Augmenting Deep Reinforcement Learning on Toribash with Expert Matches

Yash Gandhi
University of Colorado Boulder
yash.gandhi@colorado.edu

Abstract

There are a variety of environments and simulators to test deep reinforcement learning algorithms. One of the less studied is a a game-based environment called Toribash. We believe Toribash to be a useful platform for learning about common research problems in reinforcement learning such as high dimensional tabular input and complex control structures with highly coupled variables. Also, previous works in Toribash have focused on winning matches or evolving specific moves. Here, we focus on generating matches using deep reinforcement learning that behave similarly to human played matches. We do this by introducing augmentations informed by a data set of expert matches. Finally, we measure the results of our models using hidden markov models.[1]

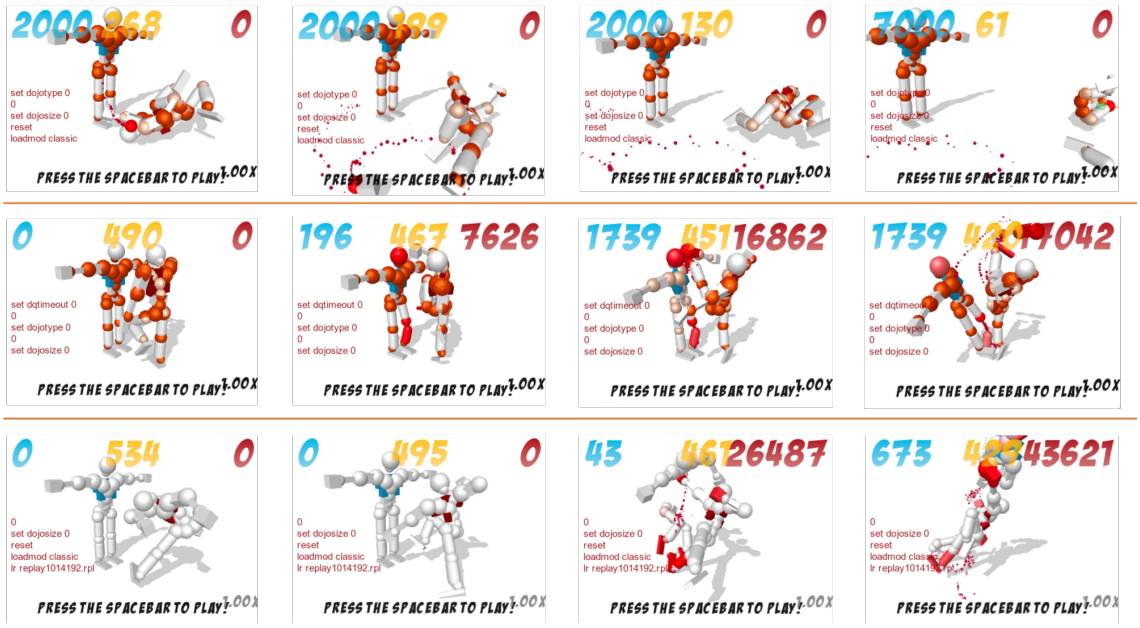

Figure 1: *We visualize certain frames from models and experts to compare their behaviors. (Top) A full multi-discrete model must learn all 22 joints simultaneously. This proves to be difficult for the deep reinforcement learning model and thus these frames demonstrate seemingly random actions and a losing strategy. (Middle) By limiting the action space to only 30 possible actions chosen from the set of expert trajectories, the agent learns to kick the opponent player. (Bottom) Human matches demonstrate much finer control and often have more build up to larger, more destructive actions.*

[1]The code is available at https://git.cs.colorado.edu/yaga6341/csci-4831-7000.

