IMITATION LEARNING ON SUPERTUXKART ICE HOCKEY

CHRIS VAN BUREN

Abstract. Training autonomous agents on the actions from high performance existing policies is an active area of research in fields such as video game play and autonomous driving. Imitation learning and the DAgger algorithm train an agent based on an expert's actions. This paper explores using these methods to play ice hockey in an adaptation of the video game SuperTuxKart. An expert agent was selected from a set of available existing agents, from which game play states and actions were collected to train a new agent. The results show that the new agent was able to learn a policy that allows it to successfully play the game using imitation learning, while not meeting the performance of the expert agent. Future work includes exploring ways to incorporate sequential state information into the policy and creating a single network that outputs actions for both players on the team to encourage coordinated teamwork.

Introduction

1.1**Existing Research** Artificial intelligence, particularly in the area of policy training methods, has seen significant advancements through imitation learning and Dataset Aggregation (DAgger). These techniques have been had some success in training autonomous video game agents, which face dynamic environments and intricate decision-making processes. Games such as SuperTuxKart and MarioKart provide a suitable platform for developing and testing AI agents while also being easily observable for humans. Training agents to play these video games also has practical applications, as they are simplified, virtual environments for autonomous driving.

Imitation learning trains agents to mimic expert behavior [1], however, it can be highly dependent on the diversity of training data and can fail to generalize in unseen situations [2]. Research has been done to forward the applicability of imitation learning, such as incorporating user inputs at inference into imitation learning-trained policies [3] for autonomous driving and training agents for real-world scenarios by learning from intermediary privileged agents [4].

1.2**Motivation** This research aims to progress the development of autonomous agent training within video game environments. This study attempts to build off the existing research of learning from expert agents for driving tasks by training an agent to learn from existing autonomous agents in SuperTuxKart ice hockey, a video game described in [5] that adapts the SuperTuxKart racing game to a simplified hockey game.

Imitation learning was selected as the primary method due to its ability to capture expert behavior to train agents. However, imitation learning's limitations, such as dependency on training data diversity and challenges with generalization, necessitate an additional method

Date: May 1, 2024.
CS342.

to address these issues. DAgger [6], offering an approach that balances exploration by using a naive agent to explore the state space and exploitation, querying an expert agent to gather high quality actions, is a suitable solution to improve the generalization capabilities of the game agent.

Methods

2.1 Expert Agent Selection

Two methods of policy training for the agents in SuperTuxKart ice hockey are explored: imitation learning, as introduced in [1] and DAgger (dataset aggregation) [6].

Imitation learning and DAgger both gather actions from an existing agent to train a new agent. Existing agents, using [5], (known as TA agents) can perform moderately well at the SuperTuxKart ice hockey game, regularly scoring goals. Games were run to play each TA agent against each other TA agent (including itself) to get an idea of which TA agent performs the best - that is, scores the most goals and wins the most games. In each of these games, the puck's starting location was set to [0, 0] (center ice) and starting velocity was set to [0, 0] (static), and each team was assigned two players. The results of these games are shown in Table 1. The TA agent called jurgen_agent scored the most goals and won all 5 games it played. Therefore, it was used going forward as the expert agent in imitation learning and DAgger.

Agent	Goals Scored	Wins
geoffrey_agent	4	3
image_jurgen_agent	5	3
jurgen_agent	**10**	**5**
yann_agent	3	2
yoshua_agent	4	1

TABLE 1. The TA agents played head-to-head games against each other, with jurgen_agent scoring the most goals and winning the most games.

2.2 Imitation Learning

2.2.1 *Data Collection*

Data for imitation learning was collected by running games and recording the player states and puck state, provided by pystk [7], paired with the expert agent actions (acceleration, steer, and brake) at each step in the game. For each game played in data collection, the expert agent played against an agent randomly picked from all the TA agents, plus the AI agent provided by pystk. The coordinates for initial ball location and velocity were randomly sampled from a uniform distribution from -1 to 1. Games were played until a team scored 3 goals or up to 1000 frames. To improve the performance of the expert agent in the training data, records from games in which the expert agent did not score were discarded. In total, records from 20 games were kept as training data.

