Learning to Play Games with Deep Reinforcement Learning

by

Son Tung Do

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science

FORDHAM UNIVERSITY

May 2023

ABSTRACT

In this work, Deep Reinforce Learning is used to train agents to play video games. The method chosen for the experiment is Policy Gradient using the REINFORCE algorithm. This algorithm tries to maximize the expected rewards an agent can receive by updating its parameters with gradient-based optimization. REINFORCE is then used to train learning agents in 3 simple video games: Cart Pole, Lunar Lander, and Pixel Copter.

ACKNOWLEDGMENTS

I would like to thank Dr. Juntao Chen, for his role as my research advisor, and Dr. Karina Hogan, for her help in the Thesis Workshop. Special thanks to all the members of the Honors class of 2023 for reading and providing valuable feedback that makes my explanation clearer. Many thanks to my partner Phuong Le and all my friends and family, who have always supported me during my undergraduate journey and in the creation of this thesis. Lastly, I would like to acknowledge Grant Sanderson, a math educator and creator of 3Blue1Brown, and DeepMind, with their breakthrough AlphaGo, for being my inspiration to pursue the study of Artificial Intelligence.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

What are the similarities between Chess and Matrix multiplication? The former is a famous board game, while the latter is a mathematical operation that college students must learn if they want to pass their Linear Algebra class. Both are products of ingenious human intelligence, but humans are no longer the best at them. At the time of this writing, Artificial Intelligence (AI) has learned to explore and perform both tasks efficiently at a superhuman level through the use of Deep Reinforcement Learning [9] [13].

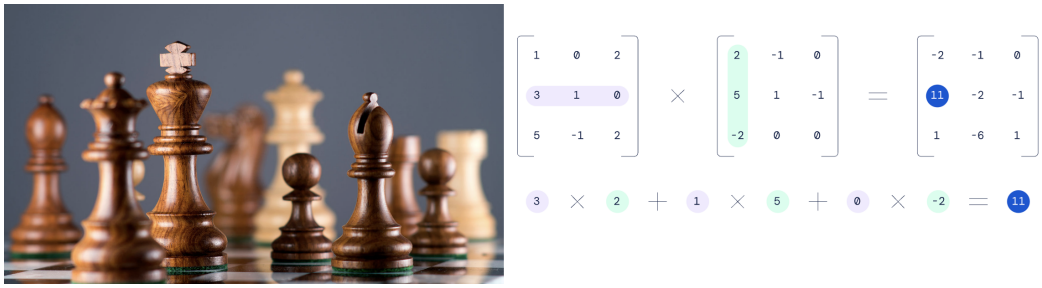

Figure 1.1: Two seemingly unrelated tasks, solved by Deep Reinforcement Learning [9]

Deep Learning and Deep Reinforcement Learning have been the driving forces behind major advances in AI over the last decade. The latest neural networks power the recommendation systems of huge social media platforms, drive cars autonomously, and beat humans in multiple board games and video games. For this thesis, I will apply Deep Reinforcement Learning to train agents that can play video games.

1.1 Related works

For a long time, games have been a topic of interest for AI researchers. Deep Reinforcement Learning achieved widespread recognition after DeepMind's breakthrough with playing Atari games at a superhuman level [11]. DeepMind later combined Deep Reinforcement Learning with Monte Carlo Tree Search to create their breakthrough AlphaGo, which reached superhuman level performance in Go, one of the most complex board games in the world [12]. In 2019, AI trained using Deep Reinforcement Learning methods achieved Grandmaster level in Starcraft II and defeated the world champion in Dota 2, 2 famous esports strategy games [16, 4].

1.2 Organization

The next section will provide the background knowledge needed to understand the content of this thesis. The explanation is guided toward a general audience. No prior knowledge is required, but knowing calculus will make it easier to understand the math behind the algorithms. The background section will be followed by the problem formulation, approach, experiments, and results.

Chapter 2

BACKGROUND

Artificial Intelligence (AI) is a broad term used to describe any artificial system exhibiting intelligent behaviors. Two main approaches to creating such a system are to explicitly tell a machine how to behave or to make the machine learn that behavior by itself. The former approach gave us the IBM chess computer Deep Blue, which beat the chess champion Garry Kasparov in 1997. The latter approach is known as Machine Learning (ML), which enabled DeepMind's computer AlphaGo to beat the Go, the ancient Chinese board game, champion Lee Sedol in 2016 [12]. ML systems are programmed with learning algorithms, and they use these algorithms to learn from a large amount of data. This mechanism allows them to learn tasks with high complexity, those we have no efficient algorithms to solve.

Inside the world of ML, we have three main types of learning paradigms. The first one is supervised learning, where ML systems learn a task from data with labels. An example of this data type is pictures of cats and dogs with labels to denote whether a picture contains a cat or a dog. The second type of ML is unsupervised learning. Unsupervised learning ML model can learn from any type of raw data without labels. The final type of ML is called Reinforcement Learning (RL), commonly acknowledged as the closest form of ML to the human learning process.

