Random Search-Based Hyperparameter Tuning Approach For Solving Lunar Lander Problem Using Deep Q-Learning

Gökay Gülsoy
Computer Engineering Department
Izmir Institute of Technology
Gülbahçe, İzmir YTE, 35433 Barbaros/Urla/İzmir
E-Mail: gokaygulsoy@iyte.edu.tr

***Abstract*—Reinforcement Learning (RL) is a field of machine learning that focuses on enabling an intelligent agent to navigate through an environment under uncertainty to maximize cumulative long-term reward. In this paper, I implement and analyze a Deep Learning based RL technique known as Deep Q-Learning on OpenAI Gym's *LunarLander-v3* environment. I utilized a random-search approach to tune hyperparameters and be able to achieve average rewards of 200+ with certain hyperparameter configuration combinations. This study also underlines the importance of how well-aligned hyperparameter configurations found by random search can lead to promising results, which may otherwise be computationally infeasible to explore. The 15 most promising hyperparameter configurations and average rewards obtained are then provided for comparative analysis to conclude which configurations lead to better rewards.**

***Index Terms*—Deep Reinforcement Learning, Neural Networks, Q-Learning, RL Agents, OpenAI Gym**

I. Introduction

Over the past several years, reinforcement learning [1] has been proven to be successful on many real-world applications, including robotics and control systems [2, 3]. Various approaches have been proposed and implemented to solve such problems [4, 5]. In this paper, a well-known robotic control problem - the Lunar Lander problem - is solved using random search-based hyperparameter tuning with the Deep Q-Learning algorithm, and then best 15 configurations are compared to find the most promising results. The problem presents a simplified version of landing optimally on the lunar surface, which in itself has been a topic of extensive research [6–8].

II. Problem Definition

This study aims to solve the Lunar Lander environment in the OpenAI gym [9] library using a deep reinforcement learning method. The environment simulates the situation where a lander needs to land at a specific location under low-gravity conditions, and has a well-defined physics engine implemented. The main goal of the simulation is to direct the agent to the landing pad as softly and fuel-efficiently as possible. The state space is continuous as in real physics, but the action space is discrete.

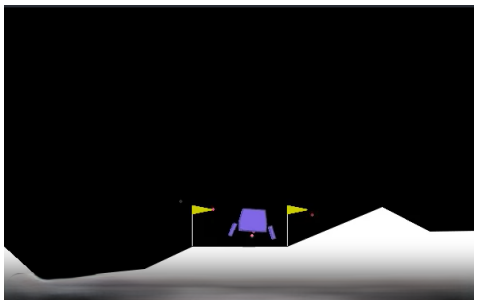

Fig. 1. Lunar Lander Problem Visualization

III. Related Works

Different techniques have been used in solving the lunar lander environment. Study [10] makes use of Evolutionary Algorithms (EAs) and Metaheuristic Algorithms for evolving neural network topologies in simple policy gradient, A2C, and Deep Deterministic Policy Gradient (DDPG) reinforcement learning. Tuning and topology selection are not a straightforward task and require experience as well as domain knowledge. That study applies EAs to optimize the hyperparameters as well as the topology, thus eliminating the need to hand-tune the hyperparameters. [11] uses a control-model-based strategy that learns the optimal control parameters instead of the dynamics of the system and provides a general framework that supports multiple RL methods running in parallel and interactively learning together. Spiking neural networks are explored as a solution to OpenAI virtual environments in [12]. It presents an exploration of possibilities and challenges associated with coupling biologically-inspired neural architectures with real-time simulation environments. The use of an Atari 2600 emulator as a reinforcement platform was introduced by [13], which applied standard reinforcement learning algorithms with linear function approximation and generic visual features. The HyperNEAT evolutionary architecture [14] has also been applied to the Atari platform, where it was used to evolve (separately, for each distinct game) a neural network representing a strategy for that game. The first deep learning model to successfully learn control policies directly from high-dimensional

sensory input using reinforcement learning is presented in [15]. The model used is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. The method is applied to seven Atari 2600 games from the Arcade Learning Environment, and it outperforms all previous approaches on six games, also surpassing a human expert on three of them. It was shown that combining model-free reinforcement algorithms such as Q-learning with non-linear function approximators [16], or indeed off-policy learning, could cause the Q-network to diverge. Thus, the majority of the subsequent work in reinforcement learning focused on linear function approximators with better convergence guarantees [16]. The divergence issues with Q-learning have been partially addressed by *gradient temporal-difference* methods. These methods are proven to converge when evaluating a fixed policy with a nonlinear function approximator [17]. Neural Fitted Q-learning (NFQ) [18] optimizes the sequence of loss functions using the RPROP algorithm to update the parameters of the Q-network. It uses a batch update that has a computational cost per iteration that is proportional to the size of the dataset.