Each step in the recorded games had an associated state and actions from the expert agent. Features of each state were extracted using the jurgen_agent's `extract_featuresV2` function. The function calculates features as described below:

```
Function extract_featuresV2 with parameters pstate, soccer_state,
        opponent_state, team_id:
```

```
    // Determine features for the player's vehicle
    Assign kart_front to the front vector of kart from pstate,
            taking x and z coordinates
    Assign kart_center to the location vector of kart from
            pstate, taking x and z coordinates
    Compute kart_direction as a unit vector from kart_front
            minus kart_center
    Compute kart_angle as the arctangent of kart_direction in
            the y/x plane

    // Determine features for the soccer ball
    Assign puck_center to the location vector of ball from
            soccer_state, taking x and z coordinates
    Compute kart_to_puck_direction as a unit vector from
            puck_center minus kart_center
    Compute kart_to_puck_angle as the arctangent of
            kart_to_puck_direction in the y/x plane

    // Compute the angular difference between kart and puck
    Normalize kart_to_puck_angle_difference to the range [-1, 1]
            by dividing the angle difference (kart_angle minus
            kart_to_puck_angle) by pi

    // Determine features for the goal line
    Calculate goal_line_center as the average position of the
            goal line from soccer_state for the opposing team,
            taking x and z coordinates
    Compute puck_to_goal_line as a unit vector from
            goal_line_center minus puck_center

    // Compile the feature tensor
    Construct features tensor with elements:
        kart_center.x, kart_center.z, kart_angle,
        kart_to_puck_angle, goal_line_center.x,
                goal_line_center.z,
        kart_to_puck_angle_difference, puck_center.x,
                puck_center.z,
        puck_to_goal_line.x, puck_to_goal_line.z

    Return features tensor
End Function
```

2.2.2*Model Architecture and Training* The features extracted from this function were used as inputs to a deep neural network that predicts values for acceleration, steer, and brake actions. The architecture of the deep neural network is shown in Figure 1. The input layer had 11 neurons, one for each feature calculated in the function above. Addititional linear layers

were connected sequentially, each followed by a ReLU activation. Dropout layers (for 0.2 of activations) were used after the first and second hidden layer activations to prevent overfitting.

When training the model, default PyTorch initialization was used.

The Adam optimizer [8] was used with an initial learning rate of 0.01. The PyTorch 'ReduceLROnPlateau' scheduler was employed to adjust the learning rate based on the validation loss, reducing it by a factor of 0.1 if there was no improvement in performance for a patience period of 5 epochs.

Mean squared error (MSE) loss was used as the loss function.

The game steps from data collection were randomly shuffled, and 20% of the steps reserved for validation. A mini-batch size of 1024 was used for training and validation.

The network parameters were updated through back-propagation for each mini-batch.

The network was trained for 75 epochs, at which point the change in loss became asymptotic.

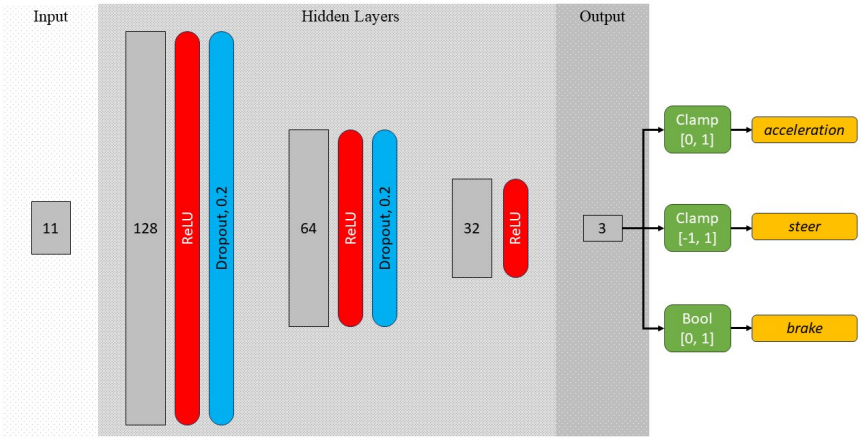

FIGURE 1. This diagram illustrates the sequential architecture of the trained state policy network, built using PyTorch. The model comprises of linear layers (gray), ReLU activations (red), and dropout layers (blue). The input layer has 11 neurons, followed by alternating fully connected layers and ReLU activation functions, with dropout layers introduced to prevent overfitting. The network culminates in an output layer with 3 neurons, corresponding actions (acceleration, steer, brake). Output values were clamped or thresholded to acceptable action values, shown in green, in the agent module, not in the network, so they did not impact training.