I. Introduction

Current applications of deep reinforcement learning (RL) vary from robotics, resource management, autonomous driving, and many others [1], [2], [3], [4]. Through the use of neural network, deep RL methods can process high dimensional inputs and produce skilled control. For example, Gu et. al. were able to train a robust robotic arm policy to open a door, Cheng et. al. were able to learn manipulation of occluded objects by controlling both the robotic arm and its gaze, and Zhu et. al provided fine control over multiple fingers for rotating valves, flipping boxes, and, again, opening doors [5], [6], [7]. Unfortunately, this can be a costly endeavour. Neural network approaches require many thousands of samples to learn and many more iterations to learn separable latent spaces. This drawback is magnified by the dynamics of an environment in deep RL and can make many robotic applications impractical. Thus, games can act as a simple, yet powerful proxy. Since game-based environments can be run indefinitely, researchers can tackle difficult research problems in simulation. The most common environment, the Arcade Learning Environment (ALE), offers a variety of platforms from the Atari 2600 to test genreal RL agents and models and demonstrated the abilities of the deep Q-Network (DQN) [8] [9], [10]. One such environment that has not been as intensely studied is Toribash. We believe Toribash to a another powerful testbed for deep RL and train many different augmentations to the original learning environment [11].

Toribash allows players to control a human-like figure consisting of 22 different control surfaces - joints. Each of the joints affects a different part of the body. Twenty of the joints can be in four possible states while two of the joints, hand grips, can be either on or off. Each turn, the player changes any subset of the joints and then moves the game forward a certain number of frames. The goal of the game is to fight a similarly controlled opponent and inflict the most damage [2]. Against a static opponent, this is a fairly simple task and can be learned without much modification to the original environment. Although, this results in nearly random movement that looks nothing like matches run by actual players. We use a number of techniques and methods to create action trajectories that are visually similar to matches played by humans.

II. Data

Deep RL relies heavily on the reward function because it dictates the direction of positive - and negative - growth. In certain situations, this can be implied by the goal of the agent or environment. If the goal is to reach a specific position, then inverse distance is an effective reward function. If the goal is only to win the game, the agent may learn to maximize the score. On the other hand, more complex goals can make engineering a reward function much more difficult. Even though the purpose of Toribash is to win the match, we want the agent to learn a constrained set of actions that are similar to the movements of human players. So, we used a set of expert trajectories to further improve our methods. Many of the methods we describe in section IV either use expert trajectories to reduce the effort of the learning algorithm or are inspired by observed trends among the many matches.

Toribash has a built in match saving feature and a large community of competitive players and match builders. The builders can create fights by moving the game a single frame at a time. Since the games development, many hundreds of thousands of replays have been shared among the community. For our project, we have chosen 47 matches that demonstrate a wide array of skill and precision. Each match averages around 1000 frames, or 1000 samples. Each frame represents a vector of values which we consider to be our state space. The state is a concatenation of player 1 and player 2's state

[2]We note that the game actually has hundreds of different game types and each with their own specific rule set. For this project we will only consider the basic rules defined at the start of the game.

values where each players state is comprised of different components shown in table 1.

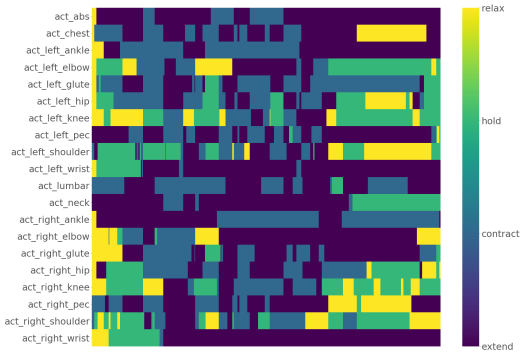

Figure 2: *Expert matches are highly dynamic and require constant fine tuning to all of the different joints throughout the duration of the game.*

From a heatmap of a specific expert match shown in figure 2, we can see exactly how dynamic the trajectories are. Futhermore, this demonstrates that a well played match consists of constantly fined-tuned actions and knowledge of how each joint will effect the overall motion of the Toribash agent. With regards to our learned agent, this implies that it must learn how every possible action can effect the outcome of the next step and of the match. To alleviate this intractable search space, we develop a set of tools that leverage these matches to simplify this problem in section IV.