2.1 Reinforcement Learning

This is the introduction to RL in the words of Richard Sutton, the father of modern RL, in his book *Reinforcement Learning, An Introduction* [14]:

> "Reinforcement learning is like many topics with names ending in -ing, such as machine learning, planning, and mountaineering, in that it is simultaneously a problem, a class of solution methods that work well on the class of problems, and the field that studies these problems and their solution methods."

The class of RL problems generally concerns a vast solution space, and the learning process involves exploring and analyzing this space to reach an optimal solution. An RL model consists of an agent and the environment, in which the agent will attempt to learn some behaviors by interacting with the environment. The agent can perform actions (according to some policies) on the environment, and the environment will return some information to the agent. The crucial data returned are the current state of the agent and a reward. Data of current states can contain a lot of attributes depending on the task we try to achieve, and in the most common case, it contains the position and velocity of the agent. The agent's behavior is modeled by a policy, which tells the agent what actions to take, given its current state. The reward is what motivates the agent: its ultimate goal is to maximize the amount of reward it receives. This reward is given according to a reward function that we can specify. With the information given by the environment, the agent learns to improve its policy and take better actions, so it can get the most amount of reward possible. This is similar to the reward mechanism that drives human behaviors. Humans are also motivated by rewards. That reward could be anything that makes us feel happy, be it food, money, or simply watching movies and listening to music.

Looking back, the goal of the system designers and the agents seems to misalign with each other. We want the agent to learn intelligent behavior (driving a car safely, playing a game and reaching high scores, or solving a math problem), while the agents just want to maximize a virtual reward at all costs. How can we align our goal with the agent's goal? This is one problem we must tackle when engineering an RL system. Designing a reward function that aligns the goal of the designer with that of the agent could be a challenging

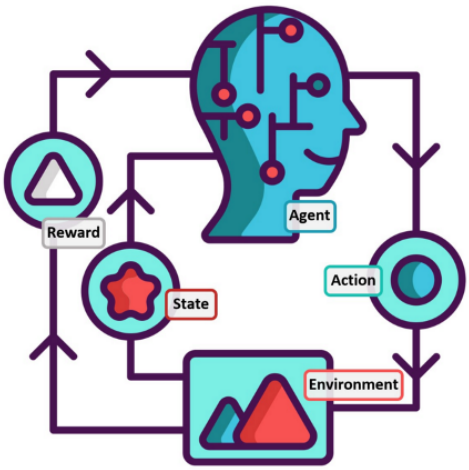

Figure 2.1: Simple diagram of an RL system

task. Failure to do this may result in the agent exploiting a system loophole to gain as much reward as possible without actually learning to do the task, a behavior known as "reward hacking." Let's say we are training an agent to learn to play tennis. To win, the agent needs to hit the ball as unpredictably as possible, so the opponent will have less time to react to its attack and lose. Now, what happens if we reward the agent for every time it hits the ball? This can be a good idea, as the agent will try to catch every attack from the opponent to hit back and get the reward. However, the agent can also exploit this reward mechanism by prolonging the match and playing to not lose. This way, the agent will attack in predictable ways, which will let the opponent have the chance to counterattack and beat the agent. The agent does not care whether it loses as long as it keeps hitting the ball in a long tennis match. It feels happy with the reward, while we, the designer, have an inefficient tennis AI that just plays to prolong a match and not to win it.

Another concept vital to RL is the value function. The reward is what the environment gives the agent at the moment, while the value is the agent's expectation of future reward. In other words, "the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state" (pg. 8). To maximize the reward it will receive, the agent will prioritize a behavior that brings it to a state that it believes to have the

highest value. There are many algorithms to learn such behaviors, many of which involve optimizing highly complex functions. In recent years, one type of optimization model has risen to prominence due to the exponential advances in computer hardware. It is the deep neural network, and the field that studies deep neural networks is known as Deep Learning.

2.2 Deep Learning

2.2.1 Neural Network

Deep Learning is a subfield of ML that involves the use of artificial neural networks for the learning process. An artificial neural network is a complex mathematical model inspired by how the biological brain works. The basis of neural networks has been explored throughout the 20th century but did not become popular due to the computational requirement of such models. Deep Learning then experienced a boom in popularity in 2012 after the success of AlexNet [10], a neural network that won the 2012 ImageNet Large Scale Visual Recognition Challenge. This impressive result showed how well neural network models could scale up and make use of the immense computing power we have in recent years, and neural network has become the hottest topic in ML ever since.

