Imitation Learning and Self Play Feasibility in SuperTuxKart

T. Li, K. Siva, E. Lee
Group 39
CS 394D: Deep Learning

PySuperTuxKart is a kart racing game intended to be used for training AI in sensor-based tasks. This paper explores our group's approach and findings in developing a model competitive against other AI opponents in the 2v2 Supertux soccer match. Our project's overall idea was to combine imitation learning and self-play to create a competitive AI. Imitation learning would be used to boost base model performance to the best starter agent. Once models were performing on-par, we would run self-play to improve performance above the limits of imitation learning. This paper records our approach and findings with respect to the topic.

Motivation

We had two questions in mind when developing this project: Can we use self-play along with imitation learning to achieve strong results? Are there specific model architectures that are more conducive to learning in a self-play environment?

To answer these questions, we decided to develop a self-play loop loosely inspired by the training environment used by AlphaStar (Vinyals et. al). One point the paper made was that pre-initialized models could potentially improve model efficiency. As such, we decided to develop three separate model architectures that would be pre-trained before being placed in the self-play loop. These models would be trained using their own separate imitation learning methods. The models would then be placed into a self-play arena to train against legacy models and their current selves. As explained below, we were unable to complete the latter half of the objective.

Methodology

Environment Approach and Challenges

On top of the provided code for generating off-policy trajectories, we realized that we would need a way to generate on-policy trajectories. As such, we developed a gym environment capable of responding in the same way as the Match class. Termination happened at either a goal or 1000 time steps. The reward was the score. The state and actions were the same form as in the act method inside the Team class. In order to simplify the training environments, we decided to only use state-based actors. One major limitation of this environment was the lack of parallelism, which drastically lowered self-play speed.

To integrate the environment into the starter code, the training loop utilized the TeamRunner classes to ensure that environment performance would match performance on the Grader. This was unexpectedly time consuming as there were many edge cases that caused our models to deviate from expected performance. Ultimately, we were forced to settle to only have a self-play trainer with random enemy selection.

Model Approaches and Challenges

Our team implemented three different model architectures to investigate their effectiveness in self-play and imitation learning. As discussed below, all models except one had trouble training properly. The following sections detail their specific approaches and what is suspected to have gone wrong.

Imitation with DAgger

The first model that our team made was first trained strictly with basic imitation learning and then further fine tuned using DAgger. The model that we aimed to imitate was the Jurgen model as it was the highest scoring model of the ones that were provided. For collecting the training data, we had Jurgen compete against all of the other models, including the AI agent, for a single goal, with random ball starting positions, and a random velocity for the ball. We generated 600 games in total in which Jurgen was Team 1 in 300 games and Team 2 for the other 300 games. This dataset contained trajectories from the Jurgen agent which contained individual states of the players at each frame along with the corresponding action that the agent would take given that state. Since we were imitating Jurgen, the only actions we considered were acceleration, steering, and braking. Furthermore, we used the Mean Squared Error when calculating the loss for acceleration and steering since these values range from [0,1] and [-1,1], respectively. Finally, we used Binary Cross Entropy with Logits when calculating the loss for brake as this is a binary action of zero or one. Using these state-action pairs, we trained an initial state agent until we reached a relatively high score when competing against that other agents. We also generated a validation set which consisted of 120 games total, 60 for each team position. Using this newly trained model, we competed against the other agents in order to generate data for DAgger training which will utilize the states of our state-based agent. We again generated 600 games for training and 120 for validation. We then fed these states into the Jurgen Agent in order to collect the action that Jurgen would have performed given those states in order to collect DAgger state-action pairs. The goal of using this data for training is to help our model understand what actions to perform in order to correct its trajectory if it ever deviates from the behavior that Jurgen would have taken in a game. This way, we can prevent the mistakes that our agent makes from compounding and have a better performance overall.

Imitation using LSTM

Our team implemented imitation learning using the LSTM model. Initially, we need to gather data from expert demonstrations. For this purpose, we use Jurgen to compete against itself or other agents, and then we collect the data from these matches to ensure that our agent learns optimal actions. Given that actions in the game often depend on a sequence of previous states and actions, we chose the LSTM model. Its notable ability to handle temporal dependencies makes it an ideal choice for our game, as it requires the consideration of a sequence of prior states and actions. To prevent overfitting, we have opted to use a single-layer and a 32 hidden size LSTM as the starting point.