The aim of this study is to solve the lunar lander problem using Deep Q-Learning by finding the most promising hyperparameter configurations through random search, which lead to better cumulative rewards.

IV. Model

A. Framework

The framework used for the lunar lander problem is Gym, a library by OpenAI for developing and comparing reinforcement learning algorithms. It supports various learning environments, ranging from robotics to Atari games. The simulator used is called *Box2D*, and the environment is called *LunarLander-v3*.

B. Observations and State Space

The observation space determines various attributes regarding the lander. Specificially there are 8 state variables associated with the state space, as shown below:

$$state \rightarrow \begin{cases} x, \text{ coordinate of the lander} \\ y, \text{ coordinate of the lander} \\ v_x, \text{ the horizontal velocity} \\ v_y, \text{ the vertical velocity} \\ \theta, \text{ the orientation in space} \\ v_\theta, \text{ the angular velocity} \\ \text{Left leg touching the ground (Boolean)} \\ \text{Right leg touching the ground (Boolean)} \end{cases}$$

All of the coordinate values are relative to the landing pad instead of the lower left corner of the window. The x coordinate of the lander is 0 when the lander is on the line connecting the center of the landing pad to the top of the screen. Thus, it is positive on the right side of the screen and negative on the left side. The y coordinate is positive at the level above the landing pad and negative at the level below.

C. Action Space

There are four discrete actions available: do nothing, fire left orientation engine, fire right orientation engine, and fire main engine. Firing left and right engines introduces a torque on the lander, which causes rotation and makes stabilizing difficult.

D. Reward

Defining a proper reward directly affects the performance of the agent. The agent needs to maintain both a good posture mid-air and reach the landing pad as quickly as possible. Specifically in this study, the reward is defined to be:

- is increased/decreased the closer/further the lander is to the landing pad.
- is increased/decreased, the slower/faster the lander is moving
- is decreased the more the lander is tilted (angle not horizontal)
- is increased by 10 points for each leg that is in contact with the ground
- is decreased by 0.03 points each frame, a side engine is firing
- is decreased by 0.3 points each frame, the main engine is firing
- The episode receives an additional -100 or +100 points for crashing or landing safely, respectively

According to Gymnasium documentation, an episode is considered a solution if it scores at least 200 points. Thus, configurations that reach 200 or above are considered as a solution in this study.

E. Deep Q-Learning

Since the state space is continuous and growing exponentially in the problem, dynamic programming approaches such as value iteration or policy iteration [19], and multi-armed bandit [20, 21] approaches are not sufficient to cover such a huge space. For this problem modified version of Q-learning, called Deep Q-learning (DQN), is used [15, 22] to account for continuous space. The DQN method makes use of a multilayer perceptron, called the Deep Q-Network (DQN), to estimate the Q values. The input to the network is the current state (8-dimensional in our case), and outputs are the Q values for all state-action pairs for that state. The Q-Learning update rule is as follows:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s,a)) \quad (1)$$

The optimal Q-value $Q^*(s,a)$ is estimated using the neural network with parameters θ. This becomes a regression task. In that case, the loss function at iteration i is acquired by the temporal difference error:

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{(s,a)\sim\rho_0}\left[(y_i - Q(s,a;\theta_i))^2\right] \quad (2)$$

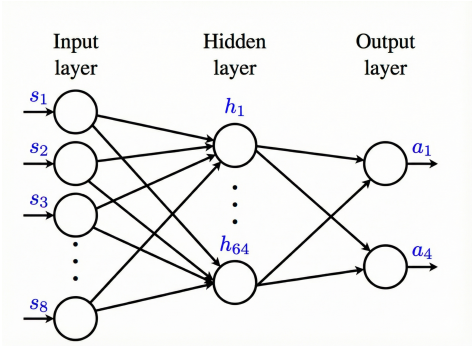

Fig. 2. Neural Network Used for Deep Q-Learning

where

$$y_i = \mathbb{E}_{s' \sim \varepsilon}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})\right] \tag{3}$$

