Final Project: SuperTuxKart Ice-Hockey

A I 394D: Deep Learning, Spring 2024, University of Texas at Austin

Group 46
27 April 2024

Christopher Mitchell, Hui Chen, Omar Meziou, Samer Najjar

Introduction and Motivation

Deep learning, as a profound advancement in artificial intelligence, provides the invaluable capability to automatically learn from experience without being explicitly programmed and self-optimizes as new experiences accumulate. In the context of gaming and game development, deep learning has the potential to greatly enhance the realism and responsiveness of non-human player behaviors. In this area, the open-source SuperTuxKart Ice-Hockey game has emerged as a prominent project and stands out in software and game development, especially in the field of deep learning. The SuperTuxKart Ice-Hockey game provides an opportunity for a wide range of computational experiments and model construction and also serves as a practical, engaging tool to illustrate principles in a context of accessible and robust machine learning.

Integrating agent-based models with reinforcement learning (RL) in the context of SuperTuxKart Ice-Hockey provides an avenue for advancing AI in games. RL, a type of machine learning approach where agents learn to make decisions by interacting with an environment, fits well with the dynamic nature of games and adapts to the changes in the environment and components. In SuperTuxKart, agents (in this case, the characters controlled by a model) can be trained using RL to optimize the strategies, better handle the object, and dynamically adjust the tactics when taking the actions of other players into account. The approach of RL not only appreciates the complexity of the games and accounts for the realism of the opponents but also provides opportunities for strategy optimization for human players and enriches the game experience.

Another strategy for training an agent to play a game is imitation learning (IL). While RL is more computationally expensive to conduct successfully, IL is a faster approach that relies on the decisions of an expert whose interactions with the environment are stored and then imitated [1]. To obtain training data, we can feed various states into the expert agent and use its subsequent actions as target behaviors for our model. We attempted at first to implement an RL model but ultimately settled on an IL model for its speed of training, as the computational needs of the RL model posed a significant impediment.

Moreover, while one option is to choose a state-based agent that is trained on the evolving state, represented numerically, of various objects in the game, another option is to use an image-based agent that trains on images of the game and learns frame-by-frame what action to take based on the inferred state of the game (inferred visually through the image). We chose to implement a state-based agent as opposed to an image-based agent because the state agent would have direct access to the numeric state of the game, and would not have to infer them from an image, which would have

added an extra learning step and thus a degree of risk. A state agent also trains faster in theory, as image processing is substantially more computationally expensive.

As such, our project is aimed at training an agent-based model using a single deep network, accounting for the states of the players' karts, the opponents' karts, and the location of the puck, to outperform the agents coded by TAs and Professor in 2 vs 2 tournaments.

Methodology

Problem overview

The challenge lies in programming these agents to effectively score, make strategic decisions in real-time, and collaborate with a teammate to outmaneuver the opposing team. Our solution, in short, is to use a state-based approach that learns state-action relationships, which enables us to capture the game state information that was not provided to the image agent.

RL operates by an agent interacting within a given space and receiving rewards, without predefined strategies for actions. Instead, the agent gains knowledge through a process similar to "trial and error". RL is characterized by a loop of actions and feedback. An agent takes actions within an environment which then responds with feedback. This feedback influences the agent's policy and prompts further actions. This loop helps the agent progress from its starting point through numerous decisions, aiming to achieve or maximize a specific goal, with similarities to learning behaviors in humans (hinting at potential evolutionary benefits).

The primary components of our project, including both the RL and IL paths, include the environment (the SuperTuxKart Ice-Hockey game itself), the agents (AI-controlled karts), the policy (the strategy the agents use to make decisions), and the reward function (the criteria for success and improvement).

- **Environment**: SuperTuxKart Ice-Hockey provides a dynamic and complex environment. This is characterized by the unpredictable agent movements and fluctuation of the puck trajectory. The opponent agents often present diverse behavioral patterns making it more difficult to anticipate their next move. Fluctuation of the puck trajectory also adds a significant layer of complexity to the environment. Since the objective of the game is to score as many goals with the puck as possible, its movement greatly impacts our agents' decisions and movements within the environment. Furthermore, there is also complexity which resides in the dynamic nature of the arena as well. From arena layout to obstacles to opponent difficulty, each round of the game presents unique challenges and opportunities necessitating strategic decision-making and the ability to adapt to constantly changing scenarios.
- **Agents**: In our project, the agents are the AI-controlled karts tasked with racing against the Professor and TA agents. Each agent's behavior is dictated by a neural network that decides actions based on input states from the game.

- **Policy**: As an algorithm that the agents use to determine the best actions based on the current state of the environment, the policy components involved in this project include velocity (speed + direction vectorized for 3 dimensional space), other actions, etc, to maximize the performance in games.
- **Reward Function**: This function measures how well an agent is performing. In the SuperTuxKart context, rewards are given for winning against opponents. The design of the reward function is critical as it directly influences learning outcomes by guiding the agents toward more effective strategies. In our project, the reward function was developed, but due to challenges in the implementation, it was ultimately not used. Despite this setback, we were still able to gain insights into our approach from this process and re-evaluate our strategy for agent development. While having a reward function is a critical aspect of a typical RL project, our focus on imitation learning allowed us to deviate from a reward system by primarily relying on expert demonstrations and capturing that into the overall learning process.

By leveraging these components, our project aims to harness the power of RL and/or IL to create intelligent, competitive players. These agents are expected to not only compete effectively in the game but also continuously learn and adapt, improving their strategies based on the outcomes of their actions and the shifting dynamics of the game environment.

Dataset

SuperTuxKart features an AI for players to compete against, and there were a number of additional human-trained or designed agents provided to compete against with their own unique emphases in targeting and strategy. To obtain training data, we played each of these agents against each other, and saved the state of each frame of the match into a structured format. The states from all frames of a single match comprise a single training example. The state data contains information about the puck, the two players of Team 1, the two players of Team 2, and the goal lines. This information contains data points like coordinate locations as well as the actions of the players.

Features and Targets

For a given training example, the following **features** were either given directly by the state data or computed therefrom; features involving angles were computed from the Cartesian coordinates, using basic trigonometry:

- **Given Features:**
    - **kart_front**: The location coordinates of the front of the kart.

- **kart_center**: The location coordinates of the center of the kart.
- **kart_velocity**: The velocity vector of the kart.
- **opponent_front**: The location coordinates of the front of the opponent's kart.
- **opponent_center**: The location coordinates of the center of the opponent's kart.
- **opponent_velocity**: The velocity vector of the opponent's kart.
- **puck_center**: The location of the puck.
- **their_goal_line_center**: The center of the opponent's goal line.
- **our_goal_line_center**: The center of the player's own goal line.

- **Engineered Features:**
  - **kart_direction**: A normalized vector derived from kart_front and kart_center showing the direction the kart is facing.
  - **kart_angle**: The angle of the kart with respect to some fixed axis, calculated using the arctangent of the normalized direction vector components.
  - **is_closest_to_puck**: A boolean indicating if the kart is closest to the puck relative to all teammates.
  - **kart_to_puck_direction**: A normalized vector pointing from the kart to the puck.
  - **kart_to_puck_angle**: The angle between the kart's front and the direction to the puck, calculated similarly to kart_angle.
  - **kart_to_puck_angle_difference**: The difference between kart_angle and kart_to_puck_angle, adjusted to be within a [-1, 1] range using limit_period.
  - **opponent_direction**: A normalized vector calculated from the opponent's front vector and their center, indicating the direction the opponent's kart is facing.
  - **opponent_distance**: The Euclidean distance from the puck to each opponent, used to evaluate how close each opponent is to the puck.
  - **opponent_target**: This is a predicted future position of the opponent based on their current velocity and direction, which can be used to anticipate opponent movements for strategic planning.
  - **kart_to_opponent_direction**: A normalized vector pointing from the player's kart to each opponent's kart.
  - **kart_to_opponent_angle**: The angle between the kart's facing direction and the direction to each opponent, calculated using the arctangent function.
  - **kart_to_opponent_angle_difference**: The angular difference between the kart's orientation and the direction to each opponent, adjusted to be within a [-1, 1] range using the limit_period function.

- **puck_to_goal_line_direction**: Normalized direction from the puck to the player's goal line.
- **puck_to_goal_line_angle**: The angle towards the player's goal line from the puck.
- **kart_to_goal_line_angle_difference**: The angular difference between the kart's orientation and the direction to the goal line, adjusted similarly to kart_to_puck_angle_difference.
- **distance_puck_to_our_goal**: The straight-line distance from the puck to the player's goal.
- **distance_puck_to_opponent_goal**: The distance from the puck to the opponent's goal.

The actions taken by the winning team comprise the target behavior that we are trying to model. For a given training example, the following **targets** were given by the state data:

- **acceleration**: A continuous value representing the throttle of the kart, indicating how fast the kart should attempt to move forward.
- **steer**: A continuous value that ranges from -1 to 1, where -1 represents full steering left, 1 represents full steering right, and 0 represents no steering.
- **brake**: A boolean value indicating whether the brake is being applied. This could also include reversing if the kart is not moving forward.

We generated two datasets:

1. For the first dataset, we ran all agents against each other many times and then treated the winner of each match as the target behavior (we did not save draws). This would theoretically lead to a model that imitates a mix of different patterns of winning behavior, since agents have inherently different patterns of behavior.
2. For the second dataset, we chose a specific pre-trained agent we want to imitate, named “Jurgen”, ran it against the other agents many times, and then only kept the matches in which Jurgen won; these matches were used for training, and Jurgen’s behavior was always the target. This would theoretically give us a model that imitates only Jurgen’s winning behavior. We chose Jurgen because it appeared to have the highest victory rate among the agents.

We generated matches from various starting conditions so that our model can generalize well. Starting conditions include the following:

- Initial location of the puck

- Initial velocity of the puck (magnitude and direction)

We also control the maximum number of frames each match runs for. For our final dataset, we generated matches using the following frame caps, as listed in **Table 1**.

Max frames	Number of matches generated
200	107
400	15
500	224
600	9
700	11
800	15
900	12
1000	361
1100	15
1200	233

Table 1: Frame caps used in generated training data

Each match in this dataset is a Jurgen victory. Our full dataset contained **1002 matches** comprised of a total of **832,400 frames**.

Model Architecture

The neural network architecture we settled on consists of three hidden layers. Each hidden layer is a sequence of a fully connected (Linear) layer, a batch normalization (BatchNorm1d) layer, a Rectified Linear Unit (ReLU) activation function, and a dropout layer. The first, second, and third hidden layers have 256, 128, and 64 neurons, respectively.

See the architecture diagram in **Figure 1** below:

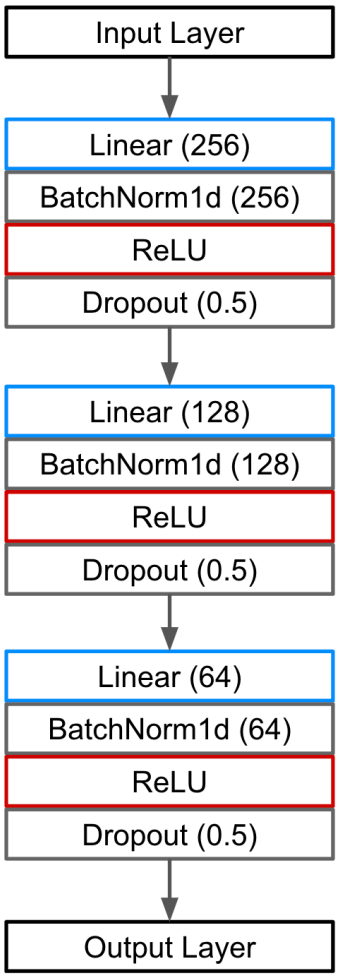

Figure 1: Model architecture

Rationale:

- The fully-connected linear layers are a natural choice for a deep learning task such as training a state-based agent, as opposed to a computer vision task which would require convolutional layers. The first layer, being the largest, captures a broad range of features, which are then progressively refined by the subsequent layers, enabling the model to learn higher-level strategies.
- The batch normalization layers ensure more stable gradients, and help avoid the vanishing/exploding gradient problem. This allows faster and more reliable convergence [2].
- ReLU activation functions were needed to introduce non-linearity into the model, allowing it to learn more complex behaviors, which is appropriate for a gamified decision-making task such as this one.

- To avoid overfitting and help the model generalize, each hidden layer includes a dropout regularization, set with a probability of 0.5 to temporarily deactivate random neurons during training.

Multiple numbers of neurons were tested; while using fewer neurons (128–64–32) trained faster, it resulted in a model that was less able to model complex decision-making behaviors. Ultimately the choice of 256–128–32 struck a balance between training time and model sophistication.

In addition, multiple levels of dropout were tested; a dropout of 0.3 was also reasonably performant, but it did not generalize as well compared to our current model. A dropout level of 0.5 seemed to strike the optimal balance, as anything higher than that took unfeasibly long to train or simply did not converge to an effective model.

Training and Testing

One of the main challenges in our project revolved around developing a state agent that would be able to navigate the intricacies and obstacles in the game environment efficiently. With this challenge in mind, we researched a variety of potential approaches that have been successful in automating game player behavior. Primarily, we focused on reinforcement learning and imitation learning as the potential development paths most likely to render results given the game's constraints and complexity in implementation.

During the initial phases of the project, we dedicated a significant amount of time to employing the Proximal Policy Optimization (PPO) model [3]. This option initially seemed to be a very reasonable and reliable approach to training the model, as it is used quite commonly in RL. The PPO implementations that we examined used two models in conjunction, one dubbed the "Actor" and the other, "Critic" [4]. Due to the constraints of needing to use a single unified deep learning model, we combined the two models into a single model with two neural networks. Both networks shared some components, specifically a series of linear and ReLU layers to process the same inputs. The actor part of the network outputs action logits for each possible action. These logits were then transformed into actual actions, which could be probabilities for boolean actions or actual values for continuous actions. The critic, on the other hand, returned a scalar state value representing the success of the current policy. This value was used to calculate the advantage function during training to update the policy in future epochs. The forward pass called both networks independently, and returned the state value from the critic and three sets of values for the actions (sigmoid for acceleration, tanh for steering, and boolean for braking, firing, and jumping).

The training loop of the PPO/Actor-critic approach used training states from stored games. These states were passed in a preprocessor (the same one used by the live

player) to extract the features needed by the model. These inputs were then passed into the model to accumulate log probabilities for all of the actions. The probabilities were stored in "Experience" objects that were captures of the log probabilities of each epoch's application of the current policy in temporal order, the summed reward calculated from the next step, the critic's state value, and the actual action that was taken in the game (as the predictive label).

However as we continued to tune and develop, we observed that the PPO model was not performing sufficiently and that a new approach needed to be taken. This may have been due to a lack of initial data (we started training on a dozen 2000 frame matches). The main difficulty here was tuning the reward function so that the policy gradient learned to weigh winning behaviors over losing strategies. The initial batch of rewards grew in complexity as we attempted to aggregate different variables at new stages of gameplay. Some examples include the current score (quickly abandoned); proximity to a target (either an opponent kart, the puck, or the goal); logic for determining the best target between the two players (switching one player from offense to defense based on the current context); rewards for possession of the puck; and whether the kart was pointed in the right direction (directly towards the puck), velocity; and absolute distance from the center of the field. These complex extra features caused the input set to balloon and become unwieldy especially when calculating loss. It was during this process that we decided to simplify the loss calculation to aggregate across all forward pass outputs instead of separating them by action data type (using MSELoss, instead of an ensemble of MSE and BCELoss).

To address this issue, we decided to pivot towards a dataset aggregation (DAgger) approach, that aims to iteratively refine the agent’s current policy by aggregating data points from expert demonstrations along with its own experiences [5], [6]. This meant we were switching from a RL to an IL strategy. After implementing DAgger, we observed significant improvement in our model’s performance. This improvement is a direct result of DAgger’s more robust learning process, as it provided a feedback loop that ultimately allowed the agent to generalize and adapt to the nuances of the game environment much more effectively. The additional support of the expert policy in the DAgger approach proved to be instrumental in overcoming the challenges posed by limited training data in our project.

In order to implement imitation learning, we had to harness a dataset of expert actions, as mentioned earlier. We tried two approaches:

- **Approach 1:** We ran all agents against each other many times, and for each match that isn’t a draw, we used the winner’s actions as the target behavior.

- **Approach 2:** By observing how the agents performed against each other, we determined the “best” agent, i.e. the agent with the highest win rate (in our case, the Jurgen agent). Then, we ran Jurgen against the other agents many times, and for each match Jurgen won, we treated Jurgen’s actions as the target behavior.

We initially experimented with Approach 1, but after some time it seemed there was a fairly low upper limit to how well the model could perform. We hypothesized that this limitation arose because different agents display distinct patterns of successful behavior, which are only effective within their specific playing strategies. Attempting to combine these divergent successful behaviors from various strategies likely leads to a conflicting mix. While these behaviors might be effective individually, when merged, they interfere with each other, resulting in a counterproductive outcome. In other words, a given agent’s “winning behavior” is only consistent and thus effective within that particular agent’s strategic framework.

This inspired us to try a more focused approach, hence Approach 2. After determining that Jurgen was the most successful agent on average, we applied the DAgger approach to data in which Jurgen was the sole victor. By training on only Jurgen's winning behaviors, we aimed to create a model with a coherent playing style, avoiding the potential strategy confusion seen in Approach 1. This strategy concentrates on imitating the successful tactics of a relatively high-performing agent, potentially leading to a more optimized model for high-performance gameplay.

For both approaches, when generating training data, we initially focused on generating matches with a maximum of 200 frames per match (this is relatively low; see Figure 1). The rationale here was that if we train on behavior that was able to reach a victory within only 200 frames, then we would be modeling fast-winning behavior, i.e. behavior that has a high concentration of winning behavior per frame, at least *in theory*.

In practice, it turned out that our agent would give up after about 200 frames of gameplay and cease to play the game sensibly afterward. It became clear that we could not simply expect the agent to extrapolate the “high-concentration” winning behavior beyond the match length it was trained on. Thus, we diversified our dataset to include frame caps up to 1200 frames. This ended up producing a significantly more performant model.

To implement DAgger, we employed a beta decay procedure whereby the model’s own actions would be injected into the training data with increasing concentration as the training evolved, with pseudocode described in **Figure 2**.

```
Initialize beta to 1.0  # start with full reliance on expert data
Set beta_decay to 0.995  # define the decay rate for beta

For each epoch of training:
    For each sample in the training data:
        Generate a random number between 0 and 1

        If random number < beta:
            Use the expert's action for this sample
        Else:
            Use the model's predicted action for this sample

    # Decay beta to gradually rely more on the model's decisions
    Update beta: beta = beta * beta_decay

Continue until beta is sufficiently small or number of epochs is reached

```

Figure 2: Pseudocode for DAgger procedure

Results

In this project, we have tried several strategies to implement imitation learning and finally chose Approach 2 (as described in the **Methodology** section) as the final model. This approach, after training using the generated dataset, yields a fair score (38/100) in our local test and the final score in the Canvas system was 57/100 (**Table 2**). **Figure 3** and the associated link provides an example of the 2 vs 2 tournament.

	Geoffrey agent	Jurgen agent	Yann agent	Yoshua agent
Game 1	1:1	0:3	0:0	0:0
Game 2	0:0	0:1	0:0	0:0
Game 3	3:0	0:1	0:1	0:0
Game 4	0:0	1:2	2:1	2:1
Total goals scored	4:1	1:7	2:2	2:1

Table 2: Goals scored against the TA agents in Canvas (our score : opponent score)

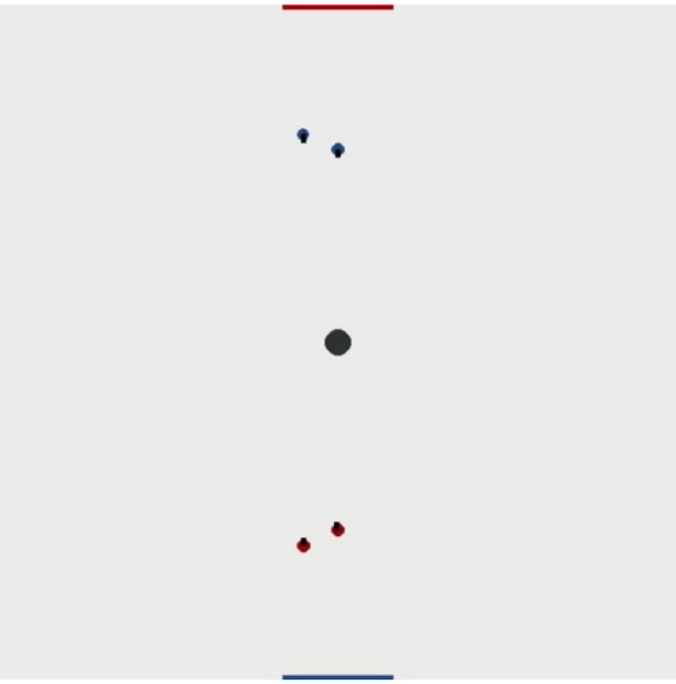

Figure 3. An example of 2 vs 2 tournament (video available at: https://github.com/SamerN88/AI-394D--Final-Project--Group-46)

Note that our model lost every match against Jurgen and could only score once in four games, while Jurgen scored seven times total. Considering that our model was trained to imitate Jurgen, this is consistent with our theory that Jurgen's performance is an upper limit on our model's performance. Moreover, seeing that our model won against two and drew against one of the other agents (besides Jurgen), it seems we made the right choice to imitate Jurgen.

Using a learning rate scheduler, the following loss evolutions were produced by our training and validation procedure (first 66 epochs only):

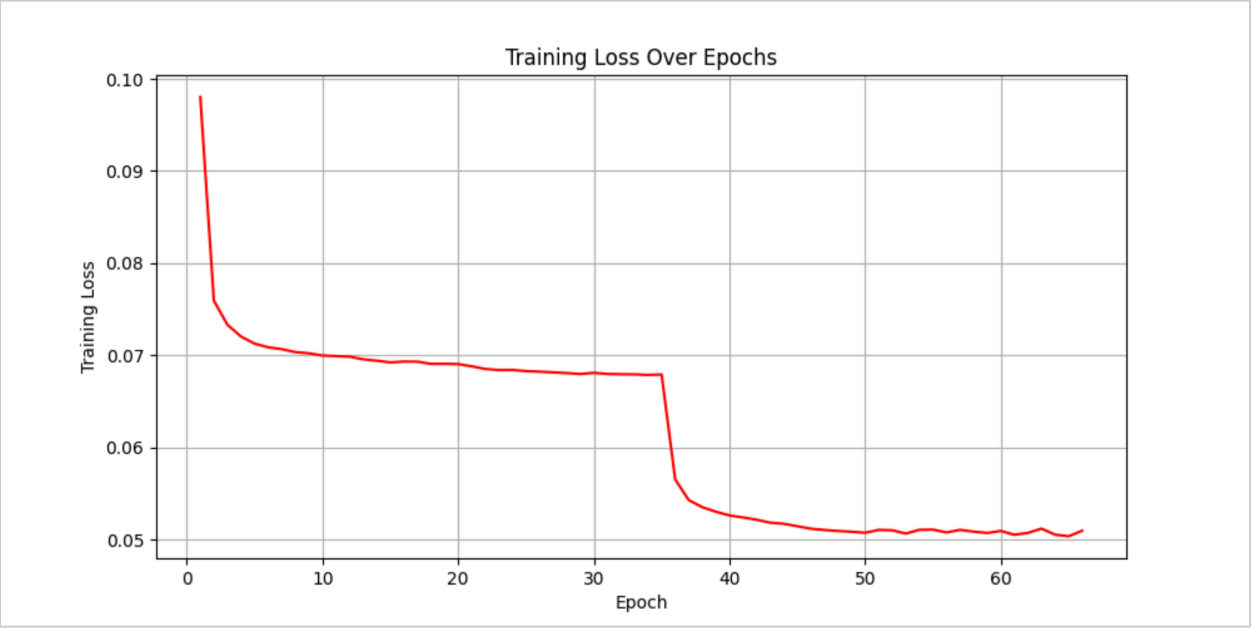

Figure 4: Training loss over epochs (first 66 epochs only)

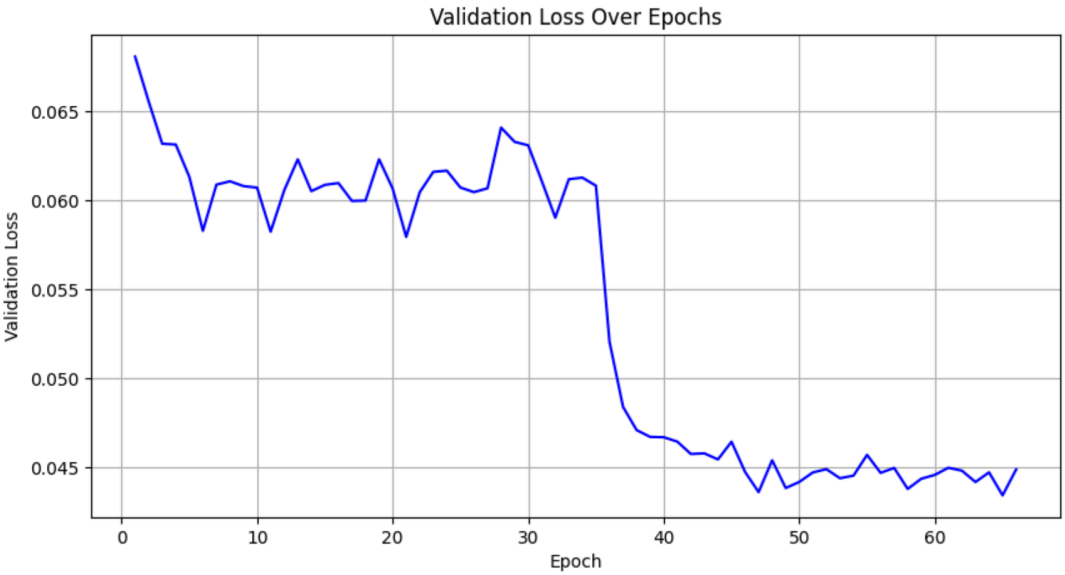

Figure 5: Validation loss over epochs (first 66 epochs only)

In the **Conclusions and Discussions** section, we discuss alternative methods that may potentially improve our current model, which we would like to explore in the future given adequate time and resources.

Conclusions and Discussions

Conclusions

Our final state-based model, trained using DAgger imitation learning, performed moderately when competing with the agents developed by the Professor and TAs. One obstacle to improving the model within this training framework is that there is a theoretical upper limit on the model's performance, defined by the Jurgen agent since that is the expert agent we imitated. One may naively hope, as we did, that we could derive some kind of synergy from imitating multiple different winning strategies from multiple agents, but as explained in our methodology and also confirmed by implementation, this only confused the model. This was overcome by choosing instead to imitate a single expert agent, chosen for its relatively higher win-rate among the given agents. And even with imitating a single agent, the model could only perform so well. Here we conclude that there is a natural limit to how well an IL-based agent could perform, at least using our approach.

Another obstacle was a difficulty for the model to generalize well. We had anticipated the problem of generalization and therefore tried to generate diverse training data by systematically shifting the starting conditions (position and velocity of the puck) within certain bounds that we considered realistic, and then running all agents on those conditions before moving to the next set of conditions. However, when faced with slightly different conditions that were not exactly accounted for in our systematic conditions-generator, the model quickly regressed and the loss stagnated earlier than expected. To overcome this, we modified our data generation strategy to use completely randomized starting conditions for each match that was generated, rather than fixing a set of starting conditions and running all agents on it. Here we conclude that the diversity of the training data is imperative, and one should not expect the model to extrapolate intuitively the way a human might.

Despite the marginal improvements derived from better data generation, a different and likely more advanced training approach is needed to achieve a highly performant model. In a future iteration, shifting to an RL approach, or a more sophisticated variant of IL, as discussed below could offer better avenues for the model to benefit from.

Potential Improvements

One way around the upper limit problem would have been to implement reinforcement learning correctly, e.g. using the PPO model we cited, given sufficient time and/or

computational resources. Since RL learns through a vast number of trials, exploring a broad range of actions not restricted to the behaviors of a fixed expert agent, without suffering from compounding errors often seen in IL methods like behavioral cloning, where deviations from optimal behavior can accumulate rapidly due to direct imitation.

Another way to incorporate RL to enhance our model's performance is via a method called Imitation Bootstrapped Reinforcement Learning (IBRL) [7]. This approach attempts to combine the strengths of both RL and IL to optimize the learning process. Initially, an UL policy is developed using expert demonstrations (like how we did) to establish a robust baseline target behavior. Then during RL training, this IL policy is employed in two key phases to enhance learning efficiency and decision-making quality:

1. **Actor Proposal Phase:** First, both IL policy and RL policy propose actions at each decision point. The action with the higher expected reward, as evaluated by the Q-network that predicts the quality of actions given the current state, is executed.

2. **Bootstrap Proposal Phase:** Then, the IL policy contributes to the training of the Q-network by proposing alternative actions for computing bootstrapping targets, which helps accelerate the convergence of the RL policy.

This strategy aids in overcoming exploration challenges in sparse reward environments, where naive and/or random action might not lead to successful outcomes. By integrating these expert-driven suggestions into the RL procedure, IBRL enhances the traditional RL process by ensuring faster learning and potentially superior performance by balancing exploration of the action space with exploitation of expert actions.

If we want to remain strictly within the IL framework, another way to improve our model would be to incorporate Human-Gated DAgger (HG-DAgger), a refined version of DAgger where a human expert dynamically intervenes during critical mistakes made by the learner [8]. Unlike our current implementation, which uses a predetermined decay rate (beta) to linearly reduce expert intervention over time, HG-DAgger adjusts based on the learner's performance at any point in the training procedure. This method potentially reduces the occurrence of uncorrected errors when the beta value has decreased.

Furthermore, given the potential scalability issues of continuous human oversight, we can further consider the possibility of an automated variant of HG-DAgger, in which we invoke an automated expert to replace the human. In this variant, we could define a criteria (with a sufficient degree of sophistication) for determining whether the learner is making a mistake that warrants expert intervention; where the human would normally intervene in HG-DAgger, our automated expert agent would instead intervene, in theory. The automated expert may be a programmatic agent, or even an AI-driven agent, depending on the complexity of the problem.

References

[1] J. Xing, A. Romero, L. Bauersfeld, D. Scaramuzza, “Bootstrapping Reinforcement Learning with Imitation for Vision-Based Agile Flight,” 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.12203

[2] J. Bjorck, C. Gomes, B. Selman, K. Q. Weinberger, “Understanding Batch Normalization,” 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1806.02375

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, “Proximal Policy Optimization Algorithms,” 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1707.06347

[4] Henanmemeda, “3.ppo.ipynb,” GitHub repository, 2021. [Online]. Available: https://github.com/henanmemeda/RL-Adventure-2/blob/master/3.ppo.ipynb

[5] N. Al-Naami, “Imitation Learning using Reward-Guided DAgger,” M.S. thesis, School of Elect, Eng. and Comp. Sci., KTH Royal Inst. of Tech., Stockholm, Sweden, 2020. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1508210/FULLTEXT01.pdf

[6] S. Ross, G. J. Gordon, J. Andrew Bagnell, “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning,” 2011. [Online]. Available: https://doi.org/10.48550/arXiv.1011.0686

[7] H. Hu, S. Mirchandani, D. Sadigh, “Imitation Bootstrapped Reinforcement Learning,” 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2311.02198

[8] M. Kelly, C. Sidrane, K. Driggs-Campbell, M. J. Kochenderfer, “HG-DAgger: Interactive Imitation Learning with Human Experts,” 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1810.02890