increased the already heavy compute. The resulting model converged quickly, with just 50 epochs with a learning rate of .001.

Features. Features used were very similar to the ones provided by the starting code. It was observed that all provided agents had extremely similar structures. List below.



FEATURES OPONENTS	FEATURES MY KART	FEATURE SCORELINE
opponent_center0	kart_fron	goal_line_center
opponent_center1	kart_center	puck_to_goal_line
kart_to_opponent0	kart_direction	puck_to_goal_line_angle
kart_to_opponent1	kart_angle	kart_to_goal_line_angle_diff
kart_to_opponent0_angle		
kart_to_opponent1_angle		FEATURE SOCCER GAME
kart_to_opponent0_angle_diff		puck_center
kart_to_opponent1_angle_diff		kart_to_puck_direction
		kart_to_puck_angle
		kart_to_puck_angle_diff

1.3. Actions Predictions

1.3.1 Player for Imitation Learning Performance. After saving the resulting model in the format required, pt. It was tested against the other agents, including AI. Several games with different games were manually played and though the results were not outstanding, a goal was achieved every intermittently. This is an impressive achievement when taking into account the simple network and data creation. It is easy to assume that more data, more epochs and/or a better Oracle agent.could create a pretty competitive model.

1.3.2 RESULTS. Phase Imitation Learning. In preparation for submission, tests with local grader were performed and got 25/100. During online grading the model also scored a goal but 18/100 as proof shown in Figure 2 below.

```
Loading assignment Loading grader * Match against Instructor/TA's agents - geoffrey agent [ 1 goals
scored in 4 games (1:1 0:0 0:0 0:0) ] - jurgen agent [ 1 goals scored in 4 games (0:1 0:3 1:2 0:2) ] -
yann agent [ 1 goals scored in 4 games (1:2 0:0 0:0 0:3) ] - yoshua agent [ 0 goals scored in 4
games (0:0 0:0 0:0 0:0) ] -------------------------------------------------- [ 18 / 100 ] total score 18 / 100
```
Figure 2. Goal Canvas results

2. REINFORCE simple

2.1. Data Generation

In this algorithm, this step is the one that consumes the most compute. The reason being a new dataset is generated on every epoch. Each of these new datasets was created with the calculated gradient generated using the dataset generated in the immediate previous step. The reason for this dataset re-generation is two elements in the system are not differentiable so not optimizable and need to be **sampled** instead from new generated distributions; these elements are the environment and the policy. A deeper explanation for this is in the following Architecture section.

2.2. Learning

2.2.1 Architecture. This phase uses the REINFORCE simple algorithm . For this project, a model-free algorithm is used since there is no formal modeling for the environment. This algorithm will try to learn by having a custom **agent** interact in the environment to build a policy, or model, that will predict how an agent can behave in such an environment.

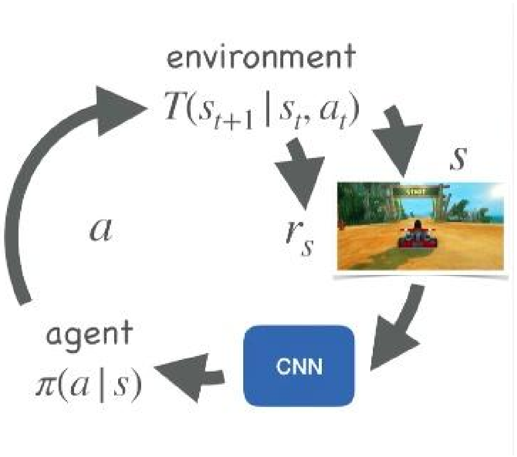
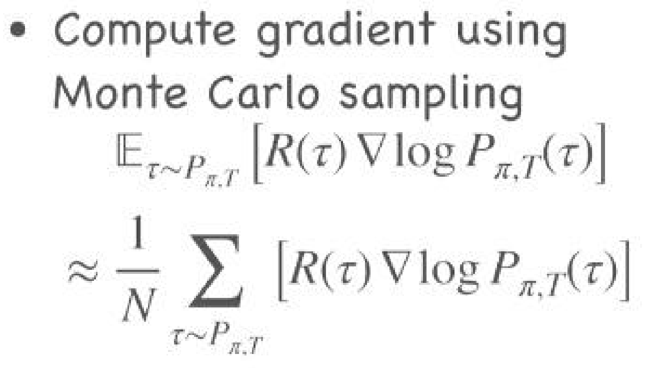


Figure 3. The Reinforcement Learning Algorithm

The Network used is a simple Linear Layer to produce the actions, similar to the sample solution presented in lectures. For this second phase, this small network was necessary as well since the purpose was to benchmark against the Imitation Learning phase so all three phases needed to be the same network. Also, this specific algorithm requires the creation of many instances of such a network so a very deep network would have been very compute expensive.