In the above equation, θ_{i-1} are the network parameters from the previous iteration, $\rho(s, a)$ is a probability distribution over states s and actions a, and ε is the environment that the agent interacts with. Gradient descent is then used to optimise the loss function and update the neural network weights. The neural network parameters, which come from the previous iteration θ_{i-1}, are kept fixed while optimizing $\mathcal{L}_i(\theta_i)$. Since the next action is selected based on the greedy policy, Q-Learning is an off-policy RL algorithm [23].

One of the intricacies here is that the successive samples are highly correlated since the next state depends on the current state and action. This is not the case in traditional supervised learning, where successive samples are independent and identically distributed. To tackle this problem, the transitions encountered by the agent are stored in a replay memory $\mathcal{D}$. Random minibatches of transitions $\{s, a, r, s'\}$ are sampled from $\mathcal{D}$ during training to train the network. This technique is called *experience replay* [24].

By using experience replay, the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. When learning by experience replay, it is necessary to learn-off policy (because our current parameters are different from those used to generate the sample), which also motivates the choice of Deep Q-learning. In practice, our algorithm only stores the last N experience tuples in the replay memory, and samples uniformly at random from $\mathcal{D}$ every 4 steps when performing updates. This approach is limited to some extent since the memory buffer does not differentiate important transitions and always overwrites with the most recent transitions because of the finite memory size N. Likewise, the uniform sampling gives equal significance to all transitions in the replay memory. A more sophisticated sampling strategy may emphasize transitions from which model can learn the most, as applied by the study [25], which uses an algorithm called *prioritized sweeping*. Algorithm 1 gives the pseudocode of Deep Q-learning with experience replay algorithm used in this study.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and
    x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$
    preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    $y_j = \begin{cases} r_j \text{ for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) \text{ for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 2
  end for
end for

In this study, a 3-layer neural network with 64 neurons in the hidden layers is used by default. *ReLU* activation function for the hidden layers and *LINEAR* activation for the output is used. Representation of neural network architecture is shown in Fig. 2. The default value of the learning rate used is 0.0001, and the minibatch size is 64. Different configurations have their own hyperparameter combinations as shown in subsection V-C.

Exploration Policy: An improved ε-greedy policy is used to select the action, with ϵ starting at 1 to favor exploration and decaying by 0.995 for every iteration, until it reaches 0.01, after it stays the same, this approach ensures that exploration is maintained with a small percentage so that agent does not get stuck in a suboptimal loop.

V. Methodology

A. The Original Problem

The goal of the problem is to direct the lunar lander to the landing pad between two flag poles as softly and fuel-efficiently as possible. Both legs of the lander should be in contact with the pad while landing. The lander should reach the landing pad as quickly as possible, while maintaining a stable posture and minimum angular velocity. Furthermore, once landed, the lander should not take off again.

B. Technologies Used and Project Structure

Implementation of random search-based hyperparameter tuning simulation for solving the $Lunarlander-v3$ environment with Deep Q-Learning is done in Python programming language using OpenAI gym-based library called gymnasium [26], numpy, matplotlib, and Pytorch [27] libraries. The project consists of 6 Python source code files, which are

agent.py, *dqn.py*, *graph.py*, *inference.py*, *main.py*, and *tune.py* respectively. The *agent.py* file implements the action selection mechanism with $\varepsilon - greedy$ strategy and uses the experience replay mechanism for sampling minibatches of transitions $\{s, a, r, s'\}$. Also maintains policy and target networks for off-policy learning. The *dqn.py* implements a 3-layer DQN neural network using the Pytorch library. The *graph.py* plots the reward and loss graphs for each hyperparameter-tuning trial and saves them under the *solved_model_plots* directory.