III. Background

Reinforcement learning is commonly used for solving complex control task where the goal is to let an agent observe certain state values and choose an action guided by a reward signal. More concretely, RL is a solution to a Markov Decision Process (MDP).

$$\begin{aligned} R &: SxAxS \to \mathbb{R} \\ R(s_t, a_t, s_{t+1}) &= r_t \end{aligned} \tag{1}$$

An MDP is defined as a 5-tuple: (S, A, P, R, γ). The space of observations or states, S, represents the information given to our agent. The available control variables, or actions, and their domains are defined by A. An MDP exists within an environment, which is constrained by some transition function P defined by equation 2. This allows us to define the probability of moving to any other state given our current state and chosen action. The goal of the control task is represented as a reward signal function on the current state, action, and next state (equation 1). Finally, γ is a discount factor that weights the importance of future rewards. A larger γ will be more considerate of future rewards while a smaller γ will create a short-sighted RL agent.

$$P(s', a, s) = P(s_{t+1} = s' | s_t = s, a_t = a) \tag{2}$$

Solving an MDP with RL means finding some optimal control sequence that satisfies our goal. Formally, an agent's policy, π, must maximize our agent's reward. The expected return of a agents current policy is usually defined as a value function on the state shown in equation 3. The optimal policy is one that maximizes equation 3 over all possible states. In classical methods, the policy at any step s_t was a greedy algorithm on the plausible future states ranked using their value function. While the classical methods offer powerful solutions for some environments, other considerations make it difficult to use naively. First, the state space grows exponentially with the number of features. If any of the features are continuous, then the summation over possible states may be intractable. Futhermore, a greedy policy assumes memorization of all possible state and action policies which is infeasible, again, with a growing number of continuous state features and actions. This has inspired many different algorithms that replace the value and policy with neural networks [9], [12], [13], [14].

IV. Methods

Because of the complexity of this environment, we employ a number of different methods and techniques to simplify learning and generate

Components	Number of Components
Position X	21
Position Y	21
Position Z	21
Velocity X	21
Velocity Y	21
Velocity Z	21
Joints	22

Table 1: *A players state consist of seven different segments that define its position, velocity, and current joint configuration.*

more human-like actions.

$$V : S \to \mathbb{R},$$
$$V^{\pi}(s_t) = \mathbb{E}\left[\sum_i \sum_{a \in A} \gamma^i R(s_{t+i}, a_{t+i})\right] \text{ or}$$
$$V^{\pi}(s_t) = \max_a \left[r_t + \gamma \sum_{s'} P(s_t, a_t, s') V(s')\right] \tag{3}$$

I. PPO and TRPO

We use the proximal policy optimization (PPO) algorithm to train all of the deep RL models [14]. PPO is a method that simplifies the implementation of the trust region policy optimization (TRPO) algorithm [13], [14]. If the network parameters are rapidly changing, then it may take longer for the agent to learn how to properly control its environment, so both TRPO and PPO introduce a policy update that limits the change in the policy. TRPO introduces a hard constraint on the distance between the policies and their KL-divergence while providing monotonic updates [13]. On the other hand, PPO forgoes the theoretical monotonicity, and clips the ratio of new to old policies to both simplify implementation and also stabilize the TRPO criterion [14].

II. Reward Functions

Defining a reward function that promotes Toribash agents to learn human-like actions is a difficult design task. So, we tested different candidate functions against one another. Some of these functions leveraged the data set of expert matches, while others were based solely on observed heuristics. Here, we describe some of the candidate functions used during testing.