2.2.2 Non-differentiability. Looking at Fig 3, specifically for this project, it is impossible to do optimization steps to calculate the gradients in a couple of places.
The two places are:

- **2.2.2.1 Environment**. For this game, no mathematical model of the environment is provided, if it even exists, so it is a black box and gradients can not be calculated. According to the lectures, the trick here is to get the gradient of the Expected Value of the rewards without having to back-propagate through the environment. Using the **log-derivative** as detailed in Simple Statistical Gradient (Williams, 1992) Fig 4. Inspired by this paper, because an infinite number of rollouts can't be done to get the real Expected Value, a sample of those is done instead using Monte Carlo.



- Compute gradient using Monte Carlo sampling

$$\mathbb{E}_{\tau \sim P_{\pi,T}}\left[R(\tau)\nabla \log P_{\pi,T}(\tau)\right]$$

$$\approx \frac{1}{N}\sum_{\tau \sim P_{\pi,T}}\left[R(\tau)\nabla \log P_{\pi,T}(\tau)\right]$$

Figure 4. Trick to compute grading for the environment.

In this project, sample code in Fig 5 below, this process was <u>quite compute intensive</u> and was implemented by:

- Sampled rollouts by having the hand created Actor class (agent) play different rollouts (also known by trajectories/games).

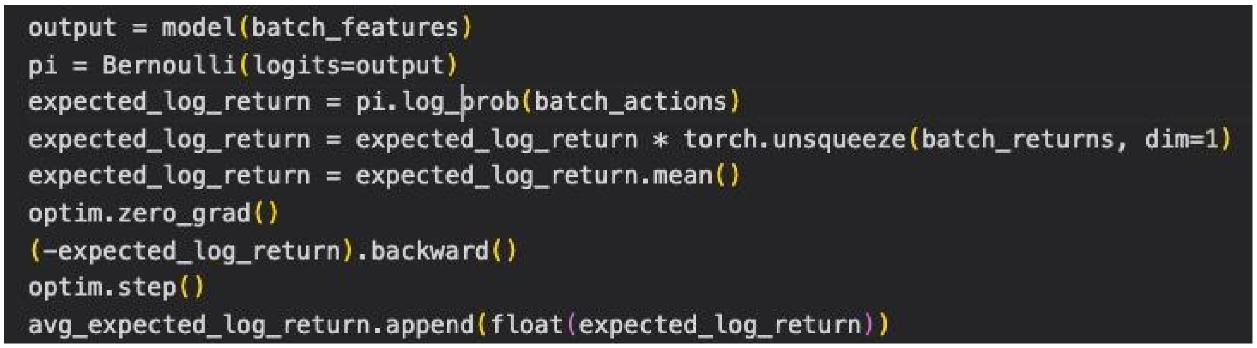
- Computed the returns of each rollout.

- Computed the gradient of the policy evaluated at the current epoch's parameters.

- Get the weighted average of all rollouts. The returns weigh the gradients. This creates greater weights for rollouts with greater reward (score in this case).

- Used these new parameters for the next epoch and created a new batch of rollouts to generate the new train data for that current epoch and calculate new parameters.

