CS 394D Deep Learning Fall 2020: Final Project Report

Abhishek Divekar * **Jason Housman** * **Ankita Sinha** * **Alex Stoken** *

Abstract

We design an automated agent to play SuperTuxKart Ice Hockey. Ice Hockey falls under the category of a game with a large state space, mixed action space and a sparse reward, a very difficult challenge even in today's state of the art reinforcement learning. We design a system that exploits potential simplifications to this system, towards the goal of designing effective players. We train a multitask model inspired by CenterNet to assist a controller network using Branching Deep Q Learning to play ice hockey. Our two-stage system composes of a "vision" stage which takes as input the image of the player's Field of View (FOV) and predicts world-state attributes. These are consumed by a "controller" stage which return actions that update the world-state. This staged approach allowed us to optimize performance of the vision and controller portions independently as sub-teams.

1. Vision system

1.1. Data Generation

We gather images and world-states of six games played by four randomly-selected AI agents for 2000 frames per game, totalling 48,000 frames for train and test datasets each. To introduce variety in the data, we play two of the six games at AI difficulty levels (0, 1, 2). Thus, in each dataset we accumulate viewing angles for 24 different players of varying competence, in 4-player game settings, similar to the "real" tournament on which we are evaluated. Our hypothesis is that choosing random players forces the vision model to learn a "blind-spot" in the lower-middle portion of the screen where the kart is located, thus allowing the vision network to predict accurately independent of the kart type; we see the results of this in Figure 2.

*Equal contribution, ordered alphabetically by last name. UTexas Email: Abhishek Divekar <adivekar@utexas.edu>, Jason Housman <jhousman@utexas.edu>, Ankita Sinha <anki0811@utexas.edu>, Alex Stoken <astoken@utexas.edu>.

At every time step we store a pickled world-state object (Van Rossum, 2020) from which we extract ground-truths for our vision tasks. During training, we can use these to compute (p)uck to (k)art distance:

$$d = \sqrt{(x_p - x_k)^2 + (z_p - z_k)^2} \quad (1)$$

(we ignore y-component i.e. vertical height difference since it is near zero once the puck lands on the field). However, puck visibility and aimpoints in the FOV must be estimated by back-calculating the screen pixel coordinates from the puck's world-state coordinates as shown in Figure 1.

$$v_{view} = M_{view} \cdot M_{projection} \cdot v_{WorldCoordinates}$$

$$v_{NDC} = \left(\frac{V_{pixel}[0]}{V_{pixel}[3]}, \frac{V_{pixel}[1]}{V_{pixel}[3]} \right)$$

$$\text{Visibility} = \begin{cases} True & \text{if } -1 < v_{NDC}[0] < 1 \\ & and \text{ if } v_{view}[2] > 0 \\ False & \text{otherwise} \end{cases} \quad (2)$$

Figure 1. Translating world-state coordinates to pixel coordinates and computing puck visibility

World-to-View space coordinates are calculated via matrix multiplication of the view matrix, M_{view}, the projection matrix, $M_{projection}$ and ground truth world coordinates, $v_{WorldCoordinates}$, which are available in the PySTK internal player and world states. The result is a vector of view coordinates which contain localization components relative to the camera, including orientation. The orientation can be used to determine if the ball is behind the player, based on the sign of the z-coordinate. Those coordinates are then converted into a vector of normalized device coordinates v_{NDC}, which represent the orientation of an object as left (-1 to 0) or right (0 to 1) of the center of a frame (de Vries). The puck is visible if the first coordinate in the normalized device-coordinates vector is between **-1** and **1** and third coordinate of v_{view} is greater than zero.

A manual audit of 100 randomly-sampled ground-truth images shows that this data generation method produces puck visibility values with 94% accuracy (5 false positives/1 false negative), which is sufficient to train a vision system for the downstream task.

After computing the center-point of the puck, we also compute pixel coordinates that enclose the puck inside a bounding box. We do this by adding and subtracting the radius of the puck (Super-Tux-Kart-Development-Team) to the first value in the world coordinate vector, and the height to the second value of the world coordinate (which we estimated to be 0.8 Blender Units). Pixel coordinates are computed for those points. This establishes a rough estimation of the bounding box over the puck in most cases.

