RL-Course 2024/25: Final Project Report

Oğuz Ata Çal, Kıvanç Tezören, Karahan Sarıtaş
Muhteshember
University of Tübingen

February 25, 2025

1 Introduction

In this project, we aim to develop a reinforcement learning agent capable of achieving a competitive performance in a 2-player hockey game environment (which inherits from `gym.Env`), simulated with Box2D physics engine. Each player controls a hockey stick and tries to score goals with a puck. Our observation space consists of 18 dimensions: position (x and y), velocity (x and y), angle and angular velocity of both players, position and velocity (x and y) of the puck, and remaining times for players to hold the puck (if `keep_mode` is set to `True`, otherwise players cannot hold the puck).
Actions in this environment are similarly represented by a list of four floats: change of direction in x and y coordinates, clockwise or counter-clockwise rotation, and shooting a puck. Agents can also pick and carry the puck before shooting.
For this problem, we implemented the following reinforcement learning algorithms:

- **Soft Actor-Critic (SAC)**, implemented by Karahan Sarıtaş,
- **Twin Delayed Deep Deterministic Policy Gradient (TD3)**, implemented by Oğuz Ata Çal,
- and **Deep Q-Networks (DQN)**, implemented by Kıvanç Tezören.

[TODO: introyu yarım sayfaya uzatmak lazım]

2 Method

2.1 SAC: Soft Actor-Critic

SAC is an off-policy algorithm that learns from a replay buffer, which contains experiences collected by the agent over time from different versions of the policy. To solve the exploration-exploitation dilemma, the policy is trained to maximize a trade-off between expected return and entropy. By incorporating entropy maximization into its objective function, SAC encourages exploration through stochastic behavior while also learning to act optimally. The entropy term helps prevent early convergence to suboptimal deterministic policies and allows the agent to discover multiple successful strategies. First, let's formulate the entropy of the policy $\pi(\cdot \mid s_t)$:

$$\mathcal{H}(\pi(\cdot \mid s_t)) = \mathbb{E}_{a \sim \pi(\cdot \mid s_t)}[-\log(P(\pi(a \mid s_t))] \tag{1}$$

In entropy-regularized reinforcement learning, the agent's objective is augmented with an entropy term. At each timestep, in addition to the environmental reward, the agent receives a bonus proportional to its policy's entropy, encouraging it to maintain stochastic behavior while learning optimal actions. Then the optimal policy is formulated as follows:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t,a_t,s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s_t))\right)\right] \quad (2)$$

In the objective function, α serves as a temperature parameter that controls the balance between reward maximization and entropy maximization. A higher α value places more emphasis on exploration through entropy maximization, while a lower value prioritizes reward maximization. This parameter can either be set as a fixed hyperparameter or automatically adjusted during training to achieve a desired target entropy level.

There are one actor π_θ, two critic Q_{ϕ_1} and Q_{ϕ_2} and respective target networks $Q_{\phi_{\text{targ},i}}$. The final form of the entropy-regularized Q-loss becomes the following:

$$Q^\pi(s_t,a_t) = \mathbb{E}_{s_{t+1}\sim P, a_{t+1}\sim\pi}\left[R(s_t,a_t,s_{t+1}) + \gamma\left(Q^\pi(s_{t+1},a_{t+1}) - \alpha\log\pi(a_{t+1}|s_{t+1})\right)\right] \quad (3)$$

Current state s_t, action a_t, next state s_{t+1} and the corresponding reward function $R(s_t,a_t,s_{t+1})$ are sampled from the replay buffer. Next action a_{t+1} is sampled using the actor policy π_θ given s_{t+1}. For the implementation, we can approximate this equation as follows:

$$Q^\pi(s_t,a_t) \approx r_t + \gamma\left(Q^\pi(s_{t+1},a_{t+1}) - \alpha\log\pi(a_{t+1}|s_{t+1})\right), \qquad a_{t+1}\sim\pi(\cdot|s_{t+1}) \quad (4)$$

To calculate the policy loss, SAC uses $\min_{i=1,2} Q_{\phi_i}$ which is the minimum of the two critic networks. In the original SAC implementation [6], a fixed entropy regularization coefficient (temperature) α is used. In the following work [7], a constrained optimization is proposed to tune the temperature during training.

Extension (1): Automating Temperature Tuning

The optimization problem for finding the optimal temperature variable is given as follows [7]:

$$\alpha_t^* = \arg\min_{\alpha_t} \mathbb{E}_{\mathbf{a}_t\sim\pi_t^*}\left[-\alpha_t\log\pi_t^*(\mathbf{a}_t|\mathbf{s}_t;\alpha_t) - \alpha_t\mathcal{H}_0\right] \quad (5)$$

Here $\mathcal{H}_0$ represents the predefined minimum policy entropy threshold. Authors of the same paper [7] choose $-\text{dim}(\mathcal{A})$ as the target entropy, where $\mathcal{A}$ represents the action space. During the implementation, we have to make sure that the temperature variable α is positive. One way to do it is to optimize $\log(\alpha)$ instead of α.

$$\log\alpha_t^* = \arg\min_{\log\alpha_t} \mathbb{E}_{\mathbf{a}_t\sim\pi_t^*}\left[-e^{\log\alpha_t}(\log\pi_t^*(\mathbf{a}_t|\mathbf{s}_t) + \mathcal{H})\right] \quad (6)$$

To ensure numerical stability, the authors chose to optimize a modified version of the objective function above where they replaced the exponent with directly the $\log\alpha_t$ - which doesn't affect the outcome of the objective function[1]. We followed the same approach in our implementation.

Extension (2): Prioritized Experience Replay (PER)

Normally, the transitions are uniformly sampled from a replay memory. Schaul et al. [13] introduces a new sampling strategy called Prioritized Experience Replay (PER) where we sample the important transitions more frequently. We prioritize the transitions from which our agent can learn significantly, by estimating the importance of a transition with TD error. Transitions with larger errors provide more valuable learning opportunities for the agent, as they represent cases where its predictions deviate significantly from actual outcomes.