Initially, the complete trajectory of each match was treated as a single input during the training process. Despite this approach, the model's output did not yield the expected results. The agent's response time was delayed, and the executed actions did not match the desired

accuracy. Consequently, an attempt was made to enhance the model's performance by applying a sequence of timestep actions encompassing a frame of 100 states within the complete match, another attempt is by increasing the number of layers in the LSTM. However, those modifications did not produce a significant improvement in accuracy nor did it effectively address the latency issue. By recording multiple match videos throughout the training process, the actions that the agent made are almost the same compared to the ones at the very beginning, which indicates that the training is not properly performed.

When implementing the LSTM-based model, we encountered a gradient explosion at the beginning, which was resolved after applying 'clip_grad_norm_'. The addition of a time frame for trajectories resulted in other issues, where the brake loss converged to 0 right after a few epochs of training, and the steering loss fluctuates in the range of 0 to 2.5 with no evident decrease in the overall trend. We tested the agent by playing against AI from various team positions, but it failed in all the matches. Based on the recorded video, the agent's speed is slower than the opponent, most of the actions are inaccurate, and it takes a long time for the agent to perform the next actions. These factors cause the failures and we decided not to move forward with this approach.

Imitation with A2C

This model was also trained using imitation learning like the previous two models. However, this model was trained by taking the winning trajectory in any match. The model generated was intended to be used with the idea of utilizing an A2C training method during self play. In order to handle the continuous action space of steering and acceleration, the model outputs an estimation of the gaussian distribution, providing a mean and standard deviation for both. The other actions selections were binary in nature, so the model outputted a probability for how likely to activate the action. From testing, the remaining actions were limited to only estimate the brake probability. These actions were converted into probabilities using a sigmoid activation. The model also outputted a value estimate (to be used during a2c training). During imitation learning, the standard deviation output for steering and acceleration was ignored along with the value estimate. That is to say, the model outputted four tensors: the mean tensor holding steer and acceleration mean, the standard deviation tensor, the brake activation tensor, and the value estimate tensor. Importantly, the standard deviation tensor was clipped so that the lower end was above 0.

During the imitation learning phase, the model was trained using MSE loss for the accuracy and steer mean outputs and BCELossWithLogits for the brake activation. The model was trained for This model performed pretty poorly even after thousands of training episodes. The main culprit was the brake activation, which failed to converge after 50,000 states. This is suspected to be due to a high learning rate (I used the same learning rate for optimizing all the outputs). If given more time, using hyperparameter optimization using grid search might have allowed the model to converge better.

After failing to achieve a competitive model using imitation learning, we attempted to train the model using A2C (Mnih et. al). Using the gym environment mentioned above, we sampled

actions from the estimated distribution using the mean and deviation tensors. Due to inefficient code and the sequential playing of matches, the training cycle was extremely slow. On a M1 macbook, performance was approximately one minute per episode trajectory. This slow training was exacerbated due to several bugs inside the training loop which were causing trajectories to be trained off policy. After fixing all known bugs, the model was still unable to achieve a strong baseline with a training run of 500 episodes. We suspect this was due to the reward sparsity– the model learned behaviors to tie the enemy agents but not to score. This is another potential avenue of exploration for future research.

From both the imitation training and the A2C approach, the model failed to reach a competitive level. Both most likely could have succeeded if given the proper hyperparameters, but we did not have the time to properly perform a grid search. The imitation approach primarily failed due to the brake activation– the loss stayed around 0.55 and never lowered. That amount corresponded to improper brake activation on about a quarter of the frames, which deeply impacted model performance. Looking at the recorded states and videos, the model clearly did not learn to track the ball distance or direction. It performed a sort of sweeping mechanism instead, which seems to sometimes block the ball. After spending a considerable amount of time attempting to improve the model, we ultimately decided to cut our losses and focus on the DAgger model.

Results

Our results are primarily centered around the DAgger model as the other models failed to reach a competitive baseline as described above. We measured our model's performance as the number of goals scored against the various agents. Below is the process used to achieve our final score of 91 on the local grader.

In order to develop a better understanding of our model, we made recordings of our agent playing against each of the four agents. Overall, our agent displayed similar behavior against all four agents which is expected since we utilized the same extract features function that Jurgen uses in order to convert the states to usable features for our model. This version only uses the states of our players and the ball so we do not use any information on the opposing team's player states. This does seem to cause one or both of our players to occasionally stall out and not move when the ball and the opposing players are far away. Additionally, there are a couple occasions where we see all four of the players drive in circles around the ball for several seconds before adjusting their actions properly.