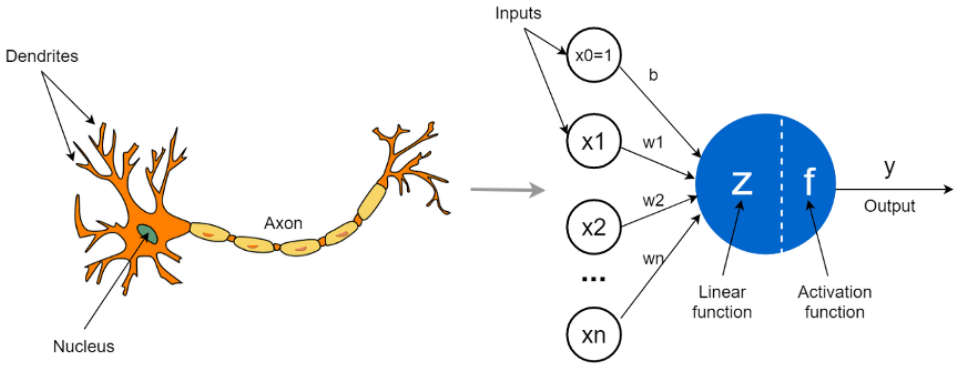

Figure 2.2: A biological neuron in the brain and an artificial neuron in a neural network

The most basic unit of a brain or a neural network is called a neuron. A biological neu-

ron receives multiple electrical signals from other neurons through dendrites, and it will fire an action potential if it receives a certain amount of signals. An artificial neuron is a computation unit containing a number that acts as its output. Similar to the biological neuron, the artificial neuron is connected to many other neurons through weighted connections. It can receive the outputs of other neurons, and how much those outputs affect the neuron will depend on the weights. Therefore, we take the weighted sum of all inputs from previous neurons, and we put that result through an activation function to achieve the output of the current neuron.

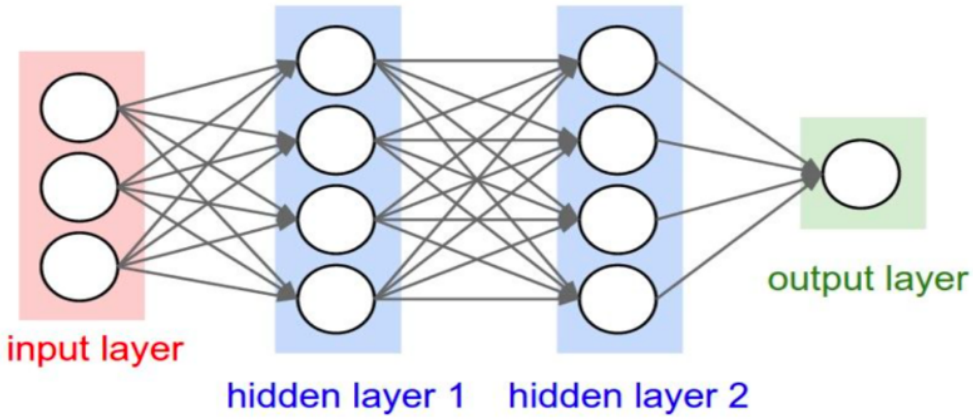

Figure 2.3: Architecture of a simple feedforward neural network with two hidden layers [19]

From individual neurons, we can construct a neural network of multiple neurons arranged into layers. The network can be tens of layers deep, which gives this field the name Deep Learning. In a neural network, each neuron will be connected to every neuron of the previous layer and every neuron of the next layer. The outputs of every neuron in a layer will be fed into every neuron of the next layer through the weighted connections until they reach the final layer. This output layer tells us the result we try to achieve in our task: whether a photo contains cats or dogs, whether a sound contains human voice or not, or what actions should an RL agent take.

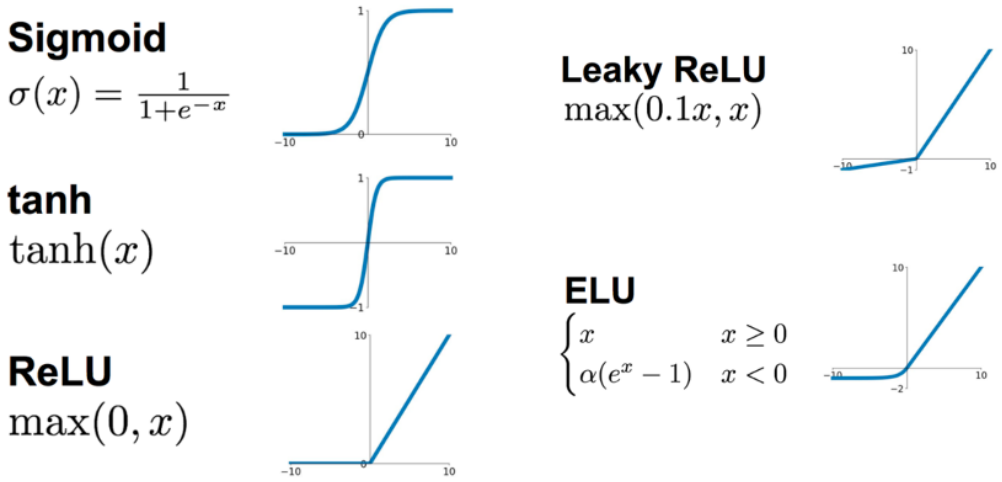

Figure 2.4: Some activation functions used in neural networks [19]