[1]https://github.com/rail-berkeley/softlearning/issues/37

In alignment with the proportional prioritization proposed in the original paper [13], we adopt the "sum tree" data structure, which is a specific type of segment tree[2]. Priority update procedure is given as follows where $\mathcal{S}$ represents the sum segment tree and $\mathcal{M}$ represents min segment tree. A small constant ϵ is used to prevent the edge-case of transitions not being revisited once their error is zero.
For the sampling, we perform "stratified sampling" where we divide the total priority into equal segments and then sample uniformly among the transitions within each segment. Since this prioritization will lead to less diversity in the sampled transitions, Schaul et al. [13] proposes calculating the weights of the samples using importance sampling $w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta_2}$ where N is the buffer size and β_2 is used to determine the level of the correction and gradually increased during the training. $P(i)$ for the sample i is calculated as $P(i) = \frac{p_i^{\beta_1}}{\sum_k p_k^{\beta_1}}$ where p_i is the priority of the sample i, calculated according to the procedure given below, and β_1 represents the degree of the priority sampling (setting $\beta_1 = 0$ leads to uniform sampling).

Extension (3): Pink Action Noise

Actions are sampled applying $\tanh$ squashing and rescaling to $x_t = \mu(s_t) + \sigma(s_t) \odot \epsilon_t$ where ϵ_t represents the action noise for better exploration. Most commonly, a Gaussian distribution (white noise) is called to generate actions during training. Eberhard et al. [4] introduces a new action noise called colored noise where the signals ϵ_t drawn from it satisfy the following property:

$$|\hat{\varepsilon}(f)|^2 \propto f^{-\beta} \tag{7}$$

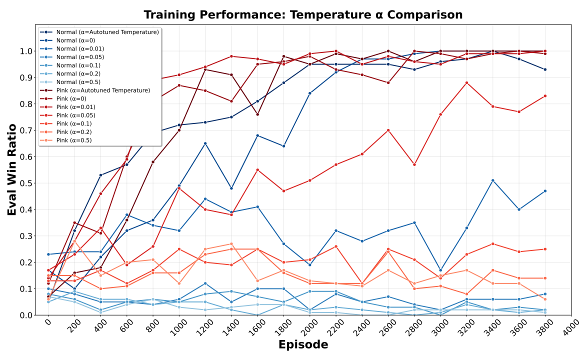
where $\hat{\varepsilon}(f) = \mathcal{F}[\varepsilon_t](f)$ corresponds to the Fourier transform of ϵ_t (f is the frequency) and $|\hat{\varepsilon}(f)|^2$ is called the power spectral density (PSD). β represents the color parameter. [4] empirically shows that the pink noise ($\beta = 1$) performs on par with the best choice of noise type in 80% of cases. We incorporated colored noise, including the specific case of pink noise, in our implementation[3].

2.1.1 Experiments

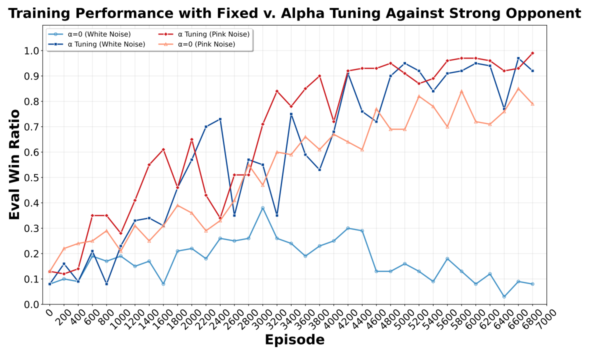
We conduct experiments with different hyperparameters for the SAC algorithm. Each hyperparameter can influence others, but due to time constraints, we kept most (or all) of the remaining hyperparameters fixed while varying one at a time. We begin by examining the impact of α tuning compared to using fixed α values. We train and evaluate the model against the weak opponent where evaluation takes place every 200 episodes and lasts 100 episodes.
First, we evaluate different temperature values (α, the entropy coefficient) using a vanilla SAC model with normal action noise. Figure 1a demonstrates that higher temperature values, which encourage excessive exploration, fail to converge. Meanwhile, automatic tuning and no temperature ($\alpha = 0$) shows similar training characteristics in both pink noise and normal action noise. To better understand the impact of not using temperature (in other words, the absence of entropy regularization), we further tested it against a **strong** opponent, this time with both pink and normal action noises. As shown in Figure 1b, automatic tuning consistently accelerates convergence in both pink and white action noise settings. Notably, white noise without temperature fails to converge altogether. Based on these findings, we decided to use **automatic tuning for the entropy coefficient** (α) in our final model.

[2]We use the Open AI implementation: https://github.com/openai/baselines/blob/master/baselines/common/segment_tree.py

[3]Original implementation of the pink noise is used: https://github.com/martius-lab/pink-noise-rl/tree/main/pink along with the Action Noise interface provided here: https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/noise.py

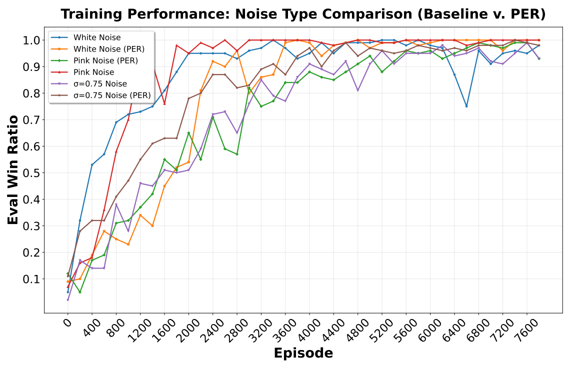


(a) Evaluation win rates against the weak opponent with different temperature values α