The *inference.py* uses saved models that reached +240 reward score in tuning trials to perform inference by all the time exploiting ($\varepsilon = 0.0$) and generates a simulation video of the lunar lander landing under the *videos* folder. The *main.py* file implements the training procedure for DQN by creating an agent instance to explore the environment by taking steps and accumulating rewards and losses. It also creates .npy files for scores and losses in each tuning trial and saves them under the *scores* directory. The *tune.py* file implements the random search-based hyperparameter tuning using learning rate(lr), discount factor(gamma γ), batch size($batch\_size$), and number of neurons in the hidden layer($hidden\_size$) as hyperparameters. It runs 20 trials by default, which can be configured as well. Implementation is provided in the GitHub[1] with usage instructions.

C. Experiments

20 Experiments are carried out in total, in which each experiment comprises 20 runs, so in total, 400 configurations are obtained. Among those 400 configurations most promising 15 configurations are selected, and each configuration is averaged over 5 runs of the same configuration. Random search-based hyperparameter tuning is applied on a discrete subset of values for *learning rate*, *gamma*, *batch size*, and *number of neurons in the hidden layer* as follows:

- learning rate (lr) $\rightarrow \{0.005, 0.001, 0.0005, 0.0001\}$
- gamma (gamma) $\rightarrow \{0.99, 0.95, 0.85, 0.80\}$
- batch size (batch_size) $\rightarrow \{32, 64, 128\}$
- hidden size (hidden_size) $\rightarrow \{32, 64, 128\}$

The most promising fifteen configurations found by random search-based hyperparameter tuning are as follows:

Configuration-1
- $lr \rightarrow 0.0001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 64$
- $64 \rightarrow 128$

Configuration-2
- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

Configuration-3
- $lr \rightarrow 0.0005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

Configuration-4
- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 64$
- $64 \rightarrow 32$

Configuration-5
- $lr \rightarrow 0.0005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 64$
- $64 \rightarrow 64$

Configuration-6
- $lr \rightarrow 0.005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 64$
- $64 \rightarrow 64$

Configuration-7
- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 64$
- $64 \rightarrow 64$

Configuration-8
- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 32$
- $64 \rightarrow 32$

Configuration-9
- $lr \rightarrow 0.005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 64$
- $64 \rightarrow 32$

Configuration-10
- $lr \rightarrow 0.0005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 64$
- $64 \rightarrow 32$

Configuration-11
- $lr \rightarrow 0.0005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 64$

Configuration-12
- $lr \rightarrow 0.0001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

[1]link_to_project_implementation

Configuration-13

- $lr \rightarrow 0.005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 32$
- $64 \rightarrow 64$

Configuration-14

- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 32$
- $64 \rightarrow 128$

Configuration-15

- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 64$

Rewards that exceed 200 are accepted as solutions, as stated in the OpenAI gym environment for the lunar lander problem, but only configurations that are $+240$ are given as best results. Best results for the rewards are obtained along with their reward and loss graphs with respect to episodes, which are given as follows:

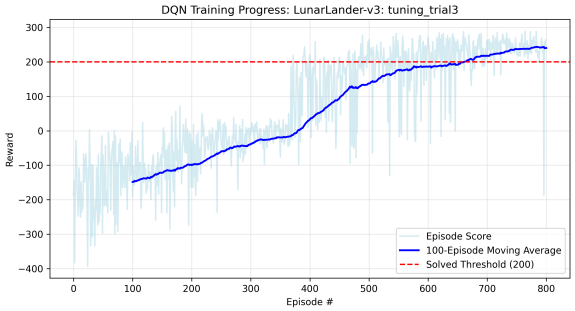

Fig. 3. Reward Graph of Experiment 1 Tuning Trial 3

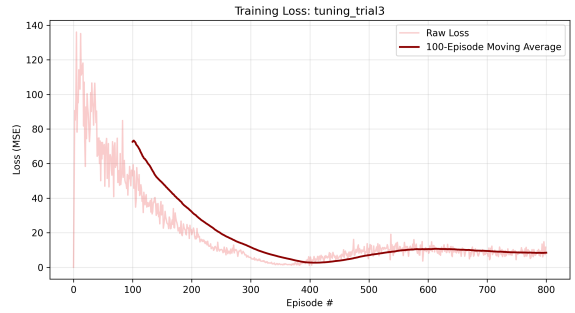

Fig. 4. Loss Graph of Experiment 1 Tuning Trial 3

Experiment 1 Tuning trial 3 $Reward$: 240.33

Configuration-3

- $lr \rightarrow 0.0005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

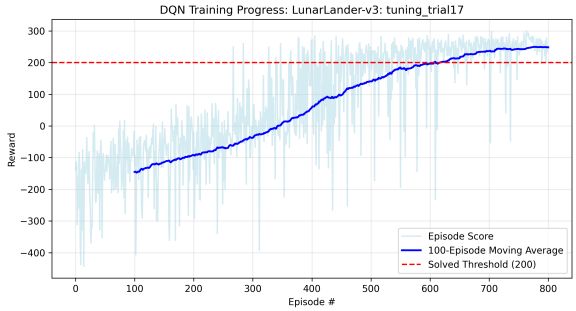

Fig. 5. Reward Graph of Experiment 5 Tuning Trial 17

Fig. 6. Loss Graph of Experiment 5 Tuning Trial 17

Experiment 5 Tuning trial 17 $Reward$: 248.17

Configuration-2

- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

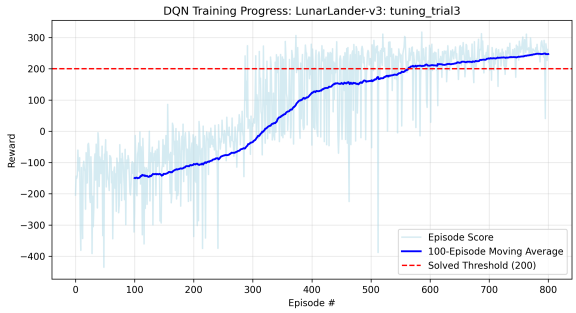

Fig. 7. Reward Graph of Experiment 10 Tuning Trial 8

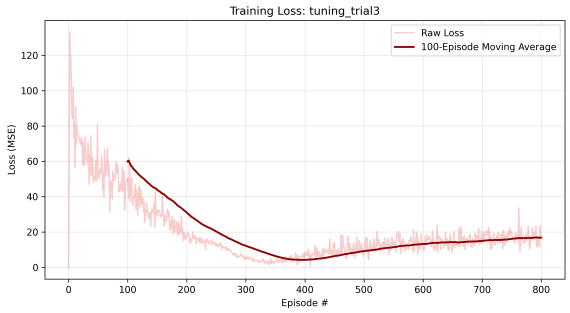

Fig. 8. Loss Graph of Experiment 10 Tuning Trial 8

Experiment 10 Tuning trial 8 $Reward$: 247.11

Configuration-2

- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

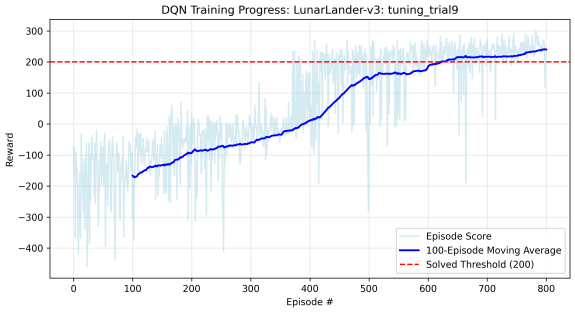

Fig. 9. Reward Graph of Experiment 11 Tuning Trial 9

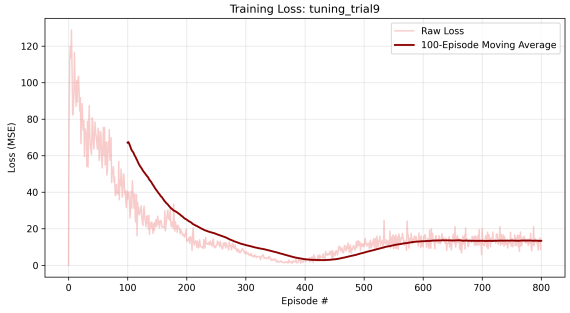

Fig. 10. Loss Graph of Experiment 11 Tuning Trial 9

Experiment 11 Tuning trial 9 $Reward$: 240.03

Configuration-3

- $lr \rightarrow 0.0005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

Experiment 13 Tuning trial 10 $Reward$: 258.05

Configuration-15

- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 64$

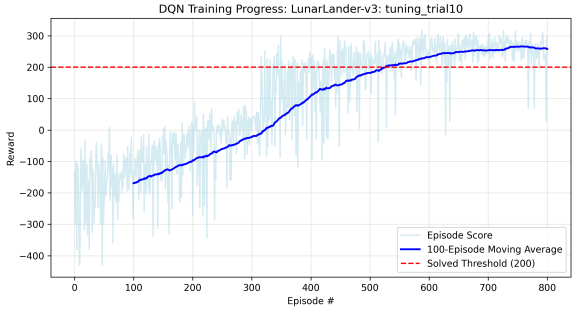

Fig. 11. Reward Graph of Experiment 13 Tuning Trial 10

Fig. 12. Loss Graph of Experiment 13 Tuning Trial 10

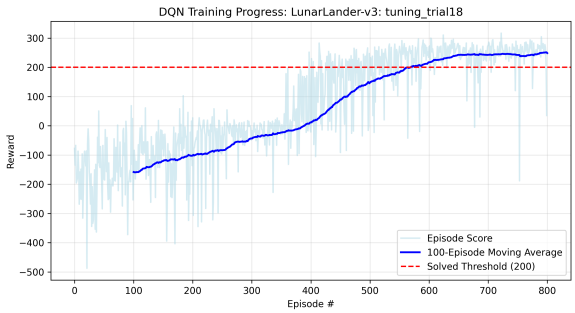

