Application of Deep Reinforcement Learning to the Slime Volleyball Gym Environment

Matthijs van der Lende (s4325621) Niclas Müller-Hof (s4351495)
Group 21

University of Groningen

Abstract

In this project, we apply two classes of model-free reinforcement learning algorithms to the Slime Volleyball Gym environment. For value function-based methods, which first learn an (action)-value function and then derives a policy (e.g. ϵ-greedy), deep Q-network (DQN) is applied. Policy gradient methods that learn a policy directly include the elementary algorithms presented in Sutton and Barto (2018): REINFORCE, REINFORCE with baseline and one-step actor-critic. The latter two are actor-critic methods, where the "actor" learns the policy, and a "critic" learns a value function that can be used to estimate how good a state is. Although none of the algorithms beat the (expert) baseline policy, they all performed better than a random policy.

Contents

1 Introduction **2**

2 Methods **2**
2.1 Environment . . . 2
2.2 Feature Engineering . . . 3
2.3 Model Architecture and Preprocessing . . . 3
2.3.1 Convolutional Neural Network . . . 3
2.3.2 Multilayer Perceptron . . . 4
2.4 Deep Q-Network . . . 4
2.4.1 General Policy Iteration . . . 5
2.5 REINFORCE, REINFORCE with Baseline and One-Step Actor-Critic . . . 6
2.5.1 Gradient Descent Formulation . . . 6
2.6 Hyperparameter tuning . . . 7

3 Results **7**

4 Discussion **8**

References **9**

A Pseudocode **10**

B Derivation REINFORCE **11**

C Abbereviations used for Reinforcement Learning Algorithms **12**

D Summary of Notation **13**

E Loss Graphs **14**

1 Introduction

(Deep) reinforcement learning has been a powerful tool in the world of video games, leading to groundbreaking achievements such as DeepMind's ability to train an agent to play 49 Atari 2600 games and OpenAI's defeat of world champions in Dota 2 (Mnih, 2015), (Berner et al., 2019).

Deep reinforcement learning is a subfield of reinforcement learning (RL) where function approximation is used for important functions within the theory of RL, like the value function $v_\pi(s)$, the action-value function $q_\pi(s, a)$ and the policy $\pi(a|s)$ and where the function approximators are deep neural networks, like convolutional neural networks (Mnih et al., 2013) and more recently, transformer models (Chen et al., 2021). In RL, the goal is to learn an optimal policy π_* that gives the highest expected (discounted) reward (return) in the environment in which the agent is placed. The environment is traditionally modelled as a (finite) Markov decision process $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ with states $s \in \mathcal{S}$, a description of the state of the world and the agent, actions $a \in \mathcal{A}$ describing what the agent can do. A transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ describing how the environment evolves over time as the agent interacts with it and a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ that returns a reward for a state-action-new-state transition, which is the crucial element that allows reinforcement agents to *learn* policies π. In model-free reinforcement learning, the agent has no knowledge of $\mathcal{P}, \mathcal{R}$, and the agent has to cope with just the rewards it receives from the environment. Two important classes of model-free reinforcement learning techniques (which can involve function approximation) are value function-based methods, like Deep Q-network (Mnih et al., 2013), that first learn an (action)-value function $\hat{q}_\pi(s, a, \theta)$, and then derive a policy (e.g., ϵ-greedy). A procedure called general policy iteration (see section 2.4.1) is used to end up with the optimal value function $q_{\pi_*} = q_*$ from which the optimal policy $\pi_*(s) = \text{argmax}_{a \in \mathcal{A}} q_*(s, a)$ can be derived. In addition, there are policy gradient methods that learn a policy π directly through a parameterized policy $\pi(a|s, \theta)$, where the final goal is also to have the agent learn the optimal policy, so $\pi(a|s, \theta) \approx \pi_*(a|s)$. More recently (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017) a form of policy gradient called actor-critic methods are being used where the "actor" (the agent) learn $\pi(a|s, \theta)$ and the "critic" learns $\hat{v}(s, \mathbf{w})$. Information from a value function $\hat{v}(s, \mathbf{w})$ helps an agent estimate how good its policy is.

Four reinforcement learning algorithms/agents are applied to the Slime Volleyball Gym environment, a simple but challenging game available as a third-party OpenAI Gym environment. The agent's goal is to land the ball on the opponent's side and win the game, causing the opponent to lose all of its lives. We will use a combination of value function-based and policy gradient-based reinforcement learning methods. From the value function-based approach, deep Q-network (Mnih et al., 2013) is used. And from the policy gradient-based approach, three elementary algorithms from Sutton and Barto (2018): REINFORCE, REINFORCE with baseline, and one-step actor-critic. The latter two being actor-critic algorithms. For each algorithm, we use a convolutional neural network (CNN) and a multilayer perceptron (MLP) as function approximators.

2 Methods

2.1 Environment

The environment that the reinforcement learning agents will be placed in is a third-party environment of the OpenAI gym library. Slime Volleyball Gym (abbreviated as Slimevolleygym) is a simple volleyball game environment where the agent's goal is to get the ball on the ground of its opponent, in which case the opponent loses a life. The agent wins an episode when the opponent has lost all (4) lives (and vice-versa). If no agent has won the game after 3000 time steps, then the one with the most lives wins. Different configuration of the Slime Volleyball environment exists. The class that encapsulates the environment has three flags:

1. `from_pixels`: If set to true, then the environment returns game frames as observations (states), which can then be pre-processed (see section 2.3.1). If set to false, hand-crafted features are returned (see section 2.2).

2. `atari_mode`: If set to true, then actions are represented by integers. Otherwise, actions are one-hot encoded.

3. `survival_bonus`: If set to true, then the agent gets a small survival reward to facilitate early learning.

The `SlimeVolleySurvivalNoFrameskip-v0` preset is used to process states as frames. Which sets `from_pixels = True`, `atari_mode = True` and `survival_bonus = True`. When using MLPs, the same setup is used, except that `from_pixels = False` so that the handcrafted feature vectors are used. We can formalize the environment as a Markov Decision Process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$:

- S is a set of preprocessed $4 \times 84 \times 84$ frames from the game.
  - Alternatively $S \subseteq \mathbb{R}^{12}$ (see section 2.2).
- $\mathcal{A} = \{0, 1, 2, 3, 4, 5\}$. The agent can jump, go right, or go left in the game.
- $\mathcal{P}$ represents how the game evolves over time.
- $\mathcal{R}$ is the reward function. The maximum possible average score (undiscounted return) is 5. As a reference, a proximal policy optimization (PPO) agent over 1000 episodes has an average score of 0.435 ± 0.961 when trained on image data (Ha, 2020).

All the presented reinforcement learning algorithms are model-free. Meaning that the agent never knows about $\mathcal{P}$ and $\mathcal{R}$.

Figure 1: The Slime Volleyball Gym environment. The red dot is the ball. At the top-left, the number of lives of the left agent is displayed, and at the top-right, the number of lives of the right agent is displayed. In the middle, there is a net, which the agents cannot cross.