I Score Reward

The simplest reward function is one that maximizes the agent's in-game score. Although, the in-game scoring system can sometimes give unreasonably high scores, so to stabilize training we return the log of the score. Furthermore, we noticed that most experts would only change at most half of their joints during any one turn, so we append a L_1 penalty on the change in action. This reward type served as a baseline to measure against others and also served as the only sparse reward function. Most expert matches have a very sparse distribution of the score as seen in figure 3.

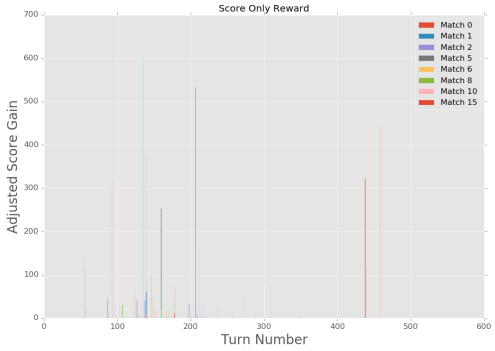

Figure 3: *In game score gains during different matches. Each spike represents a change in the score of the game.*

II Weighted Linear Reward

Because our goal is to produce policies that mimic human matches, we can leverage the saved replays to learn a reward function. This is a common method to learn a reward function given human demonstrations. This method, called inverse reinforcement learning, has an expansive literature on different methods to learn a function from previously demonstrated samples [15], [16], [17], [18]. For this project, we use a similar formulation as Hadfield-Menell et. al. and their Inverse Reward Design method [19]. In their paper, they define the reward as a linear combination of weights and a function on the state (top equation 4). We assume that $|phi$ is a function condition on some transformation matrix, and, here, we use the identity matrix for simplicity. To solve for the weight vector, w, we used the expert trajectories to solve for the normalized increase in score during a match rather than purely the score itself. We solved for w using ridge regression and by tunning the tradeoff parameter.

$$\begin{aligned} r(s_t) &= w^T \phi(s_t) \\ \phi(s_t; \mathbb{I}) &= \mathbb{I} s_t \end{aligned} \tag{4}$$

III Distance Reward

While exploring the expert trajectories, we noticed that the distance between the players and the score were often inversely correlated. Because of this observation, we introduced a reward function that negatively penalized the score reward with the distance between each players center of motion.

IV Curriculum Reward

Since the goal of Toribash - winning - is sparse, then we employ curriculum learning. Curriculum learning allows the agent to gradually learn harder tasks by changing the reward function during training. Originally introduced for classification techniques, this process can be thought of as learning gradually less smooth objectives [20]. We have four different segments based on previous reward functions that defines our curriculum. First, the agent should learn to win the game regardless of the performance. To make this a dense function, we penalize each step where the second player's score is higher than the agent's score. Next, the agent should learn to accrue more points. Instead of just adding the score to agent's reward, subtract the log of the difference between the score at time t and time $t+1$. This way, the agent has to learn how to regain its lost points and stops it from remaining stagnant. The third segment subtracts the distance to second player. This integrates the correlation we observed earlier and forces the agent to stay close to the second player. Finally, we add in the weighted linear reward to encourage movements similar to the expert trajectories.

III. Action Spaces

I Single Agent

With 22 different joints and four possible states for each joint, the space of total configurations for a single turn is exactly 4^{213}. Because of this, we introduced a few simple augments to the action space to make it more feasible to learn. First, instead of choosing from the entire space, just pick N random actions. While it drastically decreases the expressiveness, the RL model can more appropriately explore its own action space. Although, now, performance across models depends heavily on the random actions chosen. So, since we have matches played by experts, we can use that information to choose a more appropriate set of N actions. One method is to just pick the N most frequent actions from all the matches. Although, this can skew the actions towards beginning moves which are typically not useful for dealing damage to other opponent. Rather, most beginning

[3] We refer to full action space as Multi-Discrete.

moves represent a build-up phase. To remedy this, we can use a gaussian kernel that weights moves closer to the center of the match more than those at the beginning and end. Some matches use the end of a match to prepare a final pose rather than using those moves to deal damage. Although, these discrete spaces severely limit the space of possible actions.