Fig. 13. Reward Graph of Experiment 13 Tuning Trial 18

Experiment 13 Tuning trial 18 $Reward$: 248.78

Configuration-3

- $lr \rightarrow 0.005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

Fig. 14. Loss Graph of Experiment 13 Tuning Trial 18

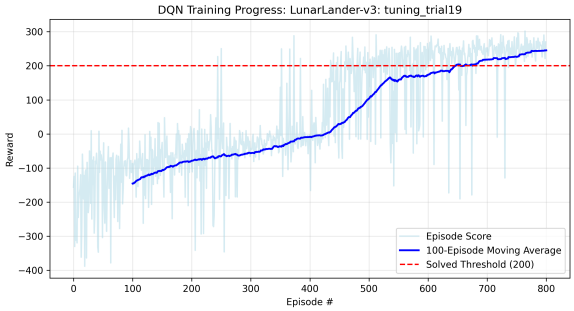

Fig. 15. Reward Graph of Experiment 13 Tuning Trial 19

Fig. 16. Loss Graph of Experiment 13 Tuning Trial 19

Experiment 13 Tuning trial 19 $Reward$: 245.29

Configuration-3

- $lr \rightarrow 0.005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 128$

Experiment 15 Tuning trial 4 $Reward$: 253.81

Configuration-14

- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 32$
- $64 \rightarrow 128$

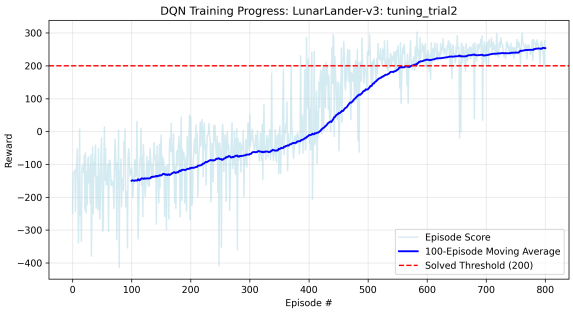

Fig. 17. Reward Graph of Experiment 15 Tuning Trial 4

Fig. 18. Loss Graph of Experiment 15 Tuning Trial 4

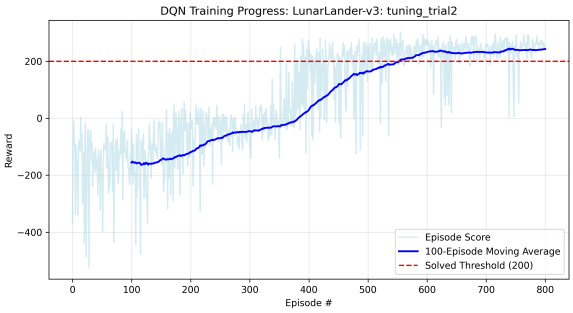

Fig. 19. Reward Graph of Experiment 15 Tuning Trial 8

Fig. 20. Loss Graph of Experiment 15 Tuning Trial 8

Experiment 15 Tuning trial 8 $Reward$: 243.01

Configuration-15

- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $64 \rightarrow 64$

Experimental results show that out of all $+240$ reward results, **4** of them come from **Configuration-3**, **2** of them come from **Configuration-2**, and **2** of them comes from **Configuration-15**, **1** of them comes from **Configuration-14**. So the majority of the best results ($\{240.33, 240.03, 248.78, 245.29\}$) are obtained from the **Configruation-3** which is as follows:

Configuration-3

- $lr \rightarrow 0.0005$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $hidden\_size \rightarrow 128$

The highest reward achieved is 258.05, which is obtained from **Configuration-15**. Configuration-15 is as follows:

Configuration-15

- $lr \rightarrow 0.001$
- $gamma \rightarrow 0.99$
- $batch\_size \rightarrow 128$
- $hidden\_size \rightarrow 64$

Results of the best 15 configurations are averaged over 5 runs and shown in Table I. All the runs that exceeded the reward score of 240 are given in bold. As a result of experiments, 4 highest reward scores are obtained from **Configuration-3**, which suggests that hyperparameters found by random search-based tuning for Configuration-3 are well-aligned with the *LunarLander-v3* problem and environment. Highest rewards obtained from the best 15 configurations that exceed 240 are given in descending order as follows:

- Configuration-15 $\rightarrow 258.05$
- Configuration-14 $\rightarrow 253.81$
- Configuration-3 $\rightarrow 248.78$
- Configuration-2 $\rightarrow 248.17$
- Configuration-2 $\rightarrow 247.11$
- Configuration-3 $\rightarrow 245.29$
- Configuration-15 $\rightarrow 243.01$
- Configuration-3 $\rightarrow 240.33$
- Configuration-3 $\rightarrow 240.03$

TABLE I
AVERAGE REWARDS OF BEST 15 CONFIGURATIONS

Conf.	Run 1	Run 2	Run 3	Run 4	Run 5	Average
1	170.94	194.58	66.36	-7.11	209.34	126.88
2	238.02	**248.17**	220.45	**247.11**	211.51	233.05
3	**240.33**	**240.03**	**248.78**	**245.29**	217.87	238.46
4	214.04	104.87	125.86	182.98	183.29	162.21
5	199.05	176.01	159.53	100.88	231.95	173.48
6	153.95	201.83	183.50	145.98	111.53	159.36
7	225.10	207.37	139.08	208.83	202.14	196.50
8	136.16	181.77	181.94	224.26	104.52	165.73
9	71.56	-3.17	87.36	196.70	132.42	96.97
10	152.41	177.51	152.11	60.43	-13.981	105.70
11	235.52	179.74	208.25	204.22	220.69	209.68
12	178.78	201.22	213.51	213.71	170.06	195.456
13	193.86	176.10	192.11	126.62	200.87	177.91
14	207.85	238.25	**253.81**	218.77	214.34	226.65
15	236.13	200.59	201.32	**243.01**	**258.05**	227.821

VI. CONCLUSION

In this study, random search-based hyperparameter-tuning is applied for solving *LunarLander-v3* environment from the OpenAI gym library. Experimental results shown that the underlying hyperparameters have a significant impact on the obtained reward scores, and the highest rewards were obtained with the **Configuration-3** ($lr \rightarrow 0.0005$, $gamma \rightarrow 0.99$, $batch\_size \rightarrow 128$, $hidden\_size \rightarrow 128$). Out of 5 runs, 4 of them reached a $+240$ score, which are $\{240.33, 240.03, 248.78, 245.29\}$. Another critical observation from the experiments is that all 15 best configurations have the discount factor (gamma γ) hyperparameter as $\gamma = 0.99$. This is because LunarLander is a "delayed reward" problem in which the most significant reward is only granted at the very end of the episode when the lander touches the ground safely.