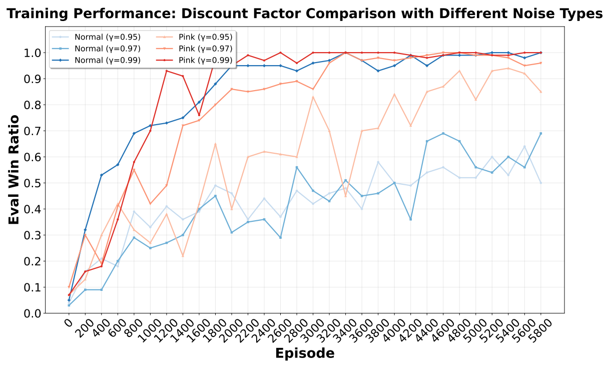


(b) Evaluation win rates against the strong opponent with different temperature values α

Having decided on which temperature value α to choose, we proceed with testing different action noises using both vanilla SAC and SAC with PER. Figure 2a shows the performance of white noise, pink noise, and colored noise with $\beta = 0.75$ (as it was shown to achieve a competitive performance in [4]). We used both vanilla SAC and SAC with PER. Our experiment against weak opponent yields that although all the configurations lead to convergence, pink noise with vanilla SAC achieves the fastest convergence among all. As outlined in [15], SAC+PER introduces additional overhead on top of vanilla SAC due to constant priority updates and sampling computations, which scale linearly with the mini-batch size. Since SAC+PER takes longer to train and does not consistently outperform the baseline—Wang and Ross [15] also reports similar or worse performance in some environments—we choose to use **vanilla SAC with pink action noise**. Next, we tested different discount rates, which scale down the effect of past experiences. As shown in Figure 2b, we evaluated $\gamma \in \{0.95, 0.97, 0.99\}$ with both normal and pink action noise. The results indicate that the default value of $\gamma = 0.99$ achieves the fastest convergence, so we chose **0.99 as our discount rate**.

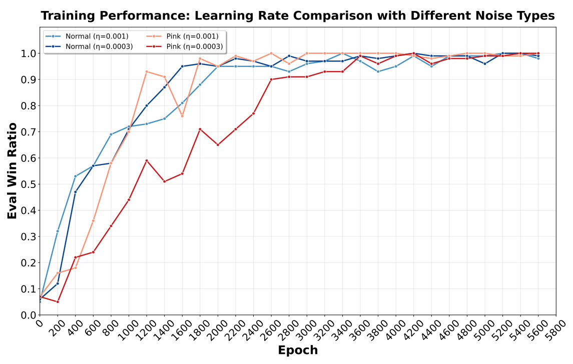


(a) Evaluation win rates against the weak opponent with different noise configurations



(b) Evaluation win rates against the weak opponent using different discounts (pink v. normal action noise)

Next, we compare two different learning rates, $\eta \in \{3 \times 10^{-4}, 1 \times 10^{-3}\}$, using both pink and white action noise to assess their general impact on training. As shown in Figure 3a, reducing the learning rate to $\eta = 3 \times 10^{-4}$ (default) negatively affects training performance with pink noise against the weak opponent. Therefore, we chose $\eta = 10^{-3}$ for our final model.
Naturally, this hyperparameter search has certain limitations. First, the choice of one hyperparameter can influence the effect of another, meaning that an optimal configuration would ideally require a grid search over all possible hyperparameter combinations. However, this approach is computationally expensive,



(a) Evaluation win rates against the weak opponent with different learning rates

and due to time and resource constraints, we could not expand the depth of our grid search.
Second, our experiments were conducted using a single seed. Given the stochastic nature of deep learning, running multiple trials would provide more robust conclusions. However, Haarnoja et al. [6] suggests that the entropy coefficient α is the most critical hyperparameter to tune, while SAC is generally less sensitive to other hyperparameter choices. Since we have already addressed this by incorporating automatic α tuning, our focus now shifts to how we train our model, which we detail in the following section.

2.1.2 Training the Final Model

Mirrored States and Actions One observation we can do about our hockey environment is that it is symmetric with respect to a horizontal reflection (across the x-axis). This means that if we flip all the y-components (and the associated angles and y-velocities) of the state, we obtain an equivalent "mirrored" state.
By using symmetry, our approach enables an agent to learn from both the original experiences and their mirrored counterparts, thereby fully exploiting the game's inherent symmetric structure. Although we omitted symmetric states during the hyperparameter search, we included both the original and mirrored states in the replay buffer during the roll-out phase in the final model training.

Reward Augmentation Currently, the initial reward uses the outcome of the game (win/loss) and then adds a bonus based on how close the agent is to the puck. However, we can incorporate additional aspects of the game such as the direction of the puck and whether or not agent touches the puck. To do so we used a simple, augmented version of the reward, using the proxy rewards already provided by the environment:

$$\begin{aligned} r_{\text{aug}} = &\, 0.5 \cdot \texttt{reward\_closeness\_to\_puck} \\ &+ 2.0 \cdot \texttt{reward\_touch\_puck} \\ &+ 1.0 \cdot \texttt{reward\_puck\_direction} \\ &+ r_{\text{w/l}}. \end{aligned}$$

where $r_{\text{w/l}}$ is +10 if our agent wins, -10 if it loses and 0 if the game ends with a tie.

Arena Naturally, transitivity does not apply in hockey—if model A defeats model B, and model B defeats model C, it does not necessarily mean that model A is superior to model C. To evaluate our models and determine the best one for the competition, we implemented a simple arena framework where each model competes against every other model. Their ELO ratings are then computed using the TrueSkill package.