Let $x_j^{(l)}$ denote the output of the *j-th* neuron of the *l-th* layer in a neural network, $w_{ij}^{(l)}$ denote the weight of the connection to the *j-th* neuron of the *l-th* layer from the *i-th* neuron of the previous layer, and θ denote some activation function:

$$x_j^{(l)} = \theta(\sum_{i=0} w_{ij}^{(l)} x_i^{(l-1)})$$

With the basis of how to feed inputs through a neural network, we need a method for training such a network so that it will behave as we want. A toy neural network can contain 1 or 2 weight values, while the largest networks in the world [7] can contain up to hundreds of billions of weights. These weights are the parameters of a neural network, which need to be trained to achieve the result we desire. Weights can be randomly initialized to a starting value, and then they need to be updated for every mistake they make. One optimization algorithm to learn from mistakes is Gradient Descent, and this algorithm will tell us how to update the weights of our model.

2.2.2 Gradient Descent Algorithm

The first step to making a neural network learn from its mistake is to define what mistakes are. One such common technique in ML utilizes a loss function, also known as cost

function or reward function depending on context. A loss function is a function that tells us how much a model's outputs deviate from the desired outputs. There are many widely used loss functions, such as Sum of Square Errors, Mean of Square Errors, or Cross Entropy loss. Given the desired outputs Y and the model's outputs $\hat{Y}$, the Sum of Square Errors is:

$$SSE = \sum (Y_i - \hat{Y}_i)^2$$

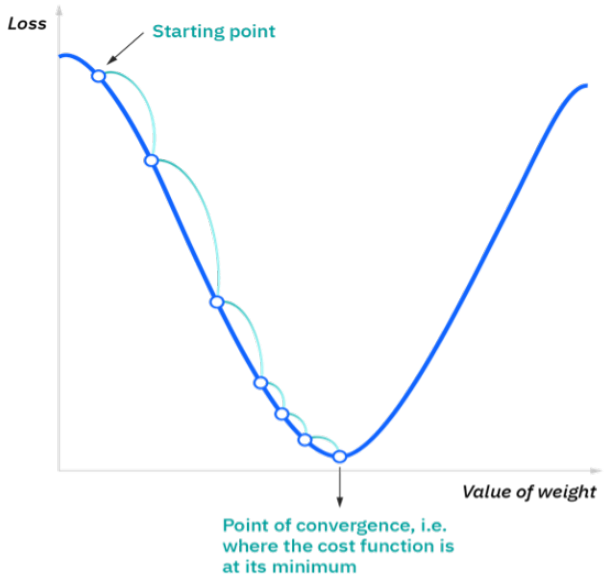

Figure 2.5: Illustration of Gradient Descent algorithm [3]

With the loss function defined, we can find the gradient of the loss with respect to the weights of our model. The gradient of a function can be understood as the direction and rate of change of the function at that location, or it can be visually understood as the slope. This step is crucial, as the gradient tells us about the direction in which the function is increasing in value. If we move in the direction of the gradient, the function will reach a higher value than the current one, and this technique is called Gradient Ascent. In the case of our neural network, we want to minimize this loss value to minimize the amount of mistakes it will make. To minimize the loss, the weights must be updated in the opposite

direction of the gradient, which gives this algorithm the name Gradient Descent. Given a learning rate parameter n, which dictates how fast the weight updates will be, the loss function L, and weight w, the rule for weight updates using Gradient Descent is:

$$w \leftarrow w - n\frac{\partial_{loss}}{\partial_w}$$

With the use of the Gradient Descent algorithm, we can train our neural network by showing it a large amount of data, calculating the mistakes it makes with a loss function, and repeatedly performing weight updates. Given enough training, the network's loss will be minimized, and it can now perform the task we train it to do. We can utilize this powerful learning model to solve Reinforcement Learning problems. The subfield combining Deep Learning with Reinforcement Learning is known as Deep Reinforcement Learning.

2.2.3 Deep Reinforcement Learning

Deep RL aims to solve RL problems through the use of deep neural networks. These networks can optimize the policy of an RL agent and lead it to accumulate the most amount of rewards. With the tools of Deep RL, I can apply them to control a dynamical system. The mathematical models and control of dynamical systems are the subjects of Control Theory.

2.3 Control Theory

John Doyle et al. wrote in the introduction of their textbook *Feedback Control Theory* [8]:

> "Without control systems there could be no manufacturing, no vehicles, no computers, no regulated environment—in short, no technology. Control systems are what make machines, in the broadest sense of the term, function as intended."