Gamma(γ) controls how far into the future an agent cares about. A high value of gamma, such as 0.99, fundamentally avoids "greedy hovering", because if gamma(γ) is low such as $\gamma = 0.80$, then agent prioritizes maximizing immediate step-by-step rewards which often results in local optimum where the agent learns to simply hover or let gravity take over (to save the fuel) rather than actively burning fuel to secure the final landing. It fears the immediate cost of fuel more than it values the distant landing reward. In contrast, a high value of gamma, like $\gamma = 0.99$, gives an agent a wider "horizon of sight" so that the agent at the top of the screen still sees the landing reward. Overall, random search-based hyperparameter-tuning can reveal promising combinations of hyperparameter configurations that can lead to high reward scores, which may otherwise be computationally infeasible and never be explored.

REFERENCES

[1] R. S. Sutton, A. G. Barto *et al.*, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[2] J. Kober, J. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, 09 2013.

[3] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Auton. Robots*, vol. 27, no. 1, p. 55–73, Jul. 2009. [Online]. Available: https://doi.org/10.1007/s10514-009-9120-4

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," 1996. [Online]. Available: https://arxiv.org/abs/cs/9605103

[5] C. Szepesvári, *Algorithms for Reinforcement Learning*, 01 2010, vol. 4.

[6] T. Brady and S. Paschall, "The challenge of safe lunar landing," 04 2010, pp. 1 – 14.

[7] D.-H. Cho, B. Jeong, D. Lee, and H. Bang, "Optimal perilune altitude of lunar landing trajectory," *International Journal of Aeronautical and Space Sciences*, vol. 10, no. 1, pp. 67–74, 2009.

[8] X.-L. Liu, G.-R. Duan, and K.-L. Teo, "Optimal soft landing control for moon lander," *Automatica*, vol. 44, no. 4, p. 1097–1103, Apr. 2008. [Online]. Available: https://doi.org/10.1016/j.automatica.2007.08.021

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[10] A. Banerjee, D. Ghosh, and S. Das, "Evolving network topology in policy gradient reinforcement learning algorithms," in *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*. IEEE, 2019, pp. 1–5.

[11] Y. Lu, M. S. Squillante, and C. W. Wu, "A control-model-based approach for reinforcement learning," *arXiv preprint arXiv:1905.12009*, 2019.

[12] C. Peters, T. C. Stewart, R. L. West, and B. Esfandiari, "Dynamic action selection in openai using spiking neural networks." in *FLAIRS*, 2019, pp. 318–323.

[13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of artificial intelligence research*, vol. 47, pp. 253–279, 2013.

[14] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 355–366, 2014.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. [Online]. Available: https://arxiv.org/abs/1312.5602

[16] J. Tsitsiklis and B. Van Roy, "Analysis of temporal-diffference learning with function approximation," *Advances in neural information processing systems*, vol. 9, 1996.

[17] H. Maei, C. Szepesvari, S. Bhatnagar, D. Precup, D. Silver, and R. S. Sutton, "Convergent temporal-difference learning with arbitrary smooth function approximation," *Advances in neural information processing systems*, vol. 22, 2009.

[18] S. Lange and M. Riedmiller, "Deep auto-encoder neural networks in reinforcement learning," in *The 2010 international joint conference on neural networks (IJCNN)*. IEEE, 2010, pp. 1–8.

[19] A. Alla, M. Falcone, and D. Kalise, "An efficient policy iteration algorithm for dynamic programming equations," 2014. [Online]. Available: https://arxiv.org/abs/1308.2087

[20] S. A. McClendon, V. Venkatesh, and J. Morinelli, "Reinforcement learning for machine learning model deployment: Evaluating multi-armed bandits in ml ops environments," 2025. [Online]. Available: https://arxiv.org/abs/2503.22595

[21] H. Combrink, V. Marivate, and B. Rosman, "Reinforcement learning in education: A multi-armed bandit approach," 2022. [Online]. Available: https://arxiv.org/abs/2211.00779

[22] G. Liu, W. Deng, X. Xie, L. Huang, and H. Tang, "Human-level control through directly trained deep spiking q-networks," *IEEE transactions on cybernetics*, vol. 53, no. 11, pp. 7187–7198, 2022.

[23] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[24] L.-J. Lin, *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[25] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time," *Machine learning*, vol. 13, no. 1, pp. 103–130, 1993.

[26] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG *et al.*, "Gymnasium: A standard interface for reinforcement learning environments," *arXiv preprint arXiv:2407.17032*, 2024.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019. [Online]. Available: https://arxiv.org/abs/1912.01703