Similar to (Zhou et al., 2019), we used a Gaussian kernel to transform the calculated bounding boxes into a heatmap, centered at the midpoint of the bounding box, suitable for a localization task. Additionally, we generated a second heatmap to capture the puck-to-kart distance. This was done through several small tweaks to the code provided in Homework 4, whereby we take the size heatmap computed via the bounding boxes, and create a matrix of all ones of the same size. We then multiply that matrix by the computed distance value, and using the same masking procedure from the function to set all non object indexes back to zero. This heatmap will be zero everywhere, except for the image coordinates where the puck is spatially located on the screen.

Each datapoint is thus represented by a 5-tuple:

$$(\text{img}, \text{visible}, (x_{\text{puck_loc}}, y_{\text{puck_loc}}), h_{\text{loc}}, h_{\text{dist}})$$

Figure 2. Sample vision network prediction. We observed that the system is able to accurately detect the puck visibility, aimpoint and distance for various player types and game combinations.

1.2. Architecture

For our vision tasks, we use a multi-headed network inspired by CenterNet (Zhou et al., 2019) with a U-Net (Ronneberger et al., 2015) backend encoder-decoder structure[1]. We normalize the channels of the input image to the mean and standard deviation calculated over the entire training set to improve convergence during training. Our U-Net architecture consists of 4 blocks of strided convolutions, doubling channels each block (16/32/64/128). Each layer uses Batch Normalization (Ioffe & Szegedy, 2015). These are followed by 4 symmetric blocks of up-convolutions which upsample the activation maps (and halve the channels) to produce a full-resolution heatmap. In the next sections, we describe each of the tasks heads which using this heatmap.

1.2.1. Task 1: Puck Onscreen Classification

To detect whether the puck is visible in the player's current Field of View, we use global-average pooling and a single linear layer on top of the U-Net output. The addition of Dropout (Srivastava et al., 2014) provides regularization during training.

1.2.2. Task 2: Puck Aimpoint Regression

Our puck aimpoint head adds a 2D-Convolution layer with one output channel on top of the U-Net output to predict the aimpoint heatmap $\tilde{h}_{\text{loc}}$). The normalized puck aimpoint is the spatial argmax of $\tilde{h}_{loc}$

$$(\tilde{\alpha_x}, \tilde{\alpha_y}) = \underset{x,y}{\text{argmax}} = \tilde{h}_{\text{loc}}$$

where $\tilde{\alpha_x}, \tilde{\alpha_y} \in [-1, 1]$.

1.2.3. Task 3: Puck Distance Regression

Similar to the aimpoint head, the puck distance head is a 2D Convolution layer on top of the U-Net output, with a single output channel (the predicted heatmap $\tilde{h}_{dist}$) which we use to compute the distance of the puck from the kart, $\tilde{d} \geq 0$.

1.3. Training

In this section, we describe the training for each of the tasks.

1.3.1. Task 1: Puck Onscreen Classification

We perform binary classification to detect whether the puck is visible in the player's current Field of View. This task is well suited to a binary cross-entropy loss (we considered Focal Loss, but found it unnecessary as the puck visibility

[1] In particular, we use the Homework 4 solution architecture with alternate heads, including swapping out the localization heatmap with the spatial argmax functionality from Homework 5.

is evenly distributed among the two classes in both train and test data-sets).
For this task, models trained with color jitter (brightness, hue, saturation and contrast) did not generalize to the test data-sets. The ice arena has a particular color palette (light blue for ice, black for puck etc) which is consistent between the train and test domains. When the puck is visible but far away, the color contrast between the puck and the surrounding ice rink becomes essential to identification even for the human eye. We thus conjecture that color jitter during training adds unnecessary noise that is not present during test time, so our final training does not use color jitter.

1.3.2. Task 2: Puck Aimpoint Regression

The puck aimpoint is a crucial input to the controller while searching for the puck on the field; divergence of the predicted aimpoint from the actual puck aimpoint onscreen causes the kart to drive in an incorrect direction and often miss the puck entirely. We regress to the aimpoint with a Mean-Squared Error loss which penalizes by the square of divergence, providing a training signal for the predicted $(\tilde{\alpha_x}, \tilde{\alpha_y})$ to be closer to (α_x, α_y) providing a more accurate input for controller steering.