The field of Control Theory aims to design controllers for dynamical systems of any type. There are two types of controllers: feedforward (open-loop) and feedback (closed-loop). A feedforward controller is the simplest type of controller. Such a controller does not take into account any input, and it will control the system without the knowledge of what is happening with the system. An example of this is a heater with a timer. The device will keep going until the timer goes to 0; at that point, it will stop. It does not need to know what the temperature of the room is. Thus, the feedforward controller has many drawbacks that prevents it from being used for controlling more complex systems. This is where a feedback controller comes in.

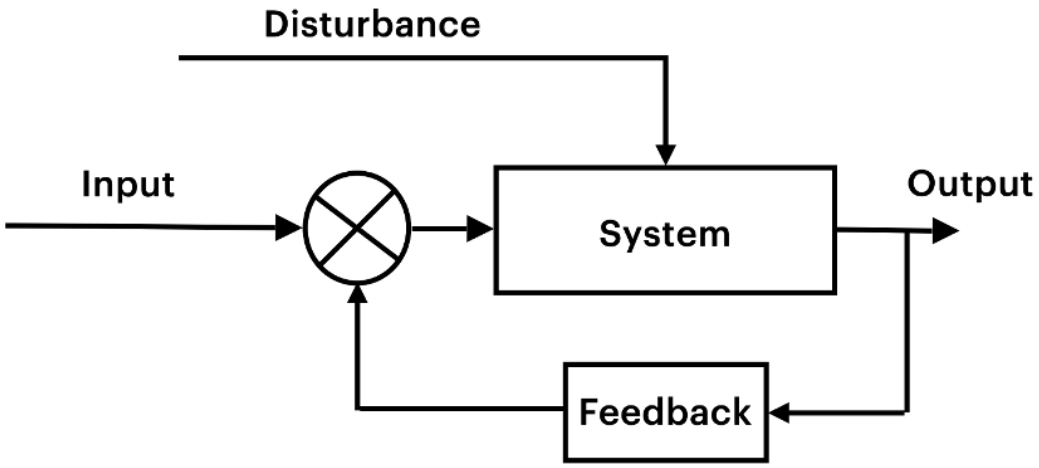

Figure 2.6: A simple feedback control system [17]

The feedback controller takes in the input from the system to determine the control signal. Back to the heating example, this time with an electric kettle, feedback control is used to stop the kettle after the water reaches boiling temperature. The controller achieves this by taking temperature measurements from the system (the kettle) in order to make decisions accordingly. Such a controller can be used to control an agent or character in a video game.

Chapter 3

PROBLEM: GAME PLAYING

An agent operates inside a virtual world created using the OpenAI Gym package [6]. It can perform a finite number of actions inside this environment, and the environment returns to the agent its states and rewards. An episode of simulation ends when a termination condition is met.

3.1 Cart Pole

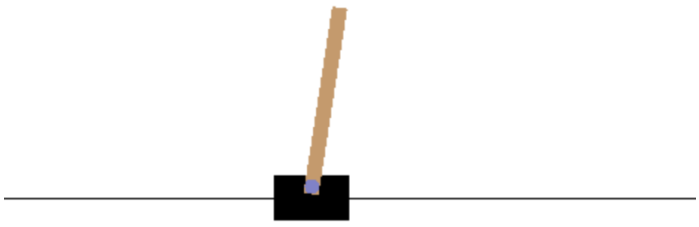

Figure 3.1: Cart Pole

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart, and the goal is to balance the pole by applying forces in the left and right direction on the cart. The environment attributes are summarized in Table 3.1.

- A Cart Pole agent has 2 possible actions:
  - Push cart to the left

Action Space	Discrete(2)
Observation Shape	(4,)
Observation High	[4.8 inf 0.42 inf]
Observation Low	[-4.8 -inf -0.42 -inf]

Table 3.1: Attributes of CartPole-v1 environment

  - Push cart to the right

- All observations are assigned a uniformly random value in (-0.05, 0.05). The observation returns 4 values:
  - Cart positions
  - Cart velocity
  - Pole angle
  - Pole angular velocity
- A reward of +1 is given to the agent for every step taken, with a threshold of 475. This means keeping the pole upright for longer will give the agent more rewards.
- The simulation episode ends if any of the following occurs:
  - Termination: Pole Angle is greater than $\pm 12°$
  - Termination: Cart Position is greater than ± 2.4 (center of the cart reaches the edge of the display)
  - Truncation: Episode length is greater than 500

3.2 Lunar Lander

This environment is a classic rocket trajectory optimization problem. The landing pad is always at coordinates (0,0). The coordinates are the first two numbers in the state vector.