2.2 Feature Engineering

Slime Volleyball Gym provides predefined feature vectors for representing the states of the environment. In this case a state $s \in \mathcal{S} \subseteq \mathbb{R}^{12}$. The 12-dimensional feature vector contains the position and velocity of relevant objects/agents in the game:

$$(x_{\text{agent}}, y_{\text{agent}}, \dot{x}_{\text{agent}}, \dot{y}_{\text{agent}}, x_{\text{ball}}, y_{\text{ball}}, \dot{x}_{\text{ball}}, \dot{y}_{\text{ball}}, x_{\text{opponent}}, y_{\text{opponent}}, \dot{x}_{\text{opponent}}, \dot{y}_{\text{opponent}})^T, \tag{1}$$

where $\dot{x}$ is the derivative of position with respect to time (i.e. velocity).

2.3 Model Architecture and Preprocessing

2.3.1 Convolutional Neural Network

Convolutional neural networks (CNNs) can be used for function approximation and can be trained on image data from a game without having to create handcrafted features. This also makes it easier to train an agent on different games. Let $\theta_q, \theta_\pi, \theta_v$ denote the parameters of CNNs with the same architecture, except for the final layer, which is used to approximate $q(s, a), \pi(a|s), v(s)$ respectively. We now describe how input is passed through the network. First, the input is preprocessed in the same way as

described by Mnih et al. (2013). Four frames from the game are taken. Then for each frame, they are converted to grayscale and then downsampled and cropped to a resolution of 84×84. N batches of $4 \times 84 \times 84$ images are provided to the network, resulting in a $N \times 4 \times 84 \times 84$ tensor. For each $4 \times 84 \times 84$ tensor $\mathbf{X}$ in the batch the network does the following: the first three hidden layers are convolution layers. In the first layer, 2D cross-correlation is performed between $\mathbf{X}$ (each channel) and an 8×8 kernel matrix W_1 with a stride (how far the filter moves from one position to the next) of 4, followed by a ReLU activation function ($\text{ReLU}(x) = \max(0, x)$) which is applied element-wise. The output of the first convolution layer transforms the $4 \times 84 \times 84$ into a $32 \times 20 \times 20$ tensor $\mathbf{X}'$. In the next convolution layer $\mathbf{X}'$ is processed in the same way, except that the kernel matrix W_2 is 4×4 and has a stride of 2. The output of the second convolution layer results in a $64 \times 9 \times 9$ tensor $\mathbf{X}''$. Similarly, the third convolution layer has a 3×3 kernel matrix W_3 with a stride of 1, resulting in a $64 \times 7 \times 7$ tensor $\mathbf{X}'''$. $\mathbf{X}'''$ is then flattened into a vector $\mathbf{x} \in \mathbb{R}^{3136}$ which is then passed through two final linear layers:

$$\mathbf{x}' = \text{ReLU}(W_4\mathbf{x}) \tag{2}$$

$$\mathbf{y} = f(W_5\mathbf{x}'), \tag{3}$$

where $W_4 \in \mathbb{R}^{3136 \times 512}$ and so $\mathbf{x}' \in \mathbb{R}^{512}$. The final layer differs depending on which function we wish to approximate:

- For $\hat{q}(s, a, \theta_q)$, $W_5 \in \mathbb{R}^{512 \times 5}$ and $f(x) = x$. This results in a vector that contains the q-values for each action (given the state).
- For $\pi(a|s, \theta_\pi)$, $W_5 \in \mathbb{R}^{512 \times 5}$ and $f(\mathbf{x}) = \text{softmax}(\mathbf{x}) = 1/Z(\exp(x_1), ..., \exp(x_d))^T$, resulting in a probability vector giving the probability for each action. $Z = \sum_{i=1}^{d} \exp(x_i)$ is the normalization constant making the elements of the vector sum to one.
- For $\hat{v}(s, \theta_v)$, $W_5 \in \mathbb{R}^{512 \times 1}$ and $f(x) = x$. Resulting in a scalar that represents the value of the state.

2.3.2 Multilayer Perceptron

When not using image data (frames), but handcrafted feature vectors for the states in the environment, a multilayer perceptron (MLP) can be used as a function approximator. The architecture of the MLP has two hidden layers, and the input is processed as follows: given an input feature vector $s \in \mathcal{S} \subseteq \mathbb{R}^{12}$:

$$\begin{aligned} &\text{ReLU}(W_1 s) = \mathbf{x}_1 \in \mathbb{R}^{128} \quad \text{where } W_1 \in \mathbb{R}^{12 \times 128} \\ &\text{ReLU}(W_2 \mathbf{x}_1) = \mathbf{x}_2 \in \mathbb{R}^{256} \quad \text{where } W_2 \in \mathbb{R}^{128 \times 256} \\ &f(W_3 \mathbf{x}_2) = \mathbf{y}. \end{aligned} \tag{4}$$

The final layer is adjusted in the same way as in section 2.3.1 to approximate either q, π or v.

2.4 Deep Q-Network

Deep Q-networks (DQN) uses a semi-gradient/stochastic-gradient form of Q-learning. The standard Q-learning update rule is given by:

$$Q(S_t, A_t) := Q(S_t, A_t) + \alpha[\underbrace{R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a)}_{\text{TD target}} - Q(S_t, A_t)], \tag{5}$$

where Q directly approximates the optimal value function q_*[1], independent of the behavioral policy of the agent. $(S_t, A_t, R_{t+1}, S_{t+1})$ is a trajectory of random variables that return the state, action, reward, and next state, respectively. $\gamma \in (0, 1]$ is the discount factor and $\alpha \in (0, 1]$ is the learning rate (see section D for a summary of notation).

DQN modified the Q-learning algorithm into a supervised learning problem, where an approximation $\hat{q}(s, a, \theta)$ of the action-value function is learned. At the end of the learning algorithm, we want

[1] All optimal policies π_* share the same action-value function.

$\hat{q}(s, a, \theta) \approx q_*(s, a)$, which is then used to derive the optimal policy π_* (see section 2.4.1). Here q_* is the optimal action-value function, and $\hat{q}$ is the approximation by the model.

θ is learned by minimizing the objective function $\mathcal{L}$, which is the mean squared error loss of the temporal difference (TD) target and the model output:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle \sim \mathcal{D}} \big[(R_{t+1} + \gamma \max_{a \in \mathcal{A}} \hat{q}(S_{t+1}, a, \theta^-) - \hat{q}(S_t, A_t, \theta))^2 \big]. \tag{6}$$

There are a few notable improvements that Mnih et al. (2013) have made over standard Q-learning with function approximation. First, trajectories are randomly sampled (in minibatches) from an experience replay buffer $\mathcal{D}$, reducing the correlation between trajectories. In addition, after n gradient updates, the primary network's parameters θ are copied to a target network θ^-. This improves performance and stability because supervised learning theory assumes a stationary distribution and independent and identically distributed (i.d.d.) samples.