1.3.3. Task 3: Puck Distance Regression

To train the output of the distance regression, we compute the *L1 Loss* between the ground truth, h_{dist} and the predicted distance heatmap $\tilde{h}_{dist}$, without aggregation. Following that, we take the ground-truth object localization heatmap $h_{loc} \in [0, 1)^2$ and multiply it by our calculated loss. This has the effect of up-weighing the loss closer to the center of the puck higher and nullifying the rest. Finally, we aggregate by computing the mean of all non zero values in the loss matrix. The rescaling factor provides a training signal which guides our predicted aim point to be closer to the center of the object, as we want to ensure distance is as accurate as possible towards the center of the puck.

1.4. Performance

We present metrics of our final model on the full test set of 48,000 frames. On this dataset, our model achieves an MSE loss of 0.281 for puck aimpoint regression and 0.03 L1 loss for puck distance prediction.

Detecting whether the puck is onscreen can be considered the preliminary task, as the aimpoint and distance are not meaningful when the puck is off screen. For puck detection, we observe that our final model achieves a test ROC-AUC score of 0.983, which we believe is sufficient for our downstream task. As the majority of error in our ground-truth data is false-positives, we attept to counter this by biasing towards a higher detection threshold from the sigmoid output. Calculating a Precision-Recall curve shows that increasing the threshold from 0.5 to 0.9 increases precision by 0.6% (from 97.2% to 97.8%) at the cost of 0.6% recall (from 96% to 95.4%); beyond 0.9, recall drops at a rate which exceeds the increase in precision.
A single detect call to the final model requires 18 ms using an NVIDIA Tesla V100 GPU (min=6ms, max=34ms) as averages across 16,000 400x300 image detections.

2. Controller

Traditional team ice hockey has a large state space with near infinite strategies. SuperTuxKart Ice Hockey is similar in principle to this in that it also has a large very large state space and a mixed simultaneous action space. In SuperTuxKart, the action space is characterized by a continuous range of values for steering and acceleration and a binary actions for braking, drifting, nitro and rescue. These actions are correlated - for example, when braking at zero acceleration, the kart will move in reverse. We explored approaches to similar problems for inspiration, in particular RoboCup Soccer (Kitano et al., 1997). While similar in problem space, RoboCup Soccer agents are more complex given that they are bipedal with more fine controls, versus the wheeled robot in SuperTuxKart. Recognizing the wide breadth of this problem, we aimed to decompose the problem space into two primary modes: (1) searching for the puck and (2) driving to and scoring with the puck. Both modes rely on smooth driving through the ice rink and avoiding crashes.

2.1. Driving Agent Action Network

In an attempt to minimize manual intervention and tuning of a driving agent, we sought to use deep reinforcement learning methods. The main driving actions are acceleration, steering and braking. Steering ranges from -1 and 1, acceleration from 0 and 1 and braking is a discrete binary action. This is a mixed action space with both continuous and discrete variables. In general mixed action action spaces and continuous action spaces are harder problems in reinforcement learning, so we discretized the continuous variables. This approach has shown to be successful in improving reinforcement learning performance (Tang & Agrawal) in other applications. Previous work (**Homework 5**) show SuperTuxKart agents using normalized device coordinates to approximate steering angle perform well in racing tasks. We build on this idea by re-configuring the steering action space into a steering scaling factor that makes turns more or less sharp. This scaling factor is discretized into 6 buckets (scalars between 0 and 5). This discretization is applied to acceleration by cutting the values into 6 buckets between zero and one. This gain in simplicity far outweights the cost of fine grained adjustments since acceleration values vary across a small scale. Braking is already discrete, but some modification was needed still needed to make it suitable for

our network. We bucket the range from zero to one and then apply a threshold, which when above the threshold would set braking to True and when below, set it to False.
With the reduced action space, we searched for a reinforcement architecture which could handle simultaneous actions, and could be implemented efficiently and simply. Deep-Q-Networks (Mnih et al., 2013) are a fairly straight forward approach that could be adapted nicely to our problem space. Vanilla Deep-Q-Networks, however, are not configured for simultaneous actions, so an extension of Deep-Q-Networks, the Branching-Dueling-Q-Network (Tavakoli et al., 2017), was used. [2]. The Network[3] was kept lightweight, consisting