2.2.3 *Agent Setup* During game play, the new agent took the state and used the `extract_featuresV2` function to extract features of the current time step. For each player (the agent was evaluated on games where each team had 2 players), the features were passed to a trained network, as described above, which output three values. To ensure these values were valid pystk Player [7] actions, each of these three values was either clamped or converted to a boolean. These functions are represented by green boxes in Figure 1. The actions were assigned from the `values` output from the network as follows:

```
\\ Clamp acceleration between 0 and 1
acceleration = min(1, max(0, values[0]))

\\ Clamp steer between -1 and 1
steer = min(1, max(-1, values[1]))

\\ Convert brake to boolean 0 or 1, with a threshold of 0.5
brake = 1 if values[2] > 0.5 else 0
```

Each player on the team followed this process to perform the calculated actions at each time step.

2.3**DAgger**

2.3.1*Data Collection* To collect data for DAgger, games were run playing the agent described in the Imitation Learning section against a random choice from the collection of TA agents. At each time step in the recorded games, the game state from the perspective of the imitation agent was passed to the jurgen_agent to query its actions at that state. The game state from the perspective of the imitation agent and the hypothetical actions of the jurgen_agent were aggregated to create a DAgger dataset. In total, records from 20 games were used.

2.3.2*Model Training and Architecture* The network architecture and training process were identical to that of the Imitation Learning section, except that the change in loss became asymptotic after 50 epochs, so training was stopped then.

2.3.3*Agent Setup* The agent setup using a network trained with the DAgger dataset is identical to the agent setup in the Imitation Learning section.

2.4**Policy Combinations** Tests were done with five different policy combinations between the two players on the agent's team. First, both players used the same network, which had been trained on the imitation learning dataset. This policy is represented as $\pi^1_{imitation}$.

Next, a second network with an independent initialization, with the same architecture and training process, was trained on the imitation learning dataset. This policy is represented as $\pi^2_{imitation}$. A test was run with one player following $\pi^1_{imitation}$ and the other following $\pi^2_{imitation}$.

A test was run with both players using the same network as described in the DAgger section. This policy is represented as π^1_{dagger}.

Another network with independent initialization was trained identically. This policy is represented as π^2_{dagger}.

A test was run with one player following π^1_{dagger} and the other following $\pi^1_{imitation}$.

Finally, a test was run with one player following π^1_{dagger} and the other following π^2_{dagger}.

Each policy combination was tested by playing 8 games with random initial puck placement and team assignments (i.e., side of the rink to score on) against each of the following TA agents: geoffrey_agent, jurgen_agent, yann_agent, and yoshua_agent. The count of total goals scored by each policy combination was recorded.

Results

3.1**Game Outcomes** The goals scored by each tested policy combination are shown in 2. The policy combination of an independently initialized network for each player, each trained on the

imitation learning dataset, scored the most goals, with 18. Overall, the policies trained exclusively on the imitation learning dataset drastically outperformed any other policy combination.

The top scoring policy combination, ($\pi^1_{imitation}$, $\pi^2_{imitation}$) was used by the agent submitted to the holdout test: playing matches against unseen agents. In the holdout test, the agent scored 9 goals in 16 games—the same rate of goals per game as in initial tests (18 goals in 32 games).

The threshold of success in this work, being focused on imitating the performance of jurgen_agent, was to score with the same goals per game ratio as jurgen_agent against the TA agents. In the head-to-head tests described in Section 3.1, jurgen_agent scored 10 goals in 5 games: a rate of 2 goals per game. With the best policy combination of those created, the best goals per game ratio of the new agent is 0.5625, well below the success threshold.