The expectation in equation 6 is unknown and is estimated by samples collected from the experience replay buffer $\mathcal{D}$. The actual objective function, given a minibatch of trajectories $(S_i, A_i, R_i, S'_i)_{i=1,\dots,N}$ drawn uniformly from the experience replay buffer, becomes:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1,\dots,N} (R_i + \gamma \max_{a \in \mathcal{A}} \hat{q}(S'_i, a, \theta^-) - \hat{q}(S_i, A_i, \theta))^2, \tag{7}$$

where N is the batch size. In this case the gradient $\nabla \mathcal{L}(\theta)$ can be written as a sum of gradients:

$$\begin{aligned} \nabla \mathcal{L}(\theta) &= \nabla \big(\tfrac{1}{N} \textstyle\sum_{i=1,\dots,N} (R_i + \gamma \max_{a \in \mathcal{A}} \hat{q}(S'_i, a, \theta^-) - \hat{q}(S_i, A_i, \theta))^2 \big) \\ &= \tfrac{1}{N} \textstyle\sum_{i=1,\dots,N} \nabla (R_i + \gamma \max_{a \in \mathcal{A}} \hat{q}(S'_i, a, \theta^-) - \hat{q}(S_i, A_i, \theta))^2. \end{aligned} \tag{8}$$

The stochastic gradient descent update can be written as follows:

$$\begin{aligned} \theta_{t+1} &= \theta_t - \alpha' \nabla \mathcal{L}(\theta) \\ &= \theta_t - \alpha' \big[\tfrac{1}{N} \textstyle\sum_{i=1,\dots,N} \nabla (R_i + \gamma \max_{a \in \mathcal{A}} \hat{q}(S'_i, a, \theta^-) - \hat{q}(S_i, A_i, \theta))^2 \big] \\ &= \theta_t + \alpha' \big[\tfrac{2}{N} \textstyle\sum_{i=1,\dots,N} \big(R_i + \gamma \max_{a \in \mathcal{A}} \hat{q}(S'_i, a, \theta^-) - \hat{q}(S_i, A_i, \theta_t) \big) \nabla \hat{q}(S_i, A_i, \theta_t) \big], \end{aligned} \tag{9}$$

The $2/N$ is absorbed into the learning rate α so the final stochastic gradient descent update becomes:

$$\theta_{t+1} = \theta_t + \alpha \Bigg[\sum_{i=1,\dots,N} \big(R_i + \gamma \max_{a \in \mathcal{A}} \hat{q}(S'_i, a, \theta^-) - \hat{q}(S_i, A_i, \theta_t) \big) \nabla \hat{q}(S_i, A_i, \theta_t) \Bigg]. \tag{10}$$

DQN is associated with using a CNN for function approximation. In this case, the states S are images as described in section 2.3.1. DQN can be "downgraded" to Q-network (QN) by using another model, like the MLP in section 2.3.2 and using handcrafted feature vectors to represent the states.

2.4.1 General Policy Iteration

Finding the optimal policy π_* is done by repeatedly *evaluating* (E) the policy and then *improving* (I) the policy by making it greedy with respect to the new value function:

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \dots \xrightarrow{I} \pi_* \xrightarrow{E} q_*. \tag{11}$$

For *on-policy* control algorithms, like Sarsa, the policy that is being evaluated is the behavioral policy. DQN (and Q-learning) is *off-policy* (Mnih et al., 2013), meaning it does not approximate the behavioral policy but instead directly approximates the optimal value function $q_{\pi_*} = q_*$. The approximation $\hat{q}$ is updated as if the agent followed a greedy policy. Note that the optimal (deterministic) policy is a greedy policy with respect to the optimal action-value function:

$$\pi_*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} q_*(s, a). \tag{12}$$

In both cases (off-policy and on-policy), the behavioral policy, for example, ϵ-greedy, is responsible for sufficient state space exploration. With ϵ-greedy, the agent chooses a random action with probability ϵ and otherwise follows the greedy policy based on the current approximation of the value function q.

2.5 REINFORCE, REINFORCE with Baseline and One-Step Actor-Critic

With policy gradient methods, we directly approximate the (possibly stochastic) policy by a model θ:

$$\pi(a|s,\theta) = P(A_t = a|S_t = s, \theta_t = \theta). \tag{13}$$

We want this model to learn $\pi(a|s,\theta) \approx \pi_*(a|s)$. Training is done by maximizing the performance measure $J(\theta)$ by gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}. \tag{14}$$

In the episodic case, performance is defined as the true value function under π_θ, the policy that is learned by θ (given some arbitrary initial state s_0):

$$J(\theta) = v_{\pi_\theta}(s_0). \tag{15}$$

The gradient used is a stochastic estimate (the sample gradient) $\widehat{\nabla J(\theta)}$ whose expectation, according to the *policy gradient theorem*, is proportional to the gradient $\nabla J(\theta)$:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a)\nabla\pi(a|s,\theta) = \mathbb{E}[\widehat{\nabla J(\theta)}], \tag{16}$$

where $\mu(s)$ is the on-policy distribution under π and $q_\pi(s,a)$ is the state-action-value function under π. The update rule for REINFORCE can be derived as follows (see appendix B for the full derivation) (adding a discount factor $\gamma_t \in (0,1]$):

$$\nabla J(\theta) \propto \textstyle\sum_s \mu(s)\gamma_t \sum_a q_\pi(s,a)\nabla\pi(a|s,\theta) = \mathbb{E}_\pi\left[\gamma_t G_t \nabla \ln \pi(A_t|S_t,\theta)\right], \tag{17}$$

where $G_t = \sum_{k=0}^{T} \gamma^{k-t-1} R_{t+k}$ is the return. The stochastic gradient ascent algorithm is derived by sampling this expectation:

$$\theta_{t+1} = \theta_t + \alpha\gamma_t G_t \nabla \ln \pi(A_t|S_t,\theta). \tag{18}$$

A *baseline* $b(s)$, such as a value function $\hat{v}(S_t, \mathbf{w})$ approximated by a model $\mathbf{w}$, can be added which can make θ learn faster:

$$\theta_{t+1} = \theta_t + \alpha\gamma_t(G_t - \hat{v}(S_t,\mathbf{w}))\nabla \ln \pi(A_t|S_t,\theta). \tag{19}$$

Deriving the one-step actor-critic method is done in the same way temporal difference (TD) methods are derived from Monte Carlo methods: by replacing the full return G_t by the one-step return $G_{t:t+1} = R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w})$ (allowing the algorithm to update without waiting for an episode to finish):

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha\gamma_t(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w}))\nabla \ln \pi(A_t|S_t,\theta) \\ &= \theta_t + \alpha\gamma_t\delta_t \nabla \ln \pi(A_t|S_t,\theta). \end{aligned} \tag{20}$$