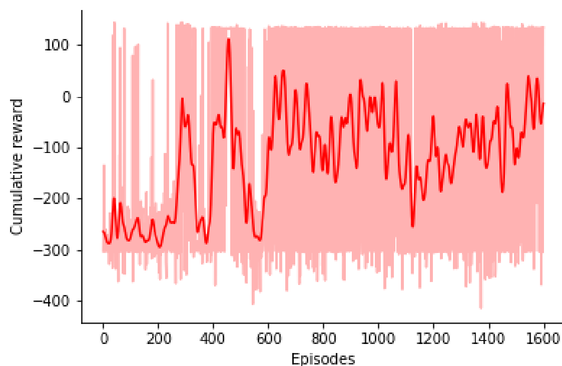

Figure 3. The Rewards of the Driving DQN, showing its performance approaching and keeping within distance of the puck just through tweaking its steering gain and acceleration gain. The initial exploration phase and its low reward can be seen at the beginning, followed by a gradual improvement, some oscillations in its reward and then some stability towards the end.

of only linear layers that took as input an aim point, distance, location vector of the player, quaternion vector of the player, and the magnitude of the player's velocity. The agent was then put into a modified tournament environment with 3 AI's. The agent was then tasked with following the puck as it moved throughout the field, getting a negative reward based on its distance to the puck (the further away from the puck, the higher the punishment) and the log of its current speed. We rewarded it for speed, in an attempt to learn to reverse out of a collision that left the player stuck in a wall, however that behaviour was not learned. We believe that the inclusion of the braking action lead to confusion so we eventually removed it from the from the learned action space and handled braking and collision recovery manually. For training, the puck represented an arbitrary location on the map, the main idea being that if the agent could keep track and stay on top of a moving object, it could move to static ones as well. After training, we found that this training process enabled the kart to arrive at any given target point without intervention, and interestingly, dribble the ball efficiently as that strategy enabled it to have control over the ball so that it wouldn't lose it.

2.2. Driving Controller Logic

We implemented a controller that simplified the task of winning a game of SuperTuxKart Ice Hockey into two modes: "search" and "ball". In search, the goal of the player is to scan the playing field for the puck. In ball mode, the player drives to the puck and, under certain conditions, attempts to dribble the puck into the goal. As each game has two agents, we created separate, time and state-dependent roles for the players: (1) striker (2) defender. These roles are closely related to the tasks. These separate roles are crucial to prevent our players from interfering with each other and both trying to dribble the puck. Details on the roles and modes follow in the next section.

Additionally, the controller "smoothed" the outputs from the networks by maintaining a memory of critical state conditions. Through this memory, the controller was able to override bad inputs from either the vision network or the action network by taking average actions over a few frames. While both networks are well trained, this smoothing is still necessary with such a large and complex world space. In all cases outlined below, the controller returns a final action to the game that is based on smooth states, not instantaneous frame states.

The final controller task is monitoring the kart for errors and rescuing the kart after a collision. When the kart crashes into a wall, or enters the goal area, the controller uses two heuristics. First it checks to see if it has direct line of sight over the center of the field using the visibility computation seen in section 1. If it does not, it will perform a reverse action in a direction determined by the quadrant on the field it is currently in. If the player is stuck against a wall, then this mode completes once it has direct line of sight of the center. The second heuristic is used when the the player is stuck in the goal it will only complete once it has successfully left the goal, as determined by the player's current depth.