Here, we describe a method that drastically decreases the number of actions the model has to learn, but still allows for access to the entire action space. For the continuous case, the model learns two actions x, y where both $x, y \in \mathbb{R}_+$. These two values represent a point on the cartesian plane. We can divide the plane into bins based on the total number of discrete actions and whatever bin the point falls into decides which action to take. Although, since our discrete action space consist of 4^{21} possible actions, iterating over each one can become difficult. So, we employ the following procedure to greatly speed up this calculation. First, we can calculate the bin number because the total number of actions is known. Then, we calculate the base 4 value of that number. Since each joint can be in only one of four possible configurations, we can take the digits of the base 4 value to be the next configurations for the joints.

II Multi-Limb Model

Instead of trying to solve the entire problem with a single model, we also tried to compartmentalize the action space by segmenting the Toribash figure. The agent is first broken up into six different sections: left and right leg, left and right arm, and upper and lower body. Each one of those networks is only responsible for a small number of actions[4]. A higher level model receives the normal state input and sends a K dimensional signal to each lower level. The limbs learn to interpret this signal and send back commands for their respective components. The higher level aggregates the actions and sends it to the environment.

$$r(s_0) = c_0 P(s_0|\sigma) - c_1 \tag{5}$$

There are two possible routes to training this set of models. One is to train all of the models at once. This is computationally difficult because it essentially amounts to training seven different networks - six lower levels and one higher level. Also, it requires a more complex communication structure between the models. The other way is to to train the limbs first and then train the higher level. The limb models can be trained by sending random permutations of signals to the lower level and then measuring how well they interpreted the signal. During training, each limb model receives some random signal as a vector $\sigma \in \mathbb{R}^4_+$. The first two values of σ are the center of a 2D distribution and the second two values are the diagonal of the covariance matrix. The lower limb models output a x and y value, similar to the continuous actions defined for the single agent model, and that defines a configuration of each limbs respective joints. Although, since each model is only concerned with a few joints, there is less granularity for each bin. The reward for the lower limb models is the probability of the chosen point given the distribution defined by the signal (equation 5, figure 4).

The controller model, now, acts as an intermediary between the limbs and the state input receiving rewards based on performance in the game.

III Embedded Models

In the game Toribash, based on domain expertise and observing the expert trajectories, there is a clear bias for certain joints for larger moves and certain joints for finer control. We incorporated this thinking into another model type we call the embedded model. Here, we train subsets of the joints in a gradually embedded style. First, we train joints that are used for larger movements: turning the entire body, jumping, rotating entire limbs. Then, we train

[4] All of the models either controlled 3 or 4 joints which means 64 or 256 actions for any one model

a separate set of joints that are used for a finer set of attack moves: punching, kicking. Finally, we noticed that a smaller set of joints were used for precision control to increase the advantage during the matches. These models are the major, minor, and detail models respectively. The major model is trained completely independently using any of single agent action types from direct state input and any reward function. Next, we train the minor model, again with any of the single agent actions and any of the reward functions, but also include the trained major model. Finally, the detail model is trained in the same way, but with both the major and minor models.

IV. Stochastic Similarity

Since the score of the match does not evaluate the components of the trajectory generated by the model and how similar it is to human players, we use a different measure for success of the model. We implement a stochastic similarity measure as described by Nechyba and Xu [21]. This paper uses a pair of Hidden Markov Models (HMM) to determine the similarity of two observations. A HMM allows us to reason about the probability of sequences given influence by some set of hidden states. A HMM, H, is defined as a 4-tuple given by equation 6. The Q represents the states of the model, Π is the distribution of the initial states where each $\pi_i \in \Pi$ is the probability of starting in state $q_i \in Q$, and O is the sequence of observations seen by the observer. Here, $\lambda = (A, B)$ where A is the transition matrix such that the probability of going from state q_i to state q_j is a_{ij} and B is the likelihood of emitting observation o_t at state q_i or $b_i(o_t)$.

