Pixel Perfect - Playing Atari Pong with Policy Gradient Reinforcement Learning

Christopher Camargo
ASEN 5264 - Decision Making Under Uncertainty
University of Colorado
Boulder, USA
Chris.Camargo@colorado.edu
https://github.com/camargo/dmu-project

***Abstract*—This paper investigates the performance of a reinforcement learning (RL) agent trained using a policy gradient to play Atari Pong. The study explores the impact of varying baseline functions and hyperparameters on the agent's learning efficiency and performance. By conducting a comparison of different settings, this work aims to identify optimal configurations for the RL agent, ultimately providing insights into the most effective strategies for achieving high performance in Atari Pong using a policy gradient.**

***Index Terms*—reinforcement learning, policy gradient, deep learning, atari, games**

I. Introduction

Reinforcement learning [10] has emerged as a powerful paradigm for training intelligent agents to interact with and learn from their environment to achieve optimal decision-making under uncertainty. One of the key breakthroughs in RL has been the application of deep learning techniques, which has led to the development of deep reinforcement learning (DRL) algorithms. DRL algorithms have shown remarkable success in various domains, ranging from robotics to game-playing, demonstrating their ability to handle complex tasks with high-dimensional input spaces.

Atari games, in particular, have become a popular benchmark [1] for evaluating the performance of RL agents, owing to their diverse challenges and the simplicity of their interfaces. Pong, one of the most iconic Atari games, has served as an excellent test bed for studying RL techniques, as it involves real-time decision-making in a relatively simple environment.

In this study, our primary focus lies on the application of policy gradient techniques to train a reinforcement learning (RL) agent for the game of Pong. Distinct from value function estimation approaches, policy gradient methods directly optimize the agent's policy. Our main objective is to examine the impact of various baseline functions and hyperparameters on the agent's performance. Crucial elements, such as the maximum steps allowed per episode, profoundly affect the learning dynamics of the agent, whereas the selection of an appropriate baseline function significantly influences the agent's capacity to extract pertinent features and rapidly generalize across diverse states. Through systematic experimentation, we strive to pinpoint optimal combinations of baseline functions and hyperparameters that lead to the maximization of the agent's performance.

The rest of this paper is organized as follows: Section 2 provides a background on related work in reinforcement learning and policy gradient methods; Section 3 details the problem formulation, including the Markov Decision Process, objective function, and policy gradient formulation; Section 4 discusses the solution approach, including the algorithm, hyperparameters, and neural network architecture chosen; Section 5 presents the results and analysis of our experiments; and finally, Section 6 concludes the paper with a summary of our findings and potential avenues for future research.

II. Background and Related Work

Reinforcement learning is a sub-field of machine learning, where an agent learns to make decisions by interacting with an environment to maximize cumulative reward. The core concept of RL is centered around the Markov Decision Process (MDP), which models the interactions between the agent and the environment in terms of states, actions, and rewards. The goal of the agent is to learn a policy, which is a mapping from states to actions, that maximizes the expected cumulative reward over time.

Policy gradient methods have been a subject of extensive research in the field of reinforcement learning. The REINFORCE algorithm, introduced by Williams [12], is a foundational policy gradient method that updates the agent's policy based on the gradient of the expected reward. The algorithm has been further improved with the introduction of variance reduction techniques, such as the use of baseline functions [11].

Deep reinforcement learning, which combines reinforcement learning with deep neural networks, has demonstrated impressive results on various tasks. Specifically, the deep Q-network (DQN) algorithm, which combines Q-learning with deep neural networks, has been shown to achieve human-level performance on a range of Atari games, including Pong [6].

There has also been research specifically focused on policy gradient methods for Atari games. For instance, Schulman et al. introduced Proximal Policy Optimization (PPO) [9], a more sample-efficient policy gradient method that demonstrated strong performance in several Atari games. Similarly, Karpathy et al. showed that using just a single layer neural network trained with a vanilla policy gradient can achieve superior performance in the game of Pong [3].

Despite the extensive research on RL algorithms for Atari games, there is always room for further exploration and understanding of the impact of baseline functions and hyperparameters on the performance of policy gradient methods. This study aims to contribute to this ongoing research by systematically investigating different configurations of the vanilla policy gradient algorithm for Atari Pong, offering additional insights into the factors that influence agent performance.

III. Problem Formulation

In this section, we formalize the problem of training a reinforcement learning agent to play Atari Pong using a policy gradient. We begin by describing the underlying Markov Decision Process and objective function, followed by the policy gradient definition and relevant equations.

A. Markov Decision Process

Atari Pong can be modeled as a Markov Decision Process (MDP), which is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where:

- $\mathcal{S}$ is the set of states representing the game's screen at each time step.
- $\mathcal{A}$ is the set of actions available to the agent, which in the case of Pong include moving the paddle up, or down.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function, with $\mathcal{T}(s'|s, a)$ denoting the probability of transitioning from state s to state s' after taking action a.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, where $\mathcal{R}(s, a)$ is the reward received after taking action a in state s.
- $\gamma \in [0, 1]$ is the discount factor, which balances the importance of immediate and future rewards.

In reinforcement learning the transition function $\mathcal{T}$ and reward function $\mathcal{R}$ are not known. Instead they are only environmental observations given by an episodic simulator - in this case the Atari emulator provided by the OpenAI Gymnasium [1].