Player 1 Policy	Player 2 Policy	Goals Scored
$\pi^1_{imitation}$	$\pi^1_{imitation}$	12
$\pi^1_{imitation}$	$\pi^2_{imitation}$	**18**
π^1_{dagger}	π^1_{dagger}	2
π^1_{dagger}	$\pi^1_{imitation}$	7
π^1_{dagger}	π^2_{dagger}	0

TABLE 2. The goals scored varied given different policy combinations between the two players. The highest scoring policy combination two distinct imitation policies.

FIGURE 2. The first network trained on the imitation learning dataset achieved a validation loss of 0.1414.

3.2**Algorithm Metrics** Figure 2 shows the training loss, validation loss, and learning rate over epochs for the training of $\pi^1_{imitation}$. (Reminder: mean squared error was used as the loss function.) The validation loss stayed below the training loss, likely due to the dropout layers, indicating that the model did not overfit the training data. The network achieved a validation loss of 0.1414. Figure 3 shows these metrics for the training of $\pi^2_{imitation}$, the other policy in the

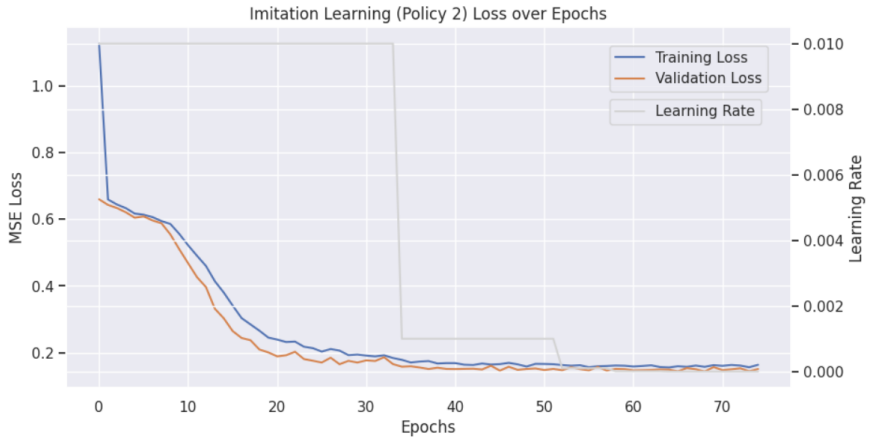

FIGURE 3. The second network trained on the imitation learning dataset achieved a validation loss of 0.1514.

best performing combination. This policy had slightly different metrics, with a final validation loss of 0.1514.

The similarity between these two graphs shows that the network trains consistently on the imitation learning dataset. For each network, the learning rate stayed at 0.01 for over 30 epochs until the scheduler dropped the learning rate by a factor of 10. Each network's learning rate dropped once more soon after the fiftieth epoch.

Conclusion

4.1**Discussion of Results** This study explored two policy training methods for agents in SuperTuxKart ice hockey: imitation learning and DAgger. The expert agent selection process identified jurgen_agent as the most adept TA agent, which was then utilized as the expert for further training. The data collection processes for imitation learning and DAgger deliberately randomized opponents and the starting position and velocity of the puck to encourage diverse training datasets. The data collection process for imitation learning kept only games in which the expert scored to increase the quality of training data.

The model architecture was designed with dropout layers to prevent overfitting and trained using the Adam optimizer with a learning rate scheduler to ensure efficient learning. The agent setup involved real-time feature extraction and action prediction and validation during gameplay, demonstrating the practical application of the trained model.

The most effective strategy among the tested policy combinations involved assigning each player an independent policy, both of which were trained using the same dataset and process for imitation learning. This approach likely improved the team's performance because it introduced a diversity of actions between the two players. With each player operating under a distinct policy, there was a reduced chance of both players converging on the same location in the rink. Instead, they were more likely to spread out and cover different areas, increasing the chances that one of them would find themselves in an optimal position to score. Essentially,

it is reasonable to posit that the independent policies help ensure that the players' actions complement rather than duplicate each other, leading to a more effective team strategy.

The poor performance of policies trained with the DAgger dataset is curious considering DAgger is typically more robust than imitation learning, as shown in [6]. It is possible that the DAgger training dataset was not large enough, or that the imitation agent used to explore the states in DAgger dataset collection did not adequately explore the state space.