$$H = (\lambda, Q, \Pi, O) \tag{6}$$

It is possible to learn the transition probabilities, A, and the emission probabilities, B, by using the forward-backward (Baum-Welch) algorithm and a set of observations [22]. Before measuring similarity across different experimental runs, Nechyba and Xu process their multi-dimensional trajectories using a codebook [21]. We use a similar strategy, but before discretizing the data, we reduce the dimensionality of our data using principle component analysis (PCA). Then, we use a K-Means algorithm to generate our codebook. We perform this process separately for the data from the experts and the data generated from executing a learned policy. We use these discretized data sets to learn a HMM for each of the models: human players and learned policy. From here, we sample matches from both sets and measure the similarity using equation 7 where α represents either the generated match or expert match and β represents the HMM learned from either the generated matches or expert matches.

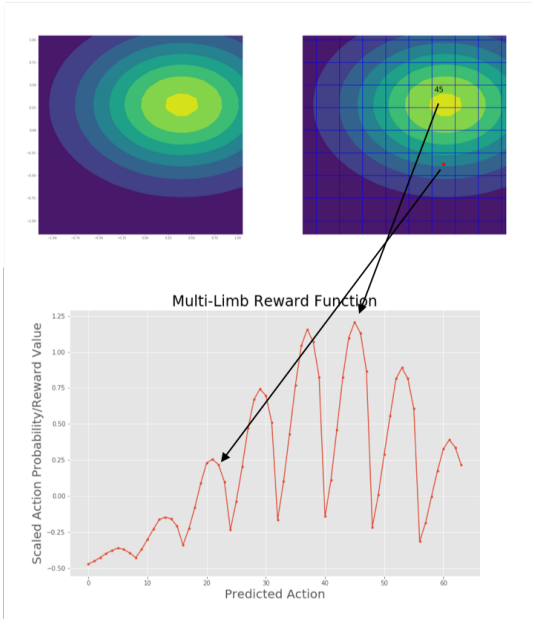

Figure 4: *The upper left image represents a distribution defined by some signal. The upper right image shows that same distribution, but with bins placed on top of the distribution. The red dot represent the action taken by the limb model. The lower graph shows what the reward values is for the given point and what it would have been if the point made it closer to the signaled bin.*

$$P_{\alpha,\beta} = 10^{\frac{Pr(\alpha|\beta)}{|\alpha|}}$$
$$\sigma(O_1, O_2) = \sqrt{\frac{P_{12}P_{21}}{P_{11}P_{22}}} \tag{7}$$

The final score is an average over the number of samples. A detailed outline of this algorithm can be found in the appendix A.

V. Results

Here we compare the many different model types, action types, and rewards against one another. Each of the single agent methods and the multi-limb model were trained for 100,000 iterations. For the multi-limb environment, each individual limb was trained until the last limb showed convergence on its reward value at about 75,000 iterations. To be able to compare models against one another, reward parameters were fine-tuned for the first model in which they appeared and then propogated to other models. For example, constant multipliers tuned in the score only reward for random actions were the ones used in the random actions model with a curriculum reward. We report similarity scores for all of these models in table 2.

We can see from the table that the curriculum reward well at creating policies with high stochastic similarities. One particular example of the curriculum reward that has a higher similarity to the human matches is the weighted frequent actions model. When observing this model, it had learned to kick the opponent player well enough to detach its arm. This example can be seen in the middle row of figure 1. From the table, we can see that certain reward functions tended to perform poorly. Most models using the weighted linear reward function alone failed to even cross a similarity of 0.001. Often, though, we would see many of the agents learn to deal a small amount of damage and then either retreat away or stay stationary on the ground.