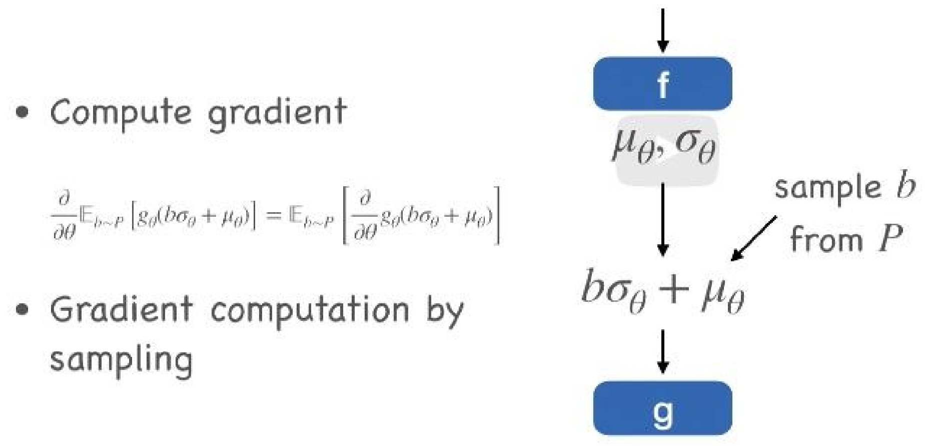
- Exact hyper parameters used in all the experiments listed further below.



```
output = model(batch_features)
pi = Bernoulli(logits=output)
expected_log_return = pi.log_prob(batch_actions)
expected_log_return = expected_log_return * torch.unsqueeze(batch_returns, dim=1)
expected_log_return = expected_log_return.mean()
optim.zero_grad()
(-expected_log_return).backward()
optim.step()
avg_expected_log_return.append(float(expected_log_return))
```

Figure 5. Section of code used to run each Expected Value of the Returns of the Environment for all rollouts. Optimization torch.optim.Adam

- **2.2.2.2. Agent's Policy/Action**. The way to select an action from a policy is by sampling the probability distribution of the policy. Sampling is not differentiable. The way to overcome this is using the Reparametrization trick, inspired by Auto-Encoding Variational (Bayes at al, 2014), that uses sampling to calculate the needed gradient. This trick works only for continuous variables but there is a trick for discrete ones like for this project's action 'break'. This trick is inspired by Gumbel-Softmax (Jang et al, 2016) and shown in Fig 6 below.



- Compute gradient

$$\frac{\partial}{\partial\theta}\mathbb{E}_{b\sim P}\left[g_\theta(b\sigma_\theta+\mu_\theta)\right]=\mathbb{E}_{b\sim P}\left[\frac{\partial}{\partial\theta}g_\theta(b\sigma_\theta+\mu_\theta)\right]$$

- Gradient computation by sampling

Figure 6. Grading computation by sampling

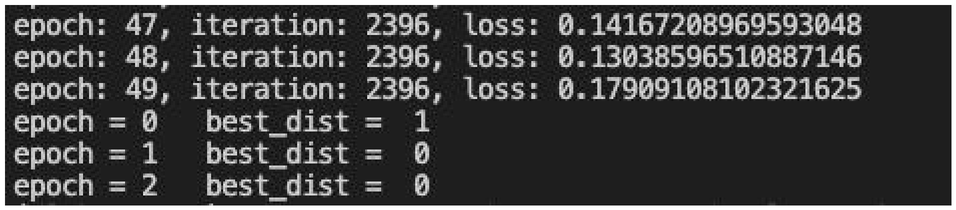
Steps coded for this project:

- Get the outputs of the model being calculated in a step (or epoch),
- Create a distribution from those outputs.
- Sample from such distribution and adjust if necessary (like convert to boolean for break action).
- Output those values.

2.2.3 REINFORCE Elements. The following elements are used in this algorithm and some were tuned to do different experiments in addition to the normal hyperparameters:

- <u>Actions</u> {Acceleration continuous[0..1], Steer continuous[-1..1], Break discrete[T/F]}
- <u>Rewards Function</u>. Goal Score of the current team in one game (also called rollout). For this phase in the project, only one reward function was used. In these experiments it became very clear that this reward function was not enough. The amount of rollouts with reward=0 was quite unbalanced so the algorithm didn't really converge. In fact, it got worse with every epoch, even with different initialization protocols described next.

- Best initialization. According to the lectures, it is critical to have good network initialization values. Two experiments were tried:
    - Run several rollouts. The lectures' example creates many Networks with many Agents doing many games/rollouts. The one with the highest reward, goal score in this case, was chosen. Unfortunately, most total rewards were 0 and the compute was quite high.
    - Use Imitation Learning model. Trained an imitation model first to get a somewhat successful one and then used that model as an initialization model for the REINFORCE part. The results were very disappointing, not only the model didn't improve but it actually got worse with each epoch. Fig 7 below.



```
epoch: 47, iteration: 2396, loss: 0.14167208969593048
epoch: 48, iteration: 2396, loss: 0.13038596510887146
epoch: 49, iteration: 2396, loss: 0.17909108102321625
epoch = 0   best_dist =  1
epoch = 1   best_dist =  0
epoch = 2   best_dist =  0
```