2.5.1 Gradient Descent Formulation

Learning θ and $\mathbf{w}$ can be formulated as trying to minimize two objective functions: the "actor loss" $\mathcal{L}_A$, for the model θ learning the policy and the "critic loss" $\mathcal{L}_C$, which is the square error loss of the target and the output of the model $\mathbf{w}$ that learns the value function:

$$\mathcal{L}_C(\mathbf{w}) := \begin{cases} (G - \hat{v}(S,\mathbf{w}))^2 & \text{for REINFORCE with baseline} \\ (R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w}))^2 & \text{for one-step actor-critic} \end{cases} \tag{21}$$

$$\mathcal{L}_A(\theta) := \begin{cases} -\left[G \ln \pi(A|S,\theta)\right] & \text{for REINFORCE} \\ -\left[(G - \hat{v}(S,\mathbf{w})) \ln \pi(A|S,\theta)\right] & \text{for REINFORCE with baseline} \\ -\left[(R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})) \ln \pi(A|S,\theta)\right] & \text{for one-step actor-critic.} \end{cases} \tag{22}$$

$\mathcal{L}_A(\theta) = -\widehat{J(\theta)}$ can be used instead of $\widehat{J(\theta)}$ to convert the update rule for the actor to a gradient descent step. Given the objective function $\mathcal{L}_C$, the stochastic gradient descent update for $\mathbf{w}$ becomes:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \alpha'\nabla\mathcal{L}_C(\mathbf{w}_t) = \mathbf{w}_t - \alpha'\nabla(U_t - \hat{v}(S,\mathbf{w}_t))^2 \\ &= \mathbf{w}_t + 2\alpha'(U_t - \hat{v}(S,\mathbf{w}_t))\nabla\hat{v}(S_t,\mathbf{w}_t) \\ &= \mathbf{w}_t + \alpha(U_t - \hat{v}(S,\mathbf{w}_t))\nabla\hat{v}(S_t,\mathbf{w}_t), \end{aligned} \tag{23}$$

where

$$U_t = \begin{cases} G_t & \text{for REINFORCE with baseline} \\ R_{t+1} + \gamma\hat{v}(S',\mathbf{w}) & \text{for one-step actor-critic} \end{cases} \tag{24}$$

2.6 Hyperparameter tuning

The hyperparameter tuning is based on informal experiments and research. For DQN/QN having a large memory buffer size $|\mathcal{D}|$ is important for getting good performance (around $10,000$ or more). In addition to a proper schedule for ϵ-greedy that (linearly) decays from 1 to 0.1 in a certain number of episodes. Batch sizes are typically around 32 and 64. For actor-critic methods, one good rule of thumb for the learning rates is that the learning rate of the actor should be lower than the critic. Intuitively because the estimated q-values of the critic are based on past policies, so the actor cannot "get ahead" of the critic. Low learning rates around 0.001 generally appear to perform the best for all algorithms.

3 Results

Each agent is trained for 1000 episodes against a baseline policy (Ha, 2015). Figure 2 shows the agents' performance as they train over 1000 episodes. The chosen hyperparameters can be seen in Table 1. The gradient-based methods all have the same learning rate for the actor and critic (if applicable), and both DQN (using a CNN) and QN (using a MLP) have the same set of hyperparameters. Every RL agent has the same discount factor γ. To show that each agent/algorithm performs better than random, a

Hyperparameters	
α (DQN, QN)	0.001
$\alpha^{\mathbf{w}}$ (critic)	0.005
α^{θ} (actor)	0.001
Parameter copying $\theta^{-} \leftarrow \theta$	20 (episodes)
Batch Size (DQN, QN)	64
$\lvert\mathcal{D}\rvert$ (DQN/QN memory buffer size)	10,000
Schedule ϵ-greedy (DQN, QN)	1 to 0.1 linearly in 400 episodes
γ (discount factor)	0.9

Table 1: Table showing the hyperparameter settings for the results.

two-sample one-tailed Wilcoxen rank-sum test[2] is used to compare the average score between a specific RL agent and a random policy:

$$H_0 : \mu_{\text{Agent}} = \mu_{\text{Random}}, \quad H_A : \mu_{\text{Agent}} > \mu_{\text{Random}}. \tag{25}$$

The results from the statistical significance test are displayed in Table 2.

Agent Type	$\hat{\mu}_{\text{Agent}}$	$\hat{\sigma}_{\text{Agent}}$	$\hat{\mu}_{\text{Random}}$	$\hat{\sigma}_{\text{Random}}$	W	p-value
REINFORCE-BL	1.140	.008	.225	.738	927962	$< 2.2 \cdot 10^{-16}$
QN	.857	.741	.225	.738	812710	$< 2.2 \cdot 10^{-16}$
AC	.642	.871	.225	.738	689359	$< 2.2 \cdot 10^{-16}$
REINFORCE-MLP	.556	.652	.225	.738	733049	$< 2.2 \cdot 10^{-16}$
REINFORCE-BL-MLP	.515	.650	.225	.738	722862	$< 2.2 \cdot 10^{-16}$
DQN	.477	.5	.225	.738	730938	$< 2.2 \cdot 10^{-16}$
AC-MLP	.496	.624	.225	.738	719490	$< 2.2 \cdot 10^{-16}$
REINFORCE	.294	.737	.225	.738	553471	$1.424 \cdot 10^{-5}$

Table 2: Table showing the results of a two-sample one-tailed Wilcoxen rank-sum test ($\alpha = 0.005$) comparing the average score for each agent against a random policy. Here $\hat{\mu}$ denotes the sample mean and $\hat{\sigma}$ the sample standard deviation. W is the value of the W-statistic that is used in the Wilcoxen rank-sum test. See appendix C for an overview of the abbreviations used for each agent type.

[2]The scores of the agents are not (or close to) normally distributed.

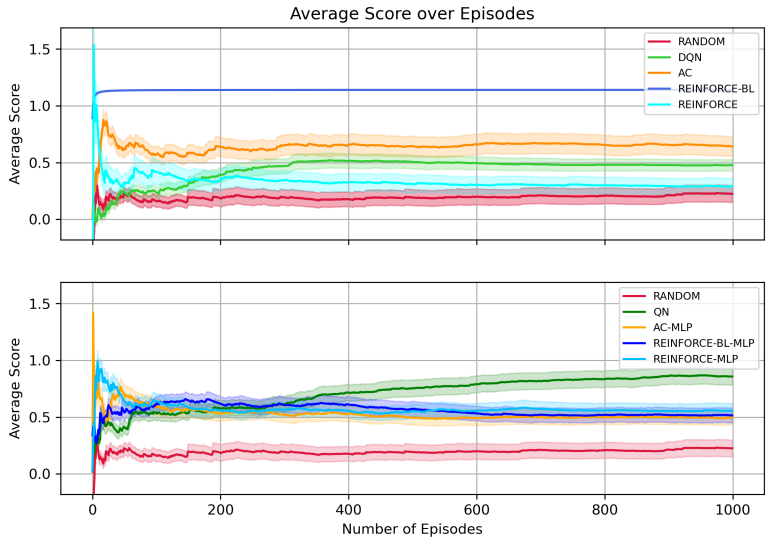

Figure 2: The average score (average undiscounted return) as the number of episodes increases for all agents. The standard deviation around the average scores is scaled by a factor of 0.1.