VI. Conclusion

The values in table 2 demonstrate that many of the methods described fall short of generating human-like actions. Curriculum rewards, though, were the most useful for accomplishing this goal. These models were able to learn through a more stable process rather, which allowed for the agent to learn a more stable set of actions to take. The multi-limb environment obtained the worst results and we believe it to be the fact that the upper level controller had to learn a general enough signals for all of the models to interpret. In future work, we would have a upper controller that sends an individualized signal to each of the lower level limbs. Stable models like the embedded model and the continuous model all demonstrated similar behaviors across the reward functions. This leads us to believe that the reward engineering was a much more vital part of learning.

While still a relatively new environment for testing deep reinforcement learning methods on, certain aspects of Toribash have been studied before. Kanervisto and Hautamäki created the original environment and trained some simple agents on a variety of different task [11]. They built a communication strategy to allow for state-of-the-art algorithms to interface with the original game software and also tested some initial competitive training environment like self-play and playing against human players [11]. Other works also include some genetic algorithm work to try and generate offensive opening moves [23]. These papers further prove that Toribash is a good environment for training learning agents. While our focus has been on generating policies to play in the standard game, Toribash offers a massive set of further modifications developed by community members.

Learning general agents is a costly endeavour and games offer a simple way to approximate common research problems while being able to be run indefinitely. We have shown that Toribash offers a number of desirable properties. Toribash is an interesting single agent platform with high dimensional input and con-

Action Type	Score	Weighted Linear	Distance	Curriculum
Multi-Discrete	0.004	0.030	0	**0.058**
Random	0.021	0	0.012	**0.067**
Frequent	0.057	0	0	**0.089**
Weighted Frequent	0	0	0	**0.273**
Continuous	0.094	0.120	**0.203**	0.001
Multi-Limb	0.001	0.001	**0.003**	0
Embedded Model	0.022	0	0.072	**0.137**

Table 2: *Stochastic similarity values for all of the single agent policy methods and the multi-limb method using all of the different reward types. Scores closer to 1 represent more similar models. Each model was trained for approximately 100,000 iterations and each learned policy was run for 50 episodes for PCA, K-Means, and HMM training. The models were tested on 25 random samples pairs from the expert matches and generated runs. Any score lower than 0.001 was floored to 0 for the purposes of this table.*

trol. It can be sectioned off into components to test multi-agent learners. By choosing specific subsets of actions, a hierarchical model can be trained where each level can either be independent or dependent. The extensive list of mods available make it simple to train a Toribash model in a new scenarios with meta-learning algorithms. And, while the basic goal of the game is to win, the purpose of generating human-like can be useful for training more complex goal oriented methods.

References

[1] Yu Fan Chen, Michael Everett, Miao Liu, and Jonathan P How. "Socially aware motion planning with deep reinforcement learning". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 1343–1350.

[2] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 3389–3396.

[3] Ning Liu et al. "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 372–382.

[4] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. "Benchmarking deep reinforcement learning for continuous control". In: *International Conference on Machine Learning*. 2016, pp. 1329–1338.

[5] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 3389–3396.

[6] Ricson Cheng, Arpit Agarwal, and Katerina Fragkiadaki. "Reinforcement Learning of Active Vision for Manipulating Objects under Occlusions". In: *arXiv preprint arXiv:1811.08067* (2018).

[7] Henry Zhu, Abhishek Gupta, Aravind Rajeswaran, Sergey Levine, and Vikash Kumar. "Dexterous manipulation with deep reinforcement learning: Efficient, general, and low-cost". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 3651–3657.

[8] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. "The ar-

cade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[9] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[10] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[11] Anssi Kanervisto and Ville Hautamäki. "ToriLLE: Learning Environment for Hand-to-Hand Combat". In: *arXiv preprint arXiv:1807.10110* (2018).

[12] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[13] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust region policy optimization". In: *International conference on machine learning*. 2015, pp. 1889–1897.