4.2 Alternate Methods

4.2.1 *Improve DAgger Dataset* One possible way to improve on the poor performance from the DAgger policies is to improve the DAgger dataset. One opportunity for exploration is ensuring the state space is widely explored. Measuring the variance of the explored state space could help quantitatively improve the DAgger dataset as the exploring agent is iterated upon. A way to improve training for either DAgger or imitation learning is to add a data augmentation step to the states in the training set. This could be done by adding Guassian noise to existing states. The standard deviation of each feature in the training set could be calculated, the Gaussian noise with each respective standard deviation and mean zero could be added to each feature at each training iteration. This was initially explored but led to degraded validation loss and worsened in-game performance. Alternatively, states could be synthesized by another model, greatly expanding the diversity of observed states with little cost, which was demonstrated as a successful adaptation of DAgger, outperforming behavioral cloning, in [9]. It may also be beneficial to collect DAgger actions from more than one expert agent. As shown in [10], policies that use DAgger based on several imperfect experts can outperform the experts themselves and imitation learning policies.

4.2.2 *Train a Hybrid Policy* When training on the DAgger dataset, the network was trained from random initialization. Training a model initially on an imitation learning dataset, then continuing training on a DAgger dataset may lead to improved performance.

4.2.3 *Train an RNN on Game Sequences* One option moving forward is to capture sequential information for use in the policy. It was initially explored to build a Jordan-style recurrent neural network (RNN) [11], using the player's previous action as an extra input to the imitation learning network, but the network would not train past a few epochs, with the loss becoming NaN, possibly due to exploding gradients. Further research into taking advantage of sequential state information would likely benefit the performance of the policy.

4.2.4 *Train a Multi-Player Policy* Finally, it could be explored to create a single network that outputs actions for both players on the team in one forward pass. Such a policy could optimize for teamwork. As it is guessed that using two independent policies benefited the agent by yielding complimentary actions, a single policy that creates both of these actions and allows them to interact could learn to deliberately predict complimentary actions. It is unknown if the TA agents deliberately use teamwork in their policies, so imitation learning and DAgger using a TA agent as an expert may not be suitable for training this type of network.

References

[1] D. Pomerleau, 'ALVINN: An Autonomous Land Vehicle In a Neural Network', in Proceedings of (NeurIPS) Neural Information Processing Systems, 1989, pp. 305–313.

[2] B. Zheng, S. Verma, J. Zhou, I. Tsang, and F. Chen, 'Imitation Learning: Progress, Taxonomies and Challenges', arXiv [cs.LG]. 2022.

[3] F. Codevilla, M. Müller, A. López, V. Koltun, and A. Dosovitskiy, 'End-to-end Driving via Conditional Imitation Learning', arXiv [cs.RO]. 2018.
[4] D. Chen, B. Zhou, V. Koltun, and P. Krähenbühl, 'Learning by Cheating', arXiv [cs.RO]. 2019.
[5] 'Final project - SuperTuxKart ice hockey.' Philipp Krähenbühl [Online]. Available: https://www.philkr.net/dl_class/homework/final/. [Accessed: Apr. 23, 2024].
[6] S. Ross, G. J. Gordon, and J. A. Bagnell, 'A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning', arXiv [cs.LG]. 2011.
[7] 'Game state.' pystk [Online]. Available: https://pystk.readthedocs.io/en/latest/state.html#pystk.Player. [Accessed: Apr. 23, 2024].
[8] D. P. Kingma and J. Ba, 'Adam: A Method for Stochastic Optimization', arXiv [cs.LG]. 2017.
[9] X. Zhang, M. Chang, P. Kumar, and S. Gupta, 'Diffusion Meets DAgger: Supercharging Eye-in-hand Imitation Learning', arXiv [cs.RO]. 2024.
[10] X. Sun, S. Yang, and R. Mangharam, 'MEGA-DAgger: Imitation Learning with Multiple Imperfect Experts', arXiv [cs.LG]. 2023.
[11] M. I. Jordan, 'Serial order: a parallel distributed processing approach. Technical report, June 1985-March 1986', vol. place = United States, year = 1986, month = 5, no. volume = , place = United States, year = 1986, month = 5, 5 1986.

Address
Email address: cvanburen@utexas.edu