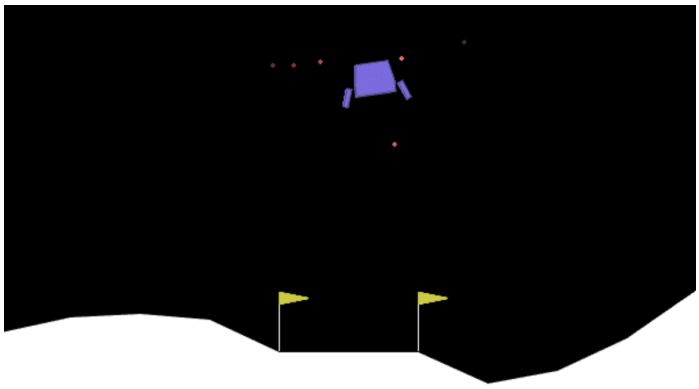

Figure 3.2: Lunar Lander

Landing outside of the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. The environment attributes are summarized in Table 3.2.

Action Space	Discrete(4)
Observation Shape	(8,)
Observation High	[1.5 1.5 5. 5. 3.14 5. 1. 1.]
Observation Low	[-1.5 -1.5 -5. -5. -3.14 -5. -0. -0.]

Table 3.2: Attributes of LunarLander-v2 environment

- There are four discrete actions available:
  - Do nothing
  - Fire left orientation engine
  - Fire main engine
  - Fire right orientation engine
- The observation returns 8 values:
  - The coordinates of the lander in x & y

  - Lander's linear velocities in x & y
  - Lander's angle
  - Lander's angular velocity
  - Two booleans that represent whether each leg is in contact with the ground or not.

- The lander starts at the top center of the viewport with a random initial force applied to its center of mass.
- Reward for moving from the top of the screen to the landing pad and coming to rest is about 100-140 points. If the lander moves away from the landing pad, it loses reward. If the lander crashes, it receives an additional -100 points. If it comes to rest, it receives an additional +100 points. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. Solved is 200 points.
- The simulation episode ends if any of the following occurs:
  - The lander crashes (the lander body gets in contact with the moon)
  - The lander gets outside of the viewport (x coordinate is greater than 1)
  - The lander doesn't move and doesn't collide with any other body

3.3 Pixel Copter

Pixelcopter is a side-scrolling game where the agent must successfully navigate through a cavern. This is a clone of the popular helicopter game, but the player is just a humble pixel. This game is simulated using the PyGame Learning Environment extension of Gym [15]. The environment attributes are summarized in Table 3.3.

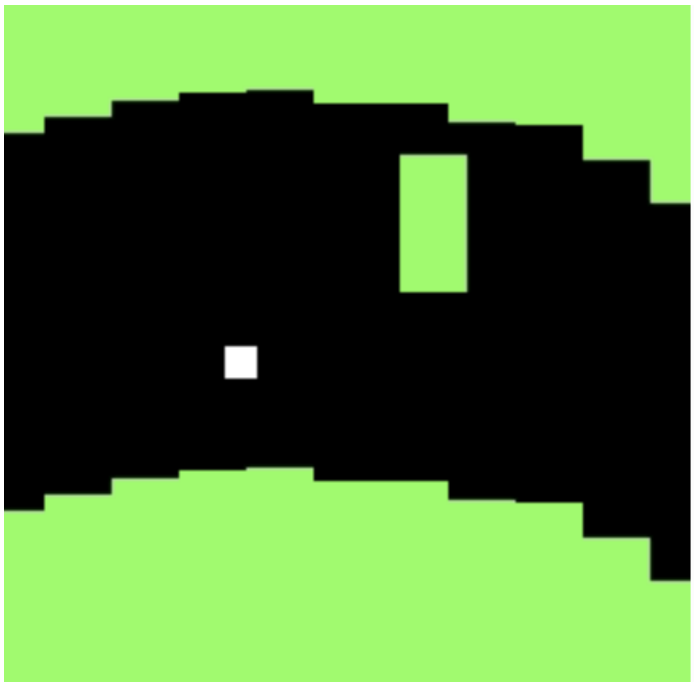

Figure 3.3: Pixel Copter

Action Space	Discrete(2)
Observation Shape	(7,)

Table 3.3: Attributes of Pixelcopter-PLE-v0 environment

- The copter can make 2 valid actions:
  - Do nothing
  - Use thrust to accelerate upward.
- The observation returns 7 values:
  - Player y position
  - Player velocity
  - Player distance to floor
  - Player distance to ceiling

  - Next block x distance to player
  - Next blocks top y location
  - Next blocks bottom y location

- For each vertical block it passes through, it gains a positive reward of +1. Each time a terminal state is reached, it receives a negative reward of -1.
- An episode terminates when the copter makes contact with any obstacle.

Chapter 4

APPROACH

A neural network is used to control an agent that plays these video games. To learn and improve the video game agent, the REINFORCE algorithm [18] is used to update this policy network. REINFORCE belongs to a class of algorithms known as Policy Gradient, which uses gradient-based optimization methods to find the best parameters that control a policy. The specific implementation is described in **Algorithm 1** [1].