Self-play: Prioritized Opponent Buffer The Prioritized Opponent Buffer[4] is used in self-play to store previously encountered opponents and select which one to face next based on Discounted UCB [9]. Each time a new opponent appears, it is added to a list, and we keep a rolling history of size τ with the outcomes of the most recent matches. If the result of the t-th match is r_t (for example, $r_t = 1$ for a win or $r_t = 0$ for a loss), we apply a discount factor γ to favor more recent games. Specifically, the "discounted count" for opponent i is $N_i = \sum_{t=1}^{\tau} \mathbf{1}\{\text{opponent}_t = i\}\, \gamma^{(\tau-t)}$, and the discounted average outcome is $X_i = \frac{\sum_{t=1}^{\tau} r_t\, \mathbf{1}\{\text{opponent}_t=i\}\, \gamma^{(\tau-t)}}{N_i}$. The bonus is calculated as follows $c_i = 2B\sqrt{\xi \frac{\ln(\sum_j N_j)}{N_i}}$, and picks the opponent i that maximizes $\text{score}_i = X_i + c_i$. This means opponents with worse performance (low X_i) or limited exposure (low N_i) receive more training attention, while still giving a chance to practice against others. After each match, the buffer updates its history with the chosen opponent and the outcome, allowing the next selection to adapt based on new results. We initialized POB with weak and strong opponents, along with different SAC and TD3 models trained with different settings (diverse reward augmentations, with our without symmetry, trained against weak/strong or in defense/shooting mode etc.). Our aim was to increase the variety of playing techniques our final model was exposed to during training. Lastly, we started our self-play training with a model that was trained against the strong opponent for 10K episodes. For self-play with POB, we trained it for 200K episodes and updated the opponent for each 50 episodes if agent is able to achieve 0.95 win rate against everyone in the buffer - otherwise we just continue with the current agent.

3 Twin Delayed Deep Deterministic Policy Gradient (TD3) [5]

3.1 Method

Twin Delayed Deep Deterministic Policy Gradient (TD3) is an off-policy reinforcement learning algorithm that addresses some of the challenges in value-based methods while maintaining stable learning in continuous action spaces. Building upon the Deep Deterministic Policy Gradient (DDPG) framework, TD3 introduces three significant innovations to mitigate value overestimation bias and improve learning stability: clipped double Q-learning, delayed policy updates, and target policy smoothing.
The algorithm specifically tackles the overestimation bias inherent in Q-learning-based methods [14] through its twin critic architecture. TD3 maintains two separate Q-networks (Q_{ϕ_1} and Q_{ϕ_2}) with independent parameters, using the minimum of their estimates to form the target value:

$$y = r_t + \gamma \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s_{t+1}, \tilde{a}_{t+1}) \tag{8}$$

where $\tilde{a}_{t+1}$ is a noise-perturbed action from the target policy:

$$\tilde{a}_{t+1} = \pi_{\theta_{\text{targ}}}(s_{t+1}) + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0,\sigma), -c, c) \tag{9}$$

This target policy smoothing regularization prevents overfitting to narrow Q-value peaks by adding clipped noise to the target action. The actor π_θ is then updated to maximize the expected Q-value from the first critic network:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\nabla_a Q_{\phi_1}(s_t, a)|_{a=\pi_\theta(s_t)} \nabla_\theta \pi_\theta(s_t) \right] \tag{10}$$

[4]Introduced by RLcochet from previous years.

Another innovation in TD3 is the delayed policy update mechanism, where the actor and target networks are updated less frequently than the critics (2:1 update ratio in our implementation) allows for more stable Q-function learning before policy improvements. The complete algorithm can be split into the following components:

3.1.1 Extension: Pink Noise

Exploration noise is a critical component in most reinforcement learning settings, preventing premature convergence to suboptimal policies. While white noise (often implemented as Gaussian noise) is commonly used, temporally correlated noise can improve exploration by making the sequence of actions smoother, so changes between actions are more gradual and natural.

What is Pink Noise and Why Use It? Pink noise, or $1/f$ noise, has a power spectral density that decreases with frequency, meaning low-frequency components dominate. Unlike white noise, which has equal power across all frequencies, pink noise produces more gradual variations, resulting in smoother transitions between actions. This property is especially useful in tasks where sudden changes in actions can be harmful. The smoother fluctuations of pink noise help maintain steady exploration, reducing unpredictable behavior and improving control. Recent studies [4] suggest that pink noise outperforms other noise types in certain problems, making it a compelling choice.

3.1.2 Extension: Random Network Distillation for Enhanced Exploration

Sparse or deceptive reward structures can be difficult for policy-gradient methods, as standard exploration strategies may not visit novel states enough. **Random Network Distillation** (RND) [3] provides an intrinsic reward that guides exploration toward unfamiliar regions of the state space. The approach uses two networks:
The **Target Network**, f_{rnd}, is a randomly initialized, fixed network that processes states and outputs a feature representation, with its parameters remaining unchanged during training. On the other hand, the **Predictor Network**, f_{pred}, is a trainable network with the same architecture as f_{rnd}, which learns to mimic the target network's outputs.
At each timestep, the intrinsic reward is computed as the squared error between the predictor's and target's outputs:

$$r_{\text{rnd}}(s_t) = \|f_{\text{pred}}(s_t) - f_{\text{rnd}}(s_t)\|^2.$$

Intuitively, rarely encountered states produce higher prediction errors, yielding larger intrinsic rewards that encourage exploration. As the predictor network learns these states, the error—and hence the intrinsic reward—decreases, preventing repeated exploitation of the same novelty.

Integration with TD3. To incorporate RND into TD3, the total reward is augmented by adding a scaled intrinsic reward to the environment's extrinsic reward:

$$r_{\text{total}} = r_t + \alpha_{\text{rnd}}\, r_{\text{rnd}}(s_t),$$

where α_{rnd} controls the influence of intrinsic rewards. This combined reward drives both critic and actor updates in TD3. Empirically, RND has shown promise in tasks that require extensive state-space coverage, proving effective in both Atari and other domains [3, 10, 1].

3.1.3 Extension: Layer Normalization

Layer normalization [2] re-centers and rescales neuron activations within a layer. For a layer output $\mathbf{h} \in \mathbb{R}^d$, it computes

$$\hat{\mathbf{h}} = \frac{\mathbf{h} - \mu_h}{\sigma_h} \alpha + \beta,$$

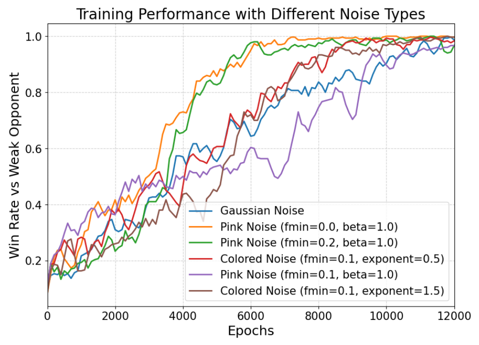
where μ_h and σ_h are the mean and standard deviation of $\mathbf{h}$ (computed across the feature dimension), and α and β are learned parameters. Unlike batch normalization, which normalizes across a batch, layer normalization standardizes each sample's features. This makes the learning process more robust to outlier activations and exploding gradients, which is particularly useful in off-policy actor-critic methods like TD3, where replay buffer experiences can vary significantly. As a result, layer normalization stabilizes updates and often leads to faster convergence with less sensitivity to hyperparameter settings [11]. Layer norm was implemented between all layers of both the actor and critic networks.