B. Objective Function

The objective of the agent is to learn a policy $\pi_\theta : \mathcal{S} \to \mathcal{A}$, which maps states to actions, that maximizes the expected cumulative reward for a trajectory $\tau \sim \pi_\theta$ of length T, also known as the return:

$$\tau = s_0, a_0, r_0, \ldots, s_T, a_T, r_T \tag{1}$$

$$R_0(\tau) = \sum_{t=0}^{T} \gamma^t r_{t+1} \tag{2}$$

We use an optimization of the return called "reward-to-go", which only calculates reward from the current time-step onward. This helps reduce variance in the return estimate [5]. The reward-to-go return is defined as:

$$R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{3}$$

We model the policy π_θ as a parameterized three-layer neural network with parameters θ. The objective function J is the expected return over all trajectories τ generated by the agent.

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R_0(\tau)] \tag{4}$$

The goal is to find the optimal parameters θ that maximize the expected return:

$$\max_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R_0(\tau)] \tag{5}$$

C. Policy Gradient

To optimize the policy π_θ we employ policy gradient methods that rely on the gradient of the expected return with respect to the policy parameters (for a full derivation of this expression see [2]):

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R_t(\tau)\right] \tag{6}$$

In this project we estimate the expectation $\mathbb{E}_{\tau\sim\pi_\theta}$ by using Monte Carlo sampling of single trajectories. This gives:

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R_t(\tau) \tag{7}$$

Additionally we explore using baseline subtraction [5] to further reduce variance in the gradient estimate. In this paper we estimate $b(s_t)$ as the mean of $R_t(\tau)$, however many other "advantage function" estimations are possible [11]:

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t(\tau) - b(s_t)) \tag{8}$$

The gradient $\nabla_\theta J(\pi_\theta)$ is then used to update the policy parameters using gradient ascent and back-propagation [8]:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta) \tag{9}$$

IV. Solution Approach

In this section, we outline the solution approach for training a reinforcement learning agent to play Atari Pong using a policy gradient. We present the chosen policy gradient algorithm, hyperparameters, and neural network architecture.

Algorithm 1 Policy Gradient Reinforcement Learning

```
1:  Initialize single-layer neural-network $\pi_\theta$
2:  Initialize learning rate $\alpha$
3:  Initialize discount factor $\gamma$
4:  Initialize MAX_EPISODES
5:  Initialize $T$ = MAX_STEPS_PER_EPISODE
6:
7:  for $episode = 0, \ldots,$MAX_EPISODES do
8:      Sample $\tau = s_0, a_0, r_0, \ldots, s_T, a_T, r_T$ from $\pi_\theta$
9:      Calculate reward-to-go $R_t(\tau)$ (2)
10:     Set $b(s_t) = 0$ or $b(s_t) = mean(R_t(\tau))$
11:     Calculate $\nabla_\theta J(\pi_\theta)$ (8)
12:     Run gradient descent and back-propagate (9)
13: end for
```

A. *Algorithm*

We implemented the REINFORCE [12] vanilla policy gradient algorithm. In one version of the algorithm we used just reward-to-go returns (3), and in the other we used reward-to-go returns and baseline subtraction (8) (see Experiment 2). Algorithm 1 gives a sketch of the main algorithm.

Notice we sample directly from π_θ each episode using the current parameters θ, which makes the policy gradient algorithm "on-policy".

To calculate reward-to-go for each time-step we used a common linear-time trick [2] that iterates the rewards from a trajectory in reverse order.

Algorithm 2 Reward-to-Go Trick

```
1: Initialize each $R_t(\tau) = 0.0$
2: Initialize $reward\_to\_go = 0.0$
3:
4: for $t = r_T, \ldots, r_0$ do
5:     $reward\_to\_go = r_t + \gamma * reward\_to\_go$
6:     $R_t(\tau) = reward\_to\_go$
7: end for
```

B. *Hyperparameters*

We set the following seven different hyperparameters during training:

1) **Discount factor** (γ) - Makes rewards in the present worth more than rewards in the future.
2) **Input dimension size** - Number of neurons in the input dimension of network.
3) **Hidden dimension size** - Number of neurons in the hidden dimension of network.
4) **Output dimension size** - Number of neurons in the output dimension of network.
5) **Learning rate** (α) - Size of step to take during gradient ascent.
6) **Max episodes** - Maximum number of episodes during training.
7) **Max steps per episode** - Maximum number of steps to take during each episode (i.e. the trajectory τ length T). Experiment 1 looks at changing this hyperparameter to see its effect on training performance.

C. *Neural Network Architecture*

We used a three-layer fully-connected perceptron [7] neural network with 6400 input neurons, 200 hidden neurons, and 2 output neurons representing the two actions - paddle up or paddle down. This architecture was chosen to keep things simple, letting us focus more on the policy gradient algorithm itself rather than more complicated representations. See figure 1 for a simplified version of the architecture.

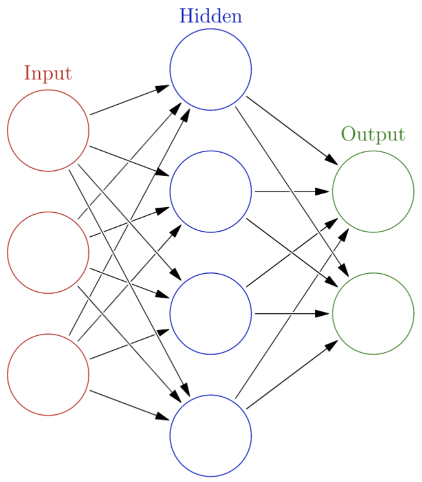

Fig. 1. Example of a three-layer fully-connected neural network with 3 input neurons, 4 hidden neurons, and 2 output neurons