2.2.1. Roles and Modes

Roles are assigned to players based on their mode and additional state conditions. In general, whichever player is closest to the ball with the ball in their field of view (in "ball" mode) becomes the striker, and the other player becomes the defender. If both players can see the ball, then the closer of the two (using estimated puck distance from the vision system) becomes the striker, and the other the defender. A defender's task is to continue searching through

[2] See Appendix for a rudimentary explanation of DQNs and related extensions

[3] Implementation of Branching Deep-Q-Network from: https://github.com/MoMe36/BranchingDQN

the space, but not to lock on to the ball unless it both sees it and is closer to it than the striker. The defender moves to coordinated world state locations to prevent opponents from scoring and to keep a look out for the ball in case they see it and have a lower distance than the current striker (in which case, the defender becomes the striker).

The striker's main objective is to drive toward the ball and, when the ball is close, dribble it to the goal. When the striker can see that ball and is in close proximity, the striker enters ball mode and slows down to dribble the ball. When the striker is in ball mode and can also see the goal, the striker biases it's aim to dribble the ball into the goal.

In search mode, both players move to specific world coordinates that are designed to form a trajectory that, when both players are combined, maximized the field of view of the team over a few frames. Thus, search involves some turning and communication between agents. When both agents are in search, the team is able to scan the field most efficiently.

Mode over time for games between Sara the Wizard and different AI difficulties is shown in Figure 4. Here, we can see long stretches of "ball" mode, which indicates periods where Sara is dribbling. It is noteworthy that Sara spends more time in search when playing against more difficult agents. This is likely because those agents maintain possession of and move the puck much faster through the hockey rink, and thus Sara must do more work to seek out a constantly changing puck location. Sara can lose the ball and go back into search when she is intercepted by another player or crashes the ball into a wall.

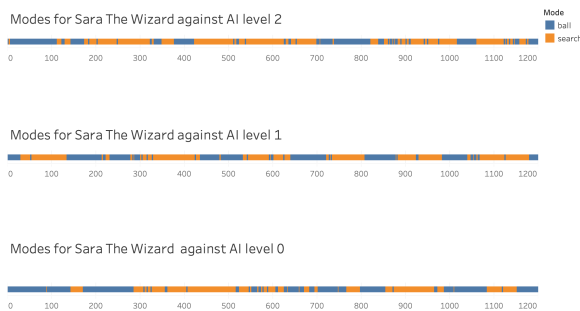

Figure 4. Modes over time for Sara the Wizard.

3. Conclusion

Our final agent consists of three parts: a multitask vision system, an automated driving agent for navigating toward an aim point and the controller that connects the two and enforces additional gameplay logic. Through the combination of these parts, our SuperTuxKart agent learned to translate it's field of view into a target, and from that target and additional state attributes, decide on an optimal action to score goals as part of a team. An established logic system was built to translate the game of ice hockey into a task of searching and controlling. Through the driving agent, the player could be fed different points around the field that the ball could be potentially close to. Once the vision system detected the ball, the controller would be able to move towards it with the driving agent. From there, using prior knowledge about the field the player would then be able to navigate the ball towards the goal. We present our benchmarked results in Table 1. In general, the performance when it came to searching for and keeping control of the ball met expectations in most cases. The more difficult controller task was finding the right angles to score goals, and in general many of the scored goals seem to come from a good orientation while retaining possession of the ball. A solution to this issue may have been a coordinated effort between the two player agents to score goals, which we leave as future scope.

Table 1. Number of goals scored across AI difficulty levels (total over 100 games)

AI Difficulty Level	Goals scored
0	28
1	32
2	31

4. Appendix

4.1. Other Avenues

While the final method was an ensemble of networks , involving Deep-Q-Learning and Computer Vision systems, the team recognizes that this could be done with an end-to-end reinforcement learning approach. In theory, SuperTuxKart Soccer, can be considered a game with a large continuous state space, and a large mixed simultaneous action space with a sparse reward. There have been a variety of approaches considered for solving tasks such as this, but are notorious for being very difficult. One source of inspiration for an end-to-end reinforcement learning approach came from Deepmind creating an agent to play Capture the Flag. (Jaderberg et al., 2019) Soccer and Capture the Flag are fundamentally similar in terms of game mechanics, and the platform used was 3d as well with similar actions. Our problem, in general was similar, but configuring the environment and translating the tasks would have been very challenging, especially without any released code to work with. In general, though reading through their approach gave the team the tools to do their own research and come