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[15] Andrew Y Ng, Stuart J Russell, et al. "Algorithms for inverse reinforcement learning." In: *Icml*. Vol. 1. 2000, p. 2.

[16] Pieter Abbeel and Andrew Y Ng. "Apprenticeship learning via inverse reinforcement learning". In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 2004, p. 1.

[17] Faraz Torabi, Garrett Warnell, and Peter Stone. *Behavioral Cloning from Observation*. 2018. eprint: arXiv:1805.01954.

[18] Jonathan Ho and Stefano Ermon. "Generative adversarial imitation learning". In: *Advances in neural information processing systems*. 2016, pp. 4565–4573.

[19] Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. "Inverse reward design". In: *Advances in neural information processing systems*. 2017, pp. 6765–6774.

[20] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. "Curriculum learning". In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 41–48.

[21] M. C. Nechyba and Yangsheng Xu. "Stochastic similarity for validating human control strategy models". In: *IEEE Transactions on Robotics and Automation* 14.3 (1998), pp. 437–451. DOI: 10.1109/70.678453.

[22] Frederick Jelinek, Lalit Bahl, and Robert Mercer. "Design of a linguistic statistical decoder for the recognition of continuous speech". In: *IEEE Transactions on Information Theory* 21.3 (1975), pp. 250–256.

[23] Jonathan Byrne, Michael Neill, and Anthony Brabazon. "Optimising offensive moves in toribash using a genetic algorithm". In: (Jan. 2010).

A. Stochastic Similarity Algorithm

Here we provide the stochastic similarity algorithm for measuring model performance.

Algorithm 1: Stochastic Similarity

Result: Similarity between trained model and expert matches

$\xi_E \leftarrow$ EXPERT TRAJECTORIES
$L \leftarrow$ NUMBER OF SAMPLES TO AVERAGE
$C \leftarrow$ K-MEAN CLUSTERS
$d_h \leftarrow$ NUMBER OF HIDDEN STATES
$d_p \leftarrow$ REDUCTION DIMENSION
$h \leftarrow$ MODEL
$\xi_G \leftarrow [\,]$
for $k \in K$ EPISODES **do**
  $\tau \leftarrow [\,]$
  for $t \in T$ TIMESTEPS **do**
    $\tau_t \leftarrow h(s_t)$
  end
  $\xi_G^k \leftarrow \tau$
end
/* $d < |S|$ */
/* S is the state space */
$\mathbf{P}_e = \text{PCA}(\xi_E, d_h)$
$\mathbf{P}_g = \text{PCA}(\xi_G, d_h)$
$\mathbf{K}_e = \text{KMEANS}(\mathbf{P}_e, C)$
$\mathbf{K}_g = \text{KMEANS}(\mathbf{P}_g, C)$
/* $\mathbf{K}_{e/g} \in \mathbf{N}^{Kx1}$ */
$\lambda_e \doteq \text{BAUM-WELCH}(\mathbf{K}_e, d_p)$
$\lambda_g \doteq \text{BAUM-WELCH}(\mathbf{K}_g, d_p)$
$s = 0$
for $l \in \{1, 2, ..., L\}$ SAMPLES **do**
  $O_e = \mathbf{K}_e^l$
  $O_g = \mathbf{K}_g^l$
  $P_{11} = 10^{log\left(\frac{P(O_e|\lambda_e)}{\text{LEN}(O_e)}\right)}$
  $P_{12} = 10^{log\left(\frac{P(O_e|\lambda_g)}{\text{LEN}(O_e)}\right)}$
  $P_{21} = 10^{log\left(\frac{P(O_g|\lambda_e)}{\text{LEN}(O_g)}\right)}$
  $P_{22} = 10^{log\left(\frac{P(O_g|\lambda_g)}{\text{LEN}(O_g)}\right)}$
  $s \leftarrow s + \sqrt{\frac{P_{12}P_{21}}{P_{11}P_{22}}}$
end
return $\frac{s}{L}$