Algorithm 1 REINFORCE algorithm for updating policy network

procedure REINFORCE
    Initialize policy π_θ
    repeat
        Generate an episode $S_0, A_0, r_0, ..., S_{T-1}, A_{T-1}, r_{T-1}$ following π_θ
        for t from $T-1$ to 0 **do**
            $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$
        $L(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} G_t log\ \pi_\theta(A_t|S_t)$
        Optimize π_θ using $\nabla L(\theta)$
    until stop

The policy network π is randomly initialized with parameters θ. The agent performs actions according to policy π_θ, generating three values State S, Action A, and Reward r every time step. Next, the discounted reward G_t of time step t is calculated with discount factor γ, which makes reward too far in the future less important. The total expected reward $L(\theta)$ is calculated as the sum of all discounted rewards times the probability of an action given the state. To reach a good policy π_θ, gradient-based optimization is used

to maximize $L(\theta)$. For the purpose of implementation using deep learning packages with built-in gradient descent optimizers, $-L(\theta)$ can be minimized to achieve the same result.

Chapter 5

EXPERIMENT AND RESULT

5.1 Experiments

All experiments are run on cloud Jupyter Notebook services like Google Colab and Kaggle [5, 2]. These platforms provide the software environment needed for running experiments and GPU for accelerating training. GPUs used are NVIDIA Tesla T4 and P100. The numbers of training episodes are 150 for Cart Pole, 500 for Lunar Lander, and 10000 for Pixel Copter.

5.2 Results

The RL agent successfully plays all 3 games Cart Pole, Lunar Lander, and Pixel Copter. The training graphs are presented below.

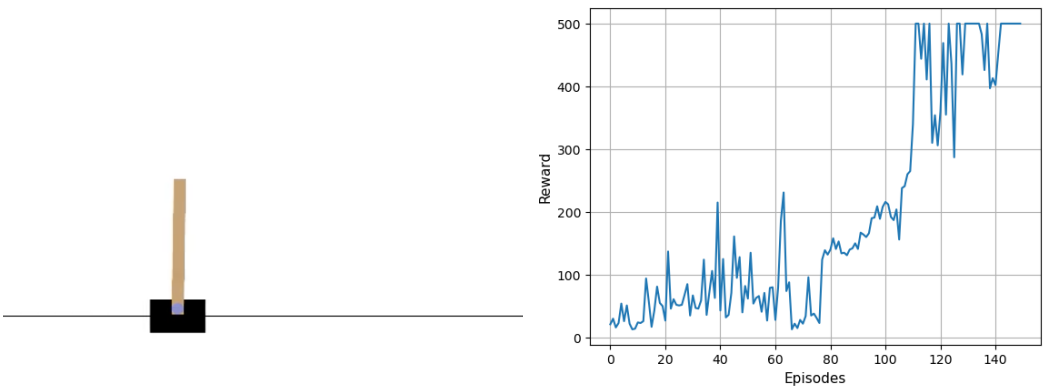

Figure 5.1: Cart Pole standing upright and its training graph

Although all agents eventually learn how to play these games, more complex games require more training episodes for the agent to successfully solve the problems. Cart Pole, being the most simple problem out of the three, only takes 150 episodes. Lunar Lander