3.2 Experiments

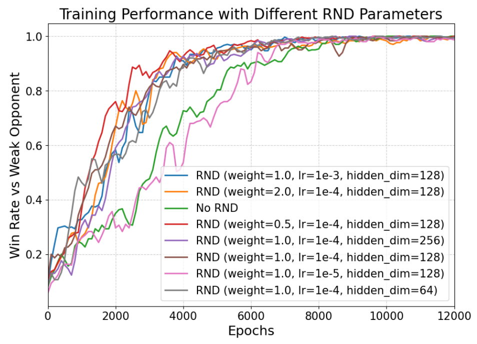
We conduct an ablation study to identify the most effective hyperparameters. Training is done against a weak opponent for 12,000 epochs, evaluating every 100 epochs over 1,000 matches to compute win rates. A moving average with a window size of 3 smooths the curves. Training accuracy refers to performance against the weak opponent. Default hyperparameters and architecture details are in the Appendix.

3.2.1 Noise Experiments

First, we examine the impact of exploration noise, testing six noise configurations with a focus on pink noise. To ensure completeness, we include two additional β values. The key hyperparameters are f_{min}, which defines the minimum frequency component (higher values reduce low-frequency dominance), and β, which controls the spectral slope ($\beta = 1.0$ for standard pink noise, with higher values emphasizing lower frequencies). Other extensions are disabled.



(a) Training Win Rates with Different Noise Types



(b) Training Win Rates in RND Experiments

Figure 4: Comparison of Training Performance Across Noise and RND Experiments

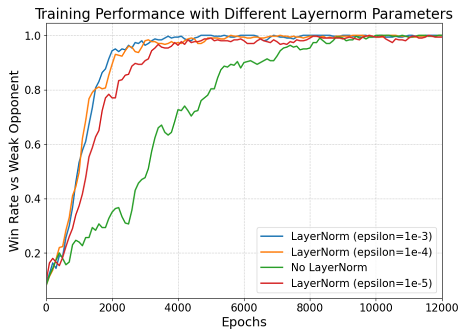
As shown in Figure 4a, all experiments eventually overfit, achieving near-optimal performance. Thus, the key comparison is convergence speed. Pink noise with $f_{\text{min}} = 0.0$ and $f_{\text{min}} = 0.2$ improved the fastest, reaching a 0.95 win rate around 6000 epochs, while $f_{\text{min}} = 0.1$ lagged behind, failing to reach this threshold. Overall, $f_{\text{min}} = 0.0$ was the best-performing configuration, maintaining the highest win rate beyond 6000 epochs.

3.2.2 RND Experiments

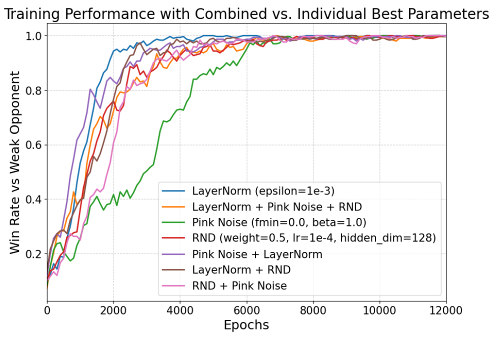
We evaluate the impact of Random Network Distillation (RND) parameters on training performance, testing different weights, learning rates, and hidden dimensions against a no-RND baseline.
As shown in Figure 4b, RND accelerates convergence, with the no-RND baseline lagging behind until 7000 epochs. One of the best configurations with weight $= 1.0$, lr $= 10^{-3} ad$hidden dimension $= 128$ reaches the 0.95 win rate the fastest. Lower learning rates and smaller hidden layers slow learning, while excessively large ones offer no advantage. Overall, it can be said that RND improves learning efficiency.

3.2.3 Layer Normalization Experiments

In this study, we analyze the impact of LayerNorm with different ϵ values on training performance. We compare three settings ($\epsilon = 10^{-3}, 10^{-4}, 10^{-5}$) against a baseline without LayerNorm.



(a) Training Win Rates with Different Layernorm Parameters



(b) Training Win Rates of Combined Best vs Individual Best Parameters

Figure 5: Comparison of Training Performance Across Noise and RND Experiments

Results show that LayerNorm significantly accelerates convergence compared to the baseline, which lags behind until 7000 epochs. The best-performing configurations are $\epsilon = 10^{-3}$ and $\epsilon = 10^{-4}$, both reaching near-optimal win rates the fastest. $\epsilon = 10^{-5}$ shows slightly slower convergence but still outperforms the no-LayerNorm case. Overall, LayerNorm enhances learning speed, with $\epsilon = 10^{-3}$ providing the most stable and rapid convergence.

3.2.4 Combining the Best Parameters

To assess whether combining the best-performing configurations yields further improvements, we trained agents using all combinations of LayerNorm ($\epsilon = 10^{-3}$), Pink Noise ($f_{\min} = 0.0, \beta = 1.0$) and RND (weight $= 0.5$, lr $= 10^{-4}$, hidden dimension $= 128$). Figure 5b shows the result that the agent with just LayerNorm converges the fastest. As a result, the final agent for the tournament was trained using using only LayerNorm ($\epsilon = 10^{-3}$) against the weak and strong opponent, training and shooting sequentially for 100,000 episodes. Self-play was used to try to mitigate overfiting but ultimately it hindered performance so it was not used for the final agent.