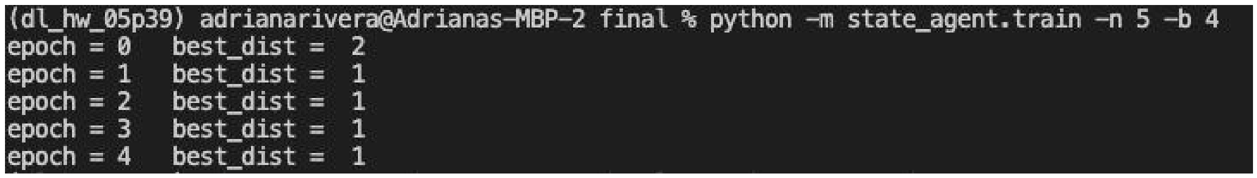
Figure 7. Training first an Imitation Learning Model as initialization to the REINFORCE training. Notice there was a score and then the model devolves by not scoring in subsequent training.

- Parameters: 10 rollouts per epoch. Each rollout was tested against one of the provided agents. Each rollout had its own instance of Actor (agent). Match instance was singleton for the entire run. Adapted provided code to create own rollout/match algorithm to be able to execute this rollouts. Optimization: torch.optim.Adam, batch_size 8 and lr=0.001. Num epoch = 6 since even though it was fast the algorithm returned fast the computer was perceived struggling and raising its temperature.

- Adjustments. Adjusting being made if time allowed: inspect the sampling methods, create a better Reward Function (angle to goal, angle to puck, distance to puck), increase the rollout/games length but this is high compute.
- Features. Again, the ones used were very similar to the ones provided by the starting code. It was observed that all provided agents had extremely similar structures. Listed in the Imitation Learning sectiont.

2.3. Actions Predictions

2.3.1 RESULTS. Phase simple REINFORCE. None of the models created with different experiments listed above had an acceptable value. Still they were used in several games against the provided agents including AI. It behaved worse than the Phase 1 Imitation Learning models. Surprising even, that the model from Imitation Learning was used as initialization to REINFORCE and not only didn't improve but made it worse, see Fig 8 below for output.



```
(dl_hw_05p39) adrianarivera@Adrianas-MBP-2 final % python -m state_agent.train -n 5 -b 4
epoch = 0   best_dist =  2
epoch = 1   best_dist =  1
epoch = 2   best_dist =  1
epoch = 3   best_dist =  1
epoch = 4   best_dist =  1
```

Figure 8. Output from training simple REINFORCE using an Imitation Learning training for best initialization. Notice scores 2 goals and then the model progressively devolves to only scoring 1 and then nothing.

Conclusions

Outcome 1. Imitation Learning

Satisfying outcome given the simple network and data gathering process. Scored one goal in Canvas. The main work was to analyze the data produced and choose the best Oracle agent. In real life, it would be imperative to have a truly expert interacting with the environment/model.

Outcome 2. REINFORCE simple.

The creation of several initialization networks experiments were done, with the best one even getting worse while training the model. Possible adjustments to work on if time allowed: inspect the sampling methods, create a better Reward Function by including more features in it, increase the rollouts/games length and number of rollouts.

Outcome 3. REINFORCE with gradient-free optimization.

Because phase 2 was still being adjusted, this algorithm was not implemented due to time concerns but it was quite obvious that it was needed for a less compute intensive implementation. Simple REINFORCE is quite computational heavy and gradient-free optimization provides a better solution to that.

Outcome 4. Vision experiments.

This project began as image_based agent and solutions for vision were attempted. Modifying the code to generate the training data/images was successful. Once the SuperTux interactions were solved, the process was similar to the other homeworks. RL was chosen to challenge and practice more DL knowledge. The modified code for interacting with Tux, was re-used for the final state-based agent.

References

Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." *arXiv preprint arXiv:1312.6114* (2013).

Jang, Eric, Shixiang Gu, and Ben Poole. "Categorical reparameterization with gumbel-softmax." *arXiv preprint arXiv:1611.01144* (2016)..

Williams, Ronald J. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." *Machine learning* 8 (1992): 229-256.

Krähenbühl, Philipp. "CS 394 Deep Learning" *UT edx* (2023).