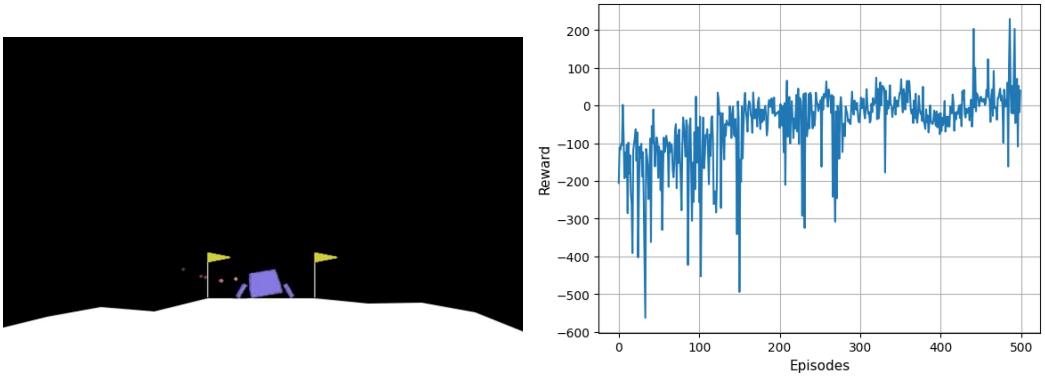

Figure 5.2: Lunar Lander successfully lands and its training graph

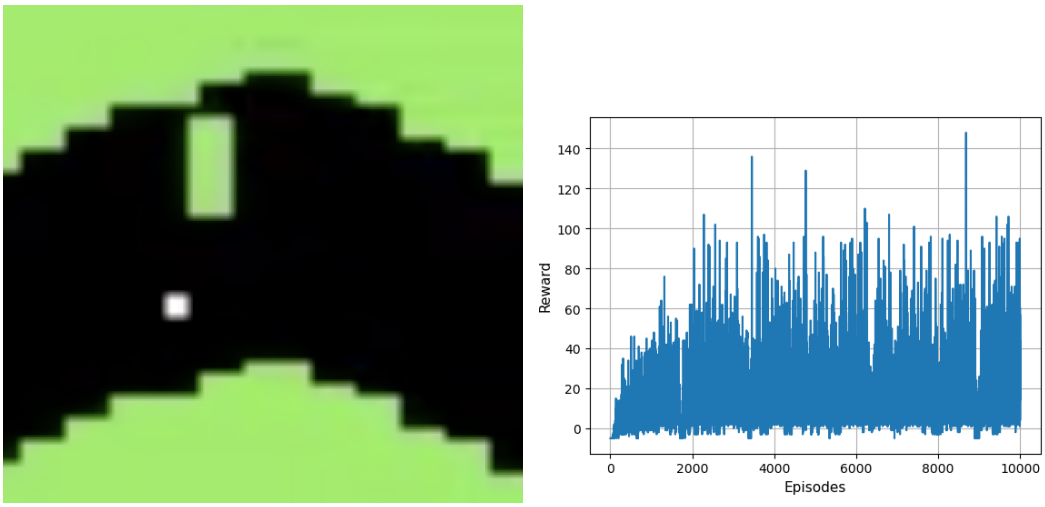

Figure 5.3: Pixel Copter avoids obstacle and its training graph

takes 500 episodes, while Pixel Copter takes 10000 episodes. Lunar Lander and Pixel Copter's training processes are also a lot more unstable due to the increased complexity. The training time varies from half an hour for Cart Pole and Lunar Lander to 3 hours for Pixel Copter.

Chapter 6

CONCLUSION

This project is a demonstration of Deep RL techniques in solving games. It shows how Policy Gradient methods can bring a policy from a random starting state with poor results to a well-optimized state that can perform the actions its designers want. Further improvements can be made to tackle increasingly complex problems. These improvements can be an increase in computing power and improved learning algorithms for more stable and efficient training.

REFERENCES

[1] "First agent: Playing cartpole-v1 - hugging face deep rl course", URL https://huggingface.co/learn/deep-rl-course/unit4/hands-on?fw=pt, "Accessed: 2023-04-10" (2023).

[2] "Kaggle", https://www.kaggle.com (2023).

[3] "What is gradient descent?", URL https://www.ibm.com/cloud/learn/gradient-descent (2023).

[4] Berner, C., G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski and S. Zhang, "Dota 2 with large scale deep reinforcement learning", (2019).

[5] Bisong, E., *Google Colaboratory*, pp. 59–64 (Apress, Berkeley, CA, 2019), URL https://doi.org/10.1007/978-1-4842-4470-8_7.

[6] Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, "Openai gym", (2016).

[7] Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever and D. Amodei, "Language models are few-shot learners", (2020).

[8] Doyle, J. C., B. A. Francis and A. Tannenbaum, *Feedback Control Theory* (Macmillan Publishing Company, 1992), illustrated edn.

[9] Fawzi, A., M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis and P. Kohli, "Discovering faster matrix multiplication algorithms with reinforcement learning", Nature **610**, 7930, 47–53, URL https://doi.org/10.1038/s41586-022-05172-4 (2022).

[10] Krizhevsky, A., I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in "Advances in Neural Information Processing Systems 25", edited by F. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger, pp. 1097–1105 (Curran Associates, Inc., 2012), URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[11] Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie,

A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, "Human-level control through deep reinforcement learning", Nature **518**, 7540, 529–533, URL https://doi.org/10.1038/nature14236 (2015).

[12] Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, "Mastering the game of go with deep neural networks and tree search", Nature **529**, 7587, 484–489, URL https://doi.org/10.1038/nature16961 (2016).

[13] Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play", Science **362**, 6419, 1140–1144, URL https://www.science.org/doi/abs/10.1126/science.aar6404 (2018).

[14] Sutton, R. S. and A. G. Barto, *Reinforcement Learning: An Introduction* (The MIT Press, 2018), second edn., URL http://incompleteideas.net/book/the-book-2nd.html.

[15] Tasfi, N., "Pygame learning environment", https://github.com/ntasfi/PyGame-Learning-Environment (2016).

[16] Vinyals, O., I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps and D. Silver, "Grandmaster level in starcraft ii using multi-agent reinforcement learning", Nature **575**, 7782, 350–354, URL https://doi.org/10.1038/s41586-019-1724-z (2019).

[17] Viswanathan, S., "A very simple physical representation of a feedback control system", URL https://medium.com/@sriramv/a-very-simple-physical-representation-of-a-feedback-control-system-34 (2020).

[18] Williams, R. J., "Simple statistical gradient-following algorithms for connectionist reinforcement learning", Machine Learning **8**, 3, 229–256, URL https://doi.org/10.1007/BF00992696 (1992).

[19] Zhao, Y., "Lecture slides in cisc6000 deep learning", (2022).