3.3 DQN: Deep Q-Networks

The prevalent approach of approximating commonly encountered functions of intractable action and state spaces was mainly introduced with Deep Q-Networks (DQN) [12]. Specifically, DQNs approximate the optimal action-value function $Q^*(s, a)$ with a deep neural network.

The DQN agent interacts with an environment $\mathcal{E}$ in discrete time steps t. At each t, the agent chooses and action a_t from its discrete action space $\mathcal{A} = \{a_1, ..., a_{|\mathcal{A}|}\}$, and observes a reward r_t. Similar to SAC and TD3, DQN is an *off-policy* algorithm learning from sampled experience from a replay buffer. It is also *model-free* and does not require an internal representation of the environment [12]. Our implementation of the DQN action space is discussed in detail in Section 3.3.4.
The agent's aim is to maximize the expected *discounted return* at time step t, defined as
$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_t$. The hyperparameter γ is the *discount factor* regulating the importance of rewards from further time steps. The optimal action-value function approximated by the DQN can thus be expressed as:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) = \max_\pi \mathbb{E}[R_t|s_t = s, a_t = a, \pi]$$

Moreover, the state-value function for a policy π is defined as:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)]$$

Since $V^*(s) = \max_a Q^*(s,a)$ under the optimal deterministic policy, this function satisfies the Bellman equation, and it can be recursively expressed as [16]:

$$Q^*(s,a) = \mathbb{E}_s'[r + \gamma \max_{a'} Q^*(s',a')|s,a]$$

This allows the action value function to be computed with dynamic programming, and in DQNs, be approximated by a deep neural network: $Q(a,s;\theta)$ with the neural network parameters θ. Based on the recursive definition, the following sequence of loss functions at iteration i can be minimized to learn the approximator [16]:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(y_i^{DQN} - Q(s,a;\theta_i)\right)\right]$$

where the training objective y_i^{DQN} is

$$y_i^{\text{DQN}} = r + \gamma \cdot \max_{\alpha'} Q(s',a';\theta^-).$$

Here, θ^- represent the parameters of the "target network", which are kept fixed and updated once in every fixed number of iterations. Even though using the previous iteration's network parameters (i.e. updating the target network each iteration) is possible [12], in practice, this leads to a more instable training and a worse approximation [16]. Thus, the target network is kept fixed for several iterations, which leads to a more stable training and improves the trained agent's performance. This is evidenced by our experiments in Section 3.3.4, demonstrating how DQN without a target network performs worse. Our implementation additionally applies target updates with Polyak averaging in order to promote a more stable training:

$$\theta_{t+1}^- = \tau\theta_t + (1-\tau)\theta_t^-$$

Since its introduction, a variety of extensions and improvements to the DQN approach were proposed in the RL literature. A prominent work which gathers many of these additions and explores an optimal combination is the Rainbow algorithm [8]. By bringing together seven main improvements, Rainbow demonstrates better performance than each method considered individually. Among these additions, **Double DQN**, **Dueling DQN**, and **Prioritized Experience Replay** were chosen and implemented for our DQN experiments.

3.3.1 Extension (1): Double DQN

DQN's training objective y_i^{DQN} can provide overly-optimistic estimates as the $\max$ operator uses the same values for selecting and evaluating an action. Double DQN replaces this training objective with the following to compensate for possible over-estimations [14]:

$$y_i^{\text{DDQN}} = r + \gamma \cdot Q(s', \arg\max_{a'} Q(s', a';\, \theta_i);\, \theta^-)$$

Double DQN decouples action selection and evaluation by evaluating the selected action with the target network. The rest of the learning procedure is kept the same as DQN, and empirical results confirm that introducing Double Q-learning to DQN improves performance [14].

3.3.2 Extension (2): Dueling DQN

The Dueling DQN architecture makes use of the advantage function, which obtains a relative measure for each action's importance [16]:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

Deriving from this, the Q estimator can be split into two streams estimating the advantage and state value functions:

$$Q(s,a;\, \theta,\alpha,\beta) = A(s,a;\, \theta,\alpha) + V(s;\, \theta,\beta) \tag{11}$$

The main intuition behind this new architecture is to avoid evaluating the value of each action choice when it is not necessary [16]. Instead of the estimator consisting entirely of fully connected layers, the later layers are split into two streams capable of separately estimating the action and value functions. Their outputs are then merged to form the Q function estimation.
However, action and value functions cannot be recovered uniquely for a given Q if merging is done as in Equation 11. To ensure identifiability and thus improve performance, the estimator A is forced to have a zero advantage for the chosen action:

$$Q(s,a;\, \theta,\alpha,\beta) = \left(A(s,a;\, \theta,\alpha) - \max_{a' \in \mathcal{A}} A(s,a';\, \theta,\alpha) \right) + V(s;\, \theta,\beta)$$

This arises from the fact that for a deterministic policy $a^* = \arg\max_{a' \in \mathcal{A}} Q(s,a')$, it holds that $Q(s,a^*) = V(s) \;\Rightarrow\; A(s,a^*) = 0$. In the new equation, for $a^* = \arg\max_{a' \in \mathcal{A}} Q(s,a';\, \theta,\alpha,\beta)$
$= \arg\max A(s,a';\, \theta,\alpha)$, it holds that $Q(s,a^*;\, \theta,\alpha,\beta) = V(s;\, \theta,\beta)$. To stabilize training, the authors suggest replacing the max operator with an average:

$$Q(s,a;\, \theta,\alpha,\beta) = \left(A(s,a;\, \theta,\alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a';\, \theta,\alpha) \right) + V(s;\, \theta,\beta)$$

Even though this introduces an offset to the original learning target, it improves the learning process and the agent's performance in practice. We have implemented DDQN using this form of the equation as well.

3.3.3 Extension (3): Prioritized Experience Replay

DQN uses experience replay, which provides data-efficiency as it re-uses previous experiments [16]. As explained in Section 2.1, Prioritized Experience Replay (PER) changes the random sampling of experience replay by prioritizing experience based on their temporal difference error. This leads to faster and more accurate convergence during training in a variety of off-policy alogirthms, including DQN agents [13, 16].

3.3.4 Experiments

Experiment Setting Each agent was trained for a maximum of 100K episodes. The primary evaluation metric for the success of compared DQN agents was the mean win-rate against the basic strong opponent in multiple games. Our later tests included training and evaluation against a competent SAC agent of our team. This was done to provide an alternative to the strong basic opponent and to represent competent tournament entries. Evaluation runs were done every 500 or 1000 episodes of agent training, as well as at the end of the training. Evaluation statistics were gathered on 100 or 200 games on each evaluation. **[TODO: all plots are smoothed using...]**

DQN addition selection Each DQN variant was implemented in their own class, which were then compared in different settings such as using different opponents, a customized action space, utilizing self-play, and using PER. A comparison of the base DQN additions against the strong basic opponent are provided in Figure 6. As the figure also suggests, our empirical results show that Double Dueling DQN (thereby shortened as 3DQN) is a prominent combination of additions. **[TODO: It also succeeds with our other more minor additions, as discussed in later subsections.]**

Custom action space **[TODO: kısalt]** DQN agents use a discrete action space $\mathcal{A} = \{a_1, ..., a_{|\mathcal{A}|}\}$, where each a_i is a predefined action on hockey environment. The default discrete action space of the hockey environment is limiting with only 7 base movements, representing movement along axes, rotation, and puck shooting. We have extended this action space to include combinations of these basic movements, providing flexible movement. To keep the action space as simple as possible and facilitate learning while maintaining flexibility, we have used a subset of 20 actions among possible combinations. The actions are provided in Appendix Section C.2. A comparison of win-rates of 3DQN agents against strong basic opponent using the default and custom action spaces are provided in Figure 7.

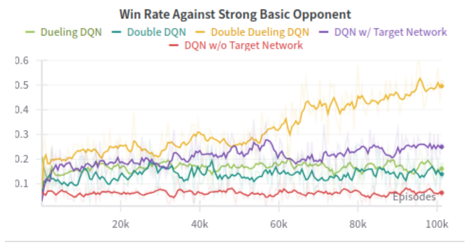


Figure 6: Win rates of base DQN additions vs. strong basic opponent

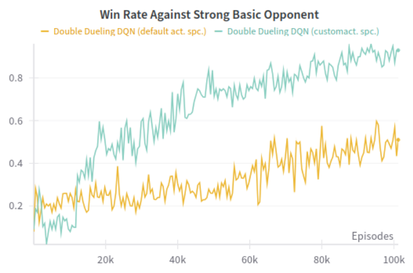


Figure 7: Win rates with different action spaces vs. strong basic opp.

Prioritized Experience Replay Our main observation of PER with DQN methods is that it provides a time-wise slower but iteration-wise quicker learning period against a given opponent. Figure 8 compares the learning schedules two 3DQN agents: One without PER trained for 32 iterations in each episode, and one with PER trained for 4 iterations each episode. It can be seen that PER helps the agent achieve a superior performance in a less number of training iterations.

Self-play Self-play was first implemented as two DQN agents initialized from scratch learning to play against each other. This provided insight on the learning capabilities of DQNs, but ultimately led to worse performance on the basic opponents. In this setting, models either struggled to win against each other with a draw-rate near 1.0, or consistently won against each other with a player win-rate near 0.5. Another type-of self play was implemented with POB as introduced in Section **??**

Hyperparameter selection 3DQN was decided as the best combination of additions, and the hyperparameters were decided on for this DQN version. Among values $\{3e-4, 3e-5, 3e-6, 3e-7\}$, a learning rate of $3e-5$ yielded the highest winrate. **[TODO: larger model exp.s - wasn't better? check]** To encourage exploration, ϵ-decay was applied in the ϵ-greedy algorithm: Starting with a high value, it was decayed at each episode by multiplying it with a decay rate close to 1 until it is at the minimum allowed value. Due to time constraints, other hyperparameters were used mostly only with their default values. Default and most commonly used hyperparameter values are provided in Appendix Section C.1.

Empirical results? **[TODO: how is the latest model designed like, how was it trained, how does it behave. show stats against multiple opp.s: wr against weak, strong, sac; draw rate against weak, strong sac... basically summarize fig 9]**

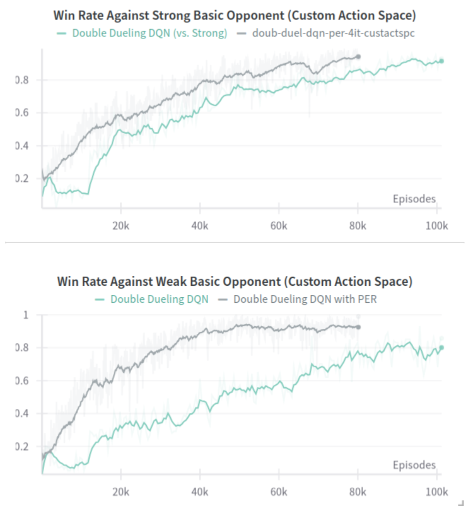


Figure 8: Win rates of 3DQNs with and without PER vs. basic opponents

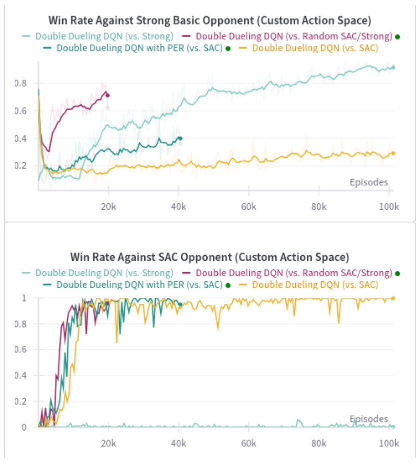


Figure 9: Win rates of 3DQN variations vs. SAC opponent

Final Competition

We tested our models against each other. **[TODO:]**

References

[1] Alain Andres, Esther Villar-Rodriguez, and Javier Del Ser. An Evaluation Study of Intrinsic Motivation Techniques Applied to Reinforcement Learning over Hard Exploration Environments, page 201–220. Springer International Publishing, 2022. ISBN 9783031144639. doi: 10.1007/978-3-031-14463-9_13. URL http://dx.doi.org/10.1007/978-3-031-14463-9_13.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

[3] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation, 2018. URL https://arxiv.org/abs/1810.12894.

[4] Onno Eberhard, Jakob Hollenstein, Cristina Pinneri, and Georg Martius. Pink noise is all you need: Colored noise exploration in deep reinforcement learning. In The Eleventh International Conference on Learning Representations, 2023. URL https://openreview.net/forum?id=hQ9V5QN27eS.

[5] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018. URL https://arxiv.org/abs/1802.09477.

[6] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL https://arxiv.org/abs/1801.01290.

[7] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications, 2019. URL https://arxiv.org/abs/1812.05905.

[8] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017. URL https://arxiv.org/abs/1710.02298.

[9] L. Kocsis and C. Szepesvári. Discounted ucb. In 2nd PASCAL Challenges Workshop, volume 2, pages 51–134, 2006.

[10] Pawel Ladosz, Lilian Weng, Minwoo Kim, and Hyondong Oh. Exploration in deep reinforcement learning: A survey. Information Fusion, 85:1–22, September 2022. ISSN 1566-2535. doi: 10.1016/j.inffus.2022.03.003. URL http://dx.doi.org/10.1016/j.inffus.2022.03.003.

[11] Clare Lyle, Zeyu Zheng, Khimya Khetarpal, James Martens, Hado van Hasselt, Razvan Pascanu, and Will Dabney. Normalization and effective learning rates in reinforcement learning, 2024. URL https://arxiv.org/abs/2407.01800.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

[13] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016. URL https://arxiv.org/abs/1511.05952.

[14] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015. URL https://arxiv.org/abs/1509.06461.

[15] Che Wang and Keith Ross. Boosting soft actor-critic: Emphasizing recent experience without forgetting the past, 2019. URL https://arxiv.org/abs/1906.04009.

[16] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016. URL https://arxiv.org/abs/1511.06581.

Appendix

A SAC: Architectural and Training Details

A.1 Training Hyperparameters

Table 1 lists the default hyperparameters used during training.

Parameter	**Value**
Number of hidden layers	2
Width of each hidden layer	256
Non-linearity	ReLU
Discount factor (γ)	0.99
Target update rate (τ)	0.005
Target update interval	1
Optimizer	Adam
Number of updates	Same as episode length K
Learning rate (η)	10^{-3}
Entropy coefficient (α)	Auto-tuned
Policy noise	0.2
Noise clip	0.5
Policy frequency	2
Batch size	256
Exploration noise exponent (β)	1.0 (Pink Noise)
Exploration noise deviation (σ)	0.1
Replay buffer size	10^6
D-UCB history window length (τ)	1000
D-UCB bonus factor (ξ)	1
D-UCB discount factor (γ)	0.95
D-UCB reward upper-bound (B)	1

Table 1: Default training hyperparameters.

B TD3: Architectural and Training Details

B.1 Training Hyperparameters

Table 2 lists the default hyperparameters used during training.

Parameter	**Value**
Discount Factor (γ)	0.99
Target Update Rate (τ)	0.005
Policy Noise	0.2
Noise Clip	0.5
Policy Frequency	2
Start Timesteps	1000
Evaluation Frequency	100
Batch Size	2048
Exploration Noise Type	Pink
Exploration Noise Scale	0.1
Pink Noise Exponent	1.0
Pink Noise f_{min}	0.0
Layer Norm ϵ	1×10^{-5}
Save Model Frequency	10 000 episodes
RND Weight	1.0
RND Learning Rate	1×10^{-4}
RND Hidden Dimension	128
Max Episode Steps	600

Table 2: Default training hyperparameters.

B.2 Network Architectures

Actor Network:

- **Input:** State vector (dimension d_{state}).
- **Hidden Layer 1:** Fully-connected layer with 256 units, optionally followed by layer normalization, and a ReLU activation.
- **Hidden Layer 2:** Fully-connected layer with 256 units, optionally followed by layer normalization, and a ReLU activation.
- **Output Layer:** Fully-connected layer mapping to the action vector (dimension d_{action}) with a $\tanh$ activation scaled by the maximum action.

Critic Network:

- Implements two separate Q-networks (denoted as Q1 and Q2) for the purpose of double Q-learning.
- **Input:** Concatenated state and action vector (dimension $d_{\text{state}} + d_{\text{action}}$).
- For each Q-network:
  - **Hidden Layer 1:** Fully-connected layer with 256 units, optionally followed by layer normalization, and a ReLU activation.
  - **Hidden Layer 2:** Fully-connected layer with 256 units, optionally followed by layer normalization, and a ReLU activation.

  - **Output Layer:** Fully-connected layer with 1 unit (Q-value).

RND (Random Network Distillation) Networks:

- Both the target and predictor networks share the same architecture.
- **Architecture:**
  - **Layer 1:** Fully-connected layer from input (state vector, dimension d_{state}) to 128 units, with ReLU activation.
  - **Layer 2:** Fully-connected layer from 128 to 128 units, with ReLU activation.
  - **Layer 3:** Fully-connected layer from 128 to 128 units.

B.3 Additional Details

- **Replay Buffer:** Maintains up to 10^6 transitions for experience replay.
- **Actor and Critic Optimizers:** Both use the Adam optimizer with a learning rate of 3×10^{-4}.
- **Target Networks:** Soft updates are performed using the parameter $\tau = 0.005$.
- **Delayed Policy Updates:** The actor network is updated once every 2 critic updates (i.e., policy frequency is 2).

C DQN: Architectural and Training Details

C.1 Default Training Hyperparameters

[TODO: hparams, including layer sizes]

C.1.1 Network Architecture

C.2 Custom Discrete Action Space

[TODO: list and explain shortly]