up with some approaches making use of some state of the art methods in reinforcement learning. In particular, we landed upon using an Asynchronous Actor Critic Approach (Babaeizadeh et al., 2016) as the base learner, and an Intrinsic Curiosity Module(Pathak et al., 2017). The explanation for how Actor Critic and Intrinsic Curiosity works is beyond this paper, but in summary an Actor Network predicts an action, and the Critic determines whether that action was good, based on the experiences it has already seen. Intrinsic Curiosity Modules, handle the problems of sparse rewards by giving the agent an internal reward based on the novelty of states it has already seen, by having a network attempt to predict the next state given its current state and action. It then optimizes by taking the loss of its predicted next state and the actual next state.

Configuring this method from scratch was beyond this team, but at the professor's suggestion in his Reinforcement learning lecture, making use of pre-written implementations [4] would be the smartest path towards implementing these systems. Tweaking the environment was straight forward, but discretizing the actions was less so. An attempt was made by essentially bucketing steering and acceleration into 20 and 10 values respectively, and then creating a mapping that took the outputted index from the Actor Critic model to a combination of those 2 actions. In general, this approach was incredibly hard to tune due to the breadth of background knowledge necessary, so was discontinued.

The team also attempted imitation learning with DAGGER but due to some limitations with the simulator, and the available time this approach was also scrapped. In general, the idea was to essentially track the players exact location and orientation, while keeping an the built-in AI off the map for one step, and then swapping it with the agent with the exact same location and orientation as the player, but this lead to some further technical difficulties.

4.2. Deep-Q-Networks and Extensions

DQNS are an extension to Q-learning, which essentially track expected reward given an action and a state via a table, known as a Q-Table. These Q-Tables don't scale very well, so DQNs are an extension that approximates this table using Deep Learning. DQNs are further augmented by introducing a replay buffer that allows the network to randomly sample state action pairs in non-sequential order to reduce correlation between states, as well as be reminded of past experiences as they continue to learn. Dueling DQNs are another extension that stabilizes this learning process. This is necessary, since the network is essentially comparing an outputted Q-Value to a target Q-Value that it estimates based on its experiences, however the target Q-Value changes rapidly as the network trains, which destabilizes training, so adding a second target network reduces that instability. The Branching Dueling-Q-Network, finally enables simultaneous action spaces to take place by creating multiple heads for different actions, given a shared state representation.

[4] Prewritten Implementation for A3C and ICM here: https://github.com/sadeqa/Super-Mario-Bros-RL

References

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., and Kautz, J. GA3C: gpu-based A3C for deep reinforcement learning. *CoRR*, abs/1611.06256, 2016. URL http://arxiv.org/abs/1611.06256.

de Vries, J. Coordinate systems. URL https://learnopengl.com/Getting-started/Coordinate-Systems.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castañeda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J. Z., Silver, D., Hassabis, D., Kavukcuoglu, K., and Graepel, T. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019. ISSN 0036-8075. doi: 10.1126/science.aau6249. URL https://science.sciencemag.org/content/364/6443/859.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. Robocup: The robot world cup initiative. In *Proceedings of the First International Conference on Autonomous Agents*, AGENTS '97, pp. 340–347, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897918770. doi: 10.1145/267658.267738. URL https://doi.org/10.1145/267658.267738.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning, 2013.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017. URL http://arxiv.org/abs/1705.05363.

Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent

neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Super-Tux-Kart-Development-Team. Making tracks: Appendix d: Soccer and battle modes. URL https://supertuxkart.net/Making_Tracks:_Appendix_D:_Soccer_and_Battle_Modes.

Tang, Y. and Agrawal, S. Discretizing continuous action space for on-policy optimization. URL https://ojs.aaai.org/index.php/AAAI/article/view/6059.

Tavakoli, A., Pardo, F., and Kormushev, P. Action branching architectures for deep reinforcement learning. 2017.

Van Rossum, G. *The Python Library Reference, release 3.8.2*. Python Software Foundation, 2020.

Zhou, X., Wang, D., and Krähenbühl, P. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019.