4 Discussion

According to the author of Slimevolleygym, DQN does not perform well in the Slimevolley environment with an average score of -4.091 ± 1.242 (after 1000 episodes) (Ha, 2020). Meanwhile, our DQN performs better with an average score of 0.447 ± 0.5 (after 1000 episodes) (see Table 2). At the same time, a random policy has an average score of -4.866 ± 0.372, according to the author, while a random policy, according to our results, gets 0.225 ± 0.738.

The results displayed in section 3 are from one training run. The performance of the algorithms in terms of average score can vary to some degree per training session (but still different from random). In some cases (more often with using CNN models), the training can fail, and the agent performs worse than random.

Our best-performing algorithm is REINFORCE with baseline (CNN), where it learned a policy that involves going to the net and continuously jumping there with probability 1. However, it is just as likely to learn a policy that involves standing completely still in the hope that the ball hits the agent's head. This is a general phenomenon with the Monte Carlo agents, where they can quickly converge to a (suboptimal) policy when using a CNN for function approximation. In this case with convergence we mean that $\exists a \in \mathcal{A}\ \forall s \in \mathcal{S}\ \pi(a|s,\theta) = 1$. In other words, after the softmax activation function, a specific action gets probability 1. Another example is that one-step actor-critic using a CNN (AC) performs better than DQN and one-step actor-critic using a MLP (AC-MLP) (in Figure 2). However, there are runs where DQN consistently outperforms AC and AC-MLP, and AC-MLP outperforms AC.

An overall trend that can be seen in Table 2 and Figure 2 is that each algorithm using a MLP can get roughly twice the average score of its counterpart using a CNN. As a comparison, DQN's average score is 0.447 ± 0.5, whereas QN's average score is 0.857 ± 0.741. The main reason for such a difference is that training an agent to play Slimevolley on just frames is harder because game-relevant information like the velocities are not explicitly provided, and the state space is bigger (Ha, 2020).

Assuming a level of significance $\alpha = 0.005$, all the reinforcement learning agents are shown to have a statistically significant higher average score than the random policy (Table 2). However, none of the agents can beat the (pre-trained) baseline policy that is provided with the game. Nevertheless, the baseline policy is very good at the game, and even human players (us) have trouble beating it.

For future research on Slimevolleygym, a better comparison between value-based and policy gradient methods can be made. The implemented policy gradient algorithms have none of the advancements DQN has. An example of a more advanced policy gradient algorithm is proximal policy optimization.

References

Berner, C., Greg, B., Brooke, C., Vicki, C., Przemysław, D., Christy, D., ... et al. (2019, Dec). Dota 2 with large scale deep reinforcement learning. *arXiv.org*. Retrieved from https://arxiv.org/abs/1912.06680

Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., ... Mordatch, I. (2021, 6). Decision Transformer: Reinforcement Learning via Sequence Modeling. *Cornell University - arXiv*. Retrieved from http://arxiv.org/pdf/2106.01345 doi: 10.48550/arxiv.2106.01345

Ha, D. (2015). Neural slime volleyball. *blog.otoro.net*. Retrieved from https://blog.otoro.net/2015/03/28/neural-slime-volleyball/

Ha, D. (2020). *Slime volleyball gym environment.* https://github.com/hardmaru/slimevolleygym. GitHub.

Mnih, V. (2015, 2). *Human-level control through deep reinforcement learning.* Retrieved from https://www.nature.com/articles/nature14236?error=cookies$_n ot_s upportedcode = ae02c795 - 8649 - 4822 - 8b18 - a98e244aa194$

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013, 12). Playing Atari with Deep Reinforcement Learning. *arXiv: Learning*. Retrieved from http://cs.nyu.edu/ koray/publis/mnih-atari-2013.pdf

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, *abs/1707.06347*. Retrieved from http://arxiv.org/abs/1707.06347

Sutton, R., & Barto, A. (2018). *Reinforcement Learning, second edition*. Amsterdam, Nederland: Amsterdam University Press.

A Pseudocode

Algorithm 1 Deep Q-learning with Experience Replay: Semi-gradient off-policy temporal difference algorithm for control (episodic). Adapted from Mnih et al. (2013). Assumes that a state S (image(s)/frame(s)) is preprocessed. The objective function for network parameters θ is the sum of (mean) square error losses:

- $\mathcal{L}(\theta) = \sum_{i=1,...,N}(R_i + \gamma \max_{a \in \mathcal{A}} \hat{q}(S_i', a, \theta^-) - \hat{q}(S_i, A_i, \theta))^2$

Algorithm parameters: step size $a \in (0, 1]$, small $\epsilon > 0$
Initialize agent, involves initializing its replay memory buffer $\mathcal{D}$ to capacity N
Initialize parameter $\theta \in \mathbb{R}^D$ of $\hat{q}(s, a, \theta)$ with random weights
for each episode **do**
    Initialize environment, get initial state S
    for $t = 1, ..., T$ (until terminal state) **do**
        Choose $A \sim \pi(\cdot|S)$ (for example, π is ϵ-greedy)
        Take action A and observe reward R and next state (frames) S' from environment
        Store $\tau = (S, A, R, S')$ in $\mathcal{D}$
        Sample random minibatch of trajectories $\tau_i = (S_i, A_i, R_i, S_i')$ from $\mathcal{D}$
        $\delta_i \leftarrow \begin{cases} R_i & \text{if } S' \text{ is terminal state} \\ R_i + \gamma \max_{a \in \mathcal{A}} \hat{q}(S_i', a, \theta^-) - \hat{q}(S_i, A_i, \theta) & \text{otherwise} \end{cases}$
        $\theta \leftarrow \theta + \alpha\big[\sum_{i=1}^{M} \delta_i \nabla \hat{q}(S_i, A_i, \theta)\big]$ (see equation 10)
        $S \leftarrow S'$

Algorithm 2 One-Step Actor-Critic (episodic), for estimating $\pi_\theta \approx \pi_*$. Adapted from Sutton and Barto (2018). δ is also referred to as the advantage here.

- Objective function critic: $\mathcal{L}_C(\mathbf{w}) = (R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}))^2$
- Objective function actor: $\widehat{J(\theta)} = -\mathcal{L}_A(\theta) = (R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})) \ln \pi(A|S, \theta)$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
Parameters: step size $\alpha^\theta > 0, \alpha^{\mathbf{w}} > 0$
Initialize policy parameters $\theta \in \mathbb{R}^{D'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^D$
for each episode **do**
    Initialize environment, get initial state S
    $I = 1$
    for $t = 1, ..., T$ (until terminal state) **do**
        $A \sim \pi(\cdot|S, \theta)$
        Take action A and observe R and next state S' from environment
        $\delta \leftarrow \begin{cases} R & \text{if } S' \text{ is terminal state} \\ R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}) & \text{otherwise} \end{cases}$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla \hat{v}(S, \mathbf{w})$ (see equation 23)
        $\theta \leftarrow \theta + \alpha^\theta I \delta \nabla \ln \pi(A|S, \theta)$ (see equation 20)
        $I \leftarrow \gamma I$
        $S \leftarrow S'$

Algorithm 3 REINFORCE: Monte-Carlo Policy Gradient Control (episodic) for π_*. Adapted from Sutton and Barto (2018).