For the first training iteration of this model, we used a 1e-3 learning rate for the first 100 epochs and then a 1e-4 learning rate for another 50 epochs using a step learning rate scheduler. When looking at the results for this Imitation with DAgger model, the initial implementation with traditional imitation learning was able to score 82/100 on the local grader with 5, 10, 8, and 5 goals scored against the geoffrey, jurgen, yann, and yoshua agents, respectively. In comparison, the Jurgen agent, which was the target for imitation, also scored 82/100 on the local grader with 9, 5, 7, and 6 goals scored against the agents. As we can see, even though

the overall score was the same, there is a large discrepancy in the goals scored, especially against the geoffrey and jurgen agents.

In order to rectify this and further improve the performance of our agent, we further trained this model using DAgger. In this iteration, we used a 1e-3 learning rate for 50 epochs and then a 1e-4 learning rate for another 50 epochs using a step learning rate. This implementation produced significantly improved results and was able to score 91/100 on the local grader with 8, 6, 7, and 8 goals against the agents. This scoring distribution is more in line with the Jurgen agent and was able to score better than the jurgen agent as a whole.

This version of our model scored relatively low on the online grader with a score of only 56/100 and scoring 1, 1, 4, and 3 goals against the other agents. We decided to increase the size of our model and we instantly saw large improvements with just traditional imitation learning. Our agent was now scoring 100/100 on the local grader and scored 12, 11, 9, and 12 goals which is significantly more goals than the previous DAgger implementation. Unfortunately, we were not able to see improvements in score when implementing DAgger with this newer model but we were still able to score 69/100 on the online grader with 3, 0, 6, and 8 goals scored.

Conclusion

When compared to our original plans,our results were pretty lacking. While our final model reached a level that we expected, the other models were not to par. However, the project was very insightful and helped each of us gain a stronger understanding of how to create strong AI models. We had rushed into this project and focused on trying to develop complex architectures and training schemes, only to realize that the best performing model had the least amount of complexity. Spending more time solidifying the data pipeline and training environment would have been more effective rather than trying different architectures and hyperparameters. Below we detail out our plans for continuing the project as well as alternative avenues for experimentation.

Next Steps

Our project has numerous avenues for continued exploration. One area we had wanted to explore with self-play was using more advanced reinforcement learning training methodologies such as PPO (Berner et. al). If there were more time, we would have attempted to apply reinforcement learning. Regarding our best agent, it currently only predicts on acceleration, brake, and steer. Adding additional features such as nitro, could potentially improve results. The actions that we input into the model can influence the agent's output, as explained above. Without taking the opponent's state into account, the agent's actions perform poorly, especially when they are distant from the target. Therefore, we need to thoughtfully consider what inputs we should use in the next stage.

Other improvements could be done to our self play environment. As we had only created a random enemy generator, the next step would be building a proper training scheduler that would select the most difficult agents for the training model to practice against. The training loop was

also extremely slow so spending time parallelizing the training environment would drastically improve our experimentation cycles.

Should we redo the project, we would change our development focus so that only a single person works on the models while the other two work on setting up and testing the training environment. We would like to utilize the same approaches with the modifications listed in the prior sections, but if we were to approach the project from a completely different lens, we would attempt standard Actor-Critic methods using TD(λ) to attempt self play directly without using imitation learning. Alternatively, we could completely discard any idea of using reinforcement learning and attempt an evolutionary computation strategy such as Evolutionary Strategy (Bäck & Schwefel), which in of itself is an interesting approach to machine learning. We could still preserve our self-play trainer, and instead introduce noise into our model parameters in an attempt to improve our model performance. The iterations would simply be re-selecting the best performing models from the current and past generations and repeating the process. This simplistic approach might allow us to iterate faster in testing as we would be able to finish development and bug fixing in a shorter period of time.

Appendix

Our development work was performed on a private github repository. For the entire code development repository, please request access by emailing us at evanlee8858@gmail.com.

Citations

Bäck, T., & Schwefel, H. P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, *1*(1), 1-23.

Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., ... & Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928-1937). PMLR.

Vinyals, O., Babuschkin, I., Czarnecki, W.M. et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature 575, 350–354 (2019). https://doi.org/10.1038/s41586-019-1724-z