- Objective function actor: $\widehat{J(\theta)} = -\mathcal{L}_A(\theta) = G \ln \pi(A|S, \theta)$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$
Parameters: step size $\alpha > 0$
Initialize policy parameters $\theta \in \mathbb{R}^{D'}$
for each episode **do**
    Initialize environment, generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
    for each step of the episode $t = 0, 1, ..., T-1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A|S, \theta)$ (see equation 18)

Algorithm 4 REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$. Adapted from Sutton and Barto (2018)

- Objective function critic: $\mathcal{L}(\mathbf{w}) = (G - \hat{v}(S, \mathbf{w}))^2$
- Objective function actor: $\widehat{J(\theta)} = -\mathcal{L}_A(\theta) = (G - \hat{v}(S_t, \mathbf{w})) \ln \pi(A|S, \theta)$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
Parameters: step size $\alpha^\theta > 0, \alpha^{\mathbf{w}} > 0$
Initialize policy parameters $\theta \in \mathbb{R}^{D'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^D$
for each episode **do**
    Initialize environment, generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
    for each step of the episode $t = 0, 1, ..., T-1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$ (see equation 23)
        $\theta \leftarrow \theta + \alpha^\theta \gamma^t G \nabla \ln \pi(A|S, \theta)$ (see equation 18)

B Derivation REINFORCE

$$
\begin{aligned}
&\nabla J(\theta) \propto \textstyle\sum_s \mu(s) \gamma_t \sum_a q_\pi(s, a) \nabla \pi(a|s, \theta) \\
&= \mathbb{E}_\pi \left[\gamma_t \textstyle\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \theta) \right] \\
&\text{(multiplying and dividing by } \pi(a|S_t, \theta)) \\
&= \mathbb{E}_\pi \left[\gamma_t \textstyle\sum_a \pi(a|S_t, \theta) \sum_a q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \theta)}{\pi(a|S_t, \theta)} \right] \\
&\text{(replacing } a \text{ by the random variable } A_t \sim \pi) \\
&= \mathbb{E}_\pi \left[\gamma_t q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right] \\
&\text{(because } \mathbb{E}_\pi[G_t|S_t, A_t] = q_\pi(S_t, A_t) \text{ and } \mathbb{E}[\mathbb{E}[X]] = \mathbb{E}[X]) \\
&= \mathbb{E}_\pi \left[\gamma_t G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right] \\
&\text{(because } \nabla \ln x = \tfrac{\nabla x}{x}) \\
&= \mathbb{E}_\pi \left[\gamma_t G_t \nabla \ln \pi(A_t|S_t, \theta) \right],
\end{aligned}
\tag{26}
$$

where $G_t = \sum_{k=0}^{T} \gamma^{k-t-1} R_{t+k}$ is the return. The stochastic gradient ascent algorithm is derived by sampling this expectation: $\theta_{t+1} = \theta_t + \alpha \gamma_t G_t \nabla \ln \pi(A_t|S_t, \theta)$.

C Abbereviations used for Reinforcement Learning Algorithms

DQN	Deep Q-network (uses a CNN)
QN	Q-network: DQN algorithm except it uses a MLP with handcrafted features
AC	One-Step Actor-Critic (using CNN)
REINFORCE	REINFORCE (using CNN)
REINFORCE-BL	REINFORCE with Baseline (using CNN)
AC-MLP	One-Step Actor-Critic using a MLP with handcrafted features
REINFORCE-MLP	REINFORCE using a MLP
REINFORCE-BL-MLP	REINFORCE with Baseline using a MLP

Table 3: Table showing the abbreviations for each agent/reinforcement learning algorithm.

D Summary of Notation

In the reinforcement learning setting, the random variables $S : \Omega \to \mathcal{S}$, $R : \Omega \to \mathbb{R}$, $A : \Omega \to \mathcal{A}$ can be viewed as measuring devices that given an elementary event $\omega \in \Omega$ gives the state, reward and action respectively. $\theta_q, \theta_v, \theta_\pi$ are parameters of the specific machine learning models mentioned in section 2.3.1, while $\theta, \mathbf{w}$ are used to refer to arbitrary models.

$P(X = x)$	probability that a random variable X returns the value x
$X \sim P_X$	P_X is the distribution of the random variable X, meaning that $P(X = x) = P_X(\{x\})$
$\mathbb{E}[X]$	expectations of a random variable $X : \Omega \to \mathcal{T}$. In case $\mathcal{T}$ is discrete, then given a probability mass function $p : \mathcal{T} \to [0, 1]$ $\mathbb{E}[X] = \sum_{x \in \mathcal{T}} x \cdot p(x)$ In case $\mathcal{T} \subseteq \mathbb{R}^n$, then given a probability density function $p : \mathbb{R}^n \to \mathbb{R}$ $\mathbb{E}[X] = \int_{\mathcal{T}} p(\mathbf{x}) d\mathbf{x}$
$\nabla f(\mathbf{x})$	gradient $\nabla f : \mathbb{R}^N \to \mathbb{R}^N$ of a function $f : \mathbb{R}^N \to \mathbb{R}$. The gradient (at a specific point $\mathbf{x} \in \mathbb{R}^N$) is a vector of partial derivatives $\nabla f(\mathbf{x}) = (\frac{\partial f}{\partial x_1}(\mathbf{x}), ..., \frac{\partial f}{\partial x_N}(\mathbf{x}))^T$
ϵ	probability of taking a random action in an ϵ-greedy policy
γ	discount factor $\gamma \in (0, 1]$
α	learning rate $\alpha \in (0, 1]$
$\mathcal{M}$	markov decision process $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$
$\mathcal{S}$	set of all states
s, s'	states $s, s' \in \mathcal{S}$
$\mathcal{A}$	set of all possible actions
$\mathcal{A}(s)$	set of all possible actions in state s
$\mathcal{R}$	reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$.
$\mathcal{P}$	transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$
A_t	random variable representing the action at time t
S_t	random variable representing the state at time t
R_t	random variable representing the reward at time t
π	policy $\pi : \mathcal{S} \to \mathcal{A}$ (deterministic case), $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$ (stochastic case)
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under optimal policy
$q_\pi(s, a)$	value of taking action a in state s under policy π
$q_*(s, a)$	value of taking action a in state s under the optimal policy
V, V_t	estimate of state-value function v_π or v_*
Q, Q_t	estimate of action-value function q_π or q_*
δ_t	temporal-difference (TD) error at time t (a random variable)
$\theta, \theta_t, \mathbf{w}$	parameters of a machine learning model. Typically $\theta, \mathbf{w} \in \mathbb{R}^D$
θ^-	in Deep Q-network (DQN), the "frozen" parameters of the target network that are copied from the primary network
θ_q	parameters of the convolutional neural network $\hat{q}(s, a, \theta_q)$ that is used to approximate the value function $q_\pi(s, a)$
θ_v	parameters of the convolutional neural network $\hat{v}(s, \theta_v)$ that is used to approximate the value function $v_\pi(s)$
θ_π	parameters of the convolutional neural network $\pi(a\|s, \theta_\pi)$ that is used to approximate the policy $\pi(a\|s)$
$\hat{v}(s, \theta)$	approximation of the value function by a model with parameters θ
$\hat{q}(s, a, \theta)$	approximation of the action-value function by a model with parameters θ
$\mathcal{D}$	the experience replay buffer.
τ	trajectory $\tau = (S_t, A_t, R_{t+1}, S_{t+1})$

E Loss Graphs

DQN

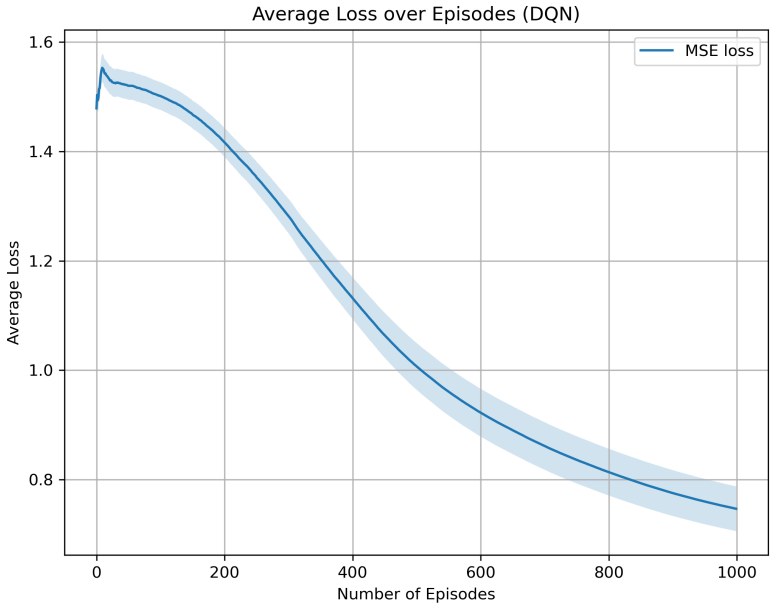

Figure 3: Average loss deep Q-network (variance scaled by 0.1)

AC

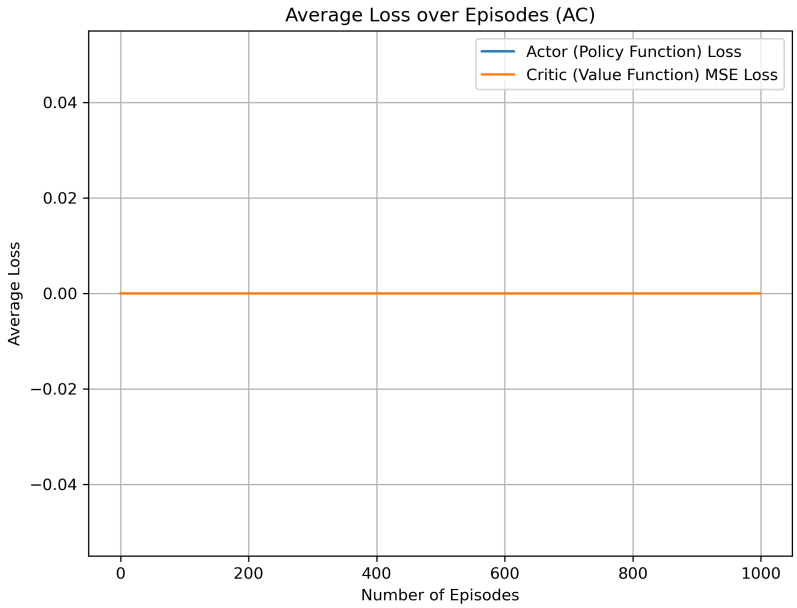

Figure 4: Average loss one-step actor-critic (variance scaled by 0.1)

REINFORCE-BL

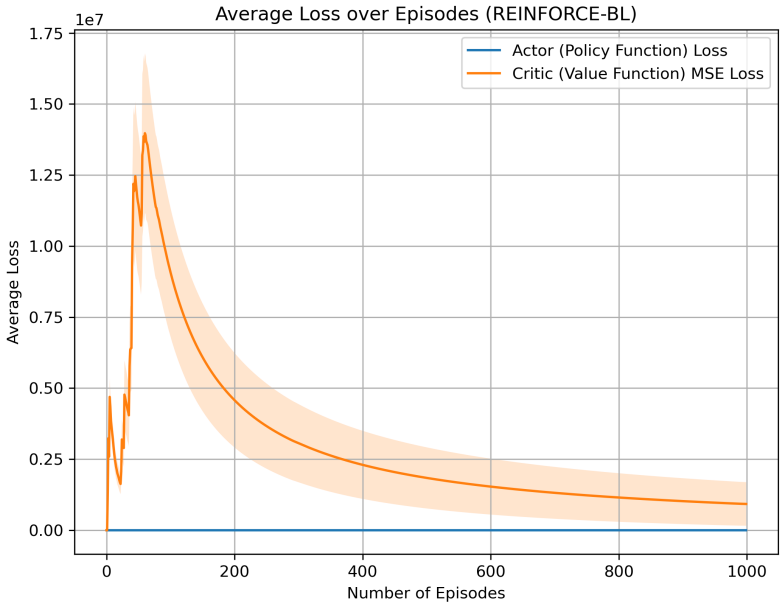

Figure 5: Average loss REINFORCE with baseline (variance scaled by 0.1)

REINFORCE

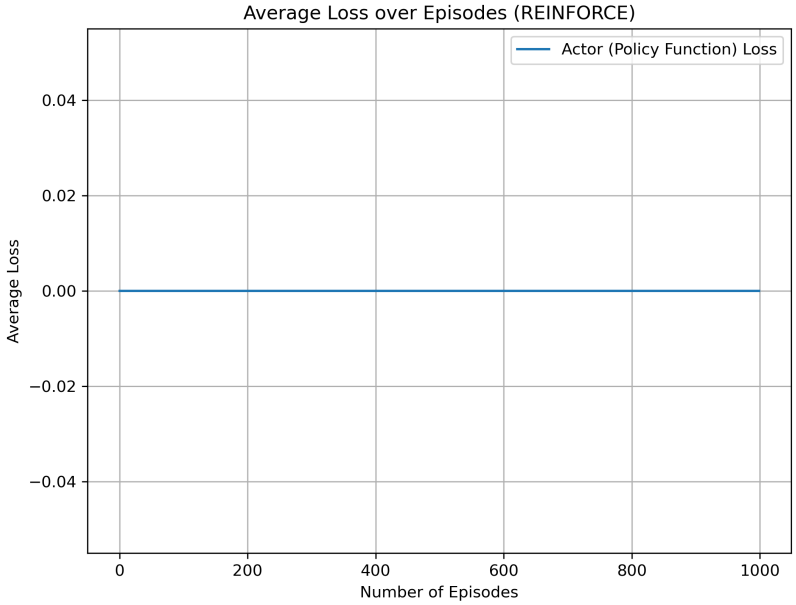

Figure 6: Average loss REINFORCE (variance scaled by 0.1)

QN

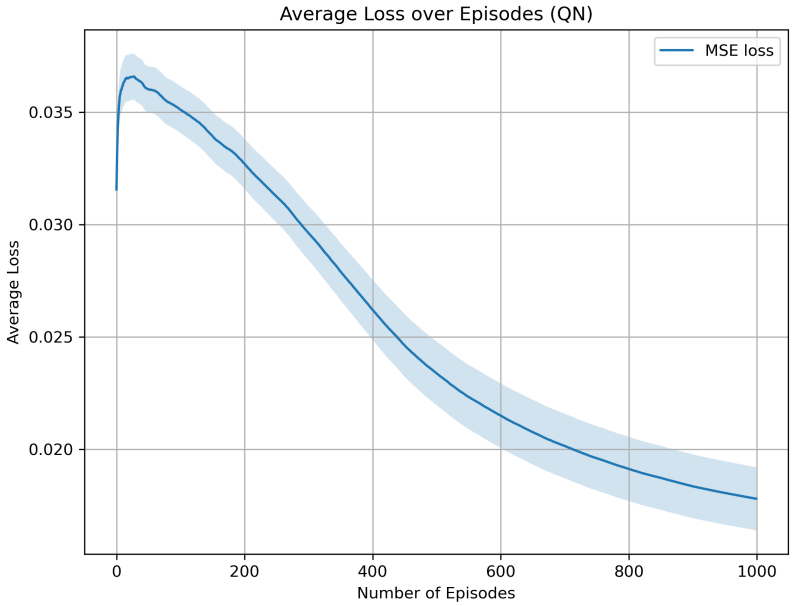

Figure 7: Average loss Q-network (variance scaled by 0.1)

AC-MLP

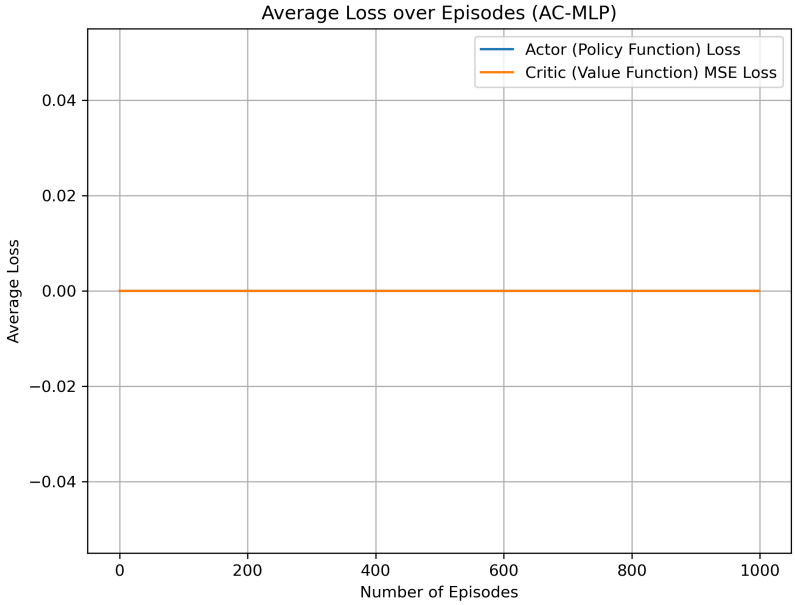

Figure 8: Average loss one-step actor-critic (MLP) (variance scaled by 0.1)

REINFORCE-BL-MLP

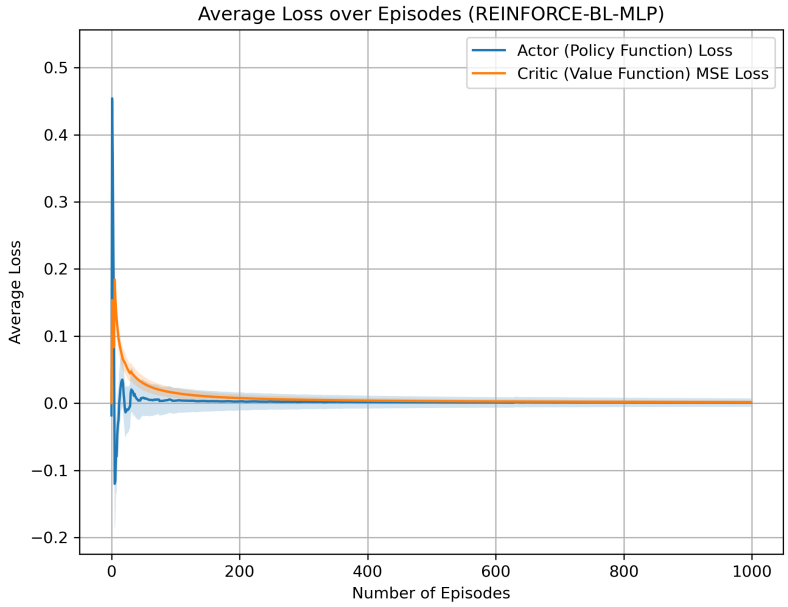

Figure 9: Average loss REINFORCE with baseline (MLP) (variance scaled by 0.1)

REINFORCE-MLP

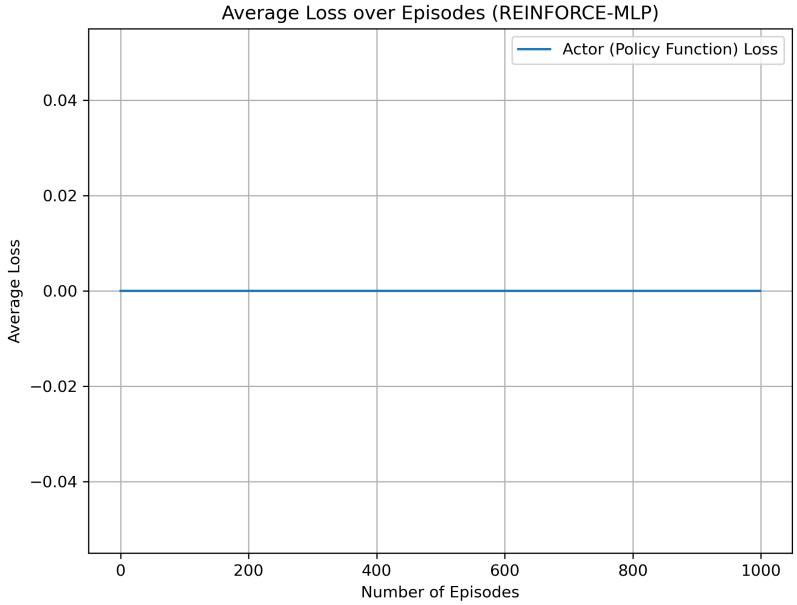

Figure 10: Average loss REINFORCE (MLP) (variance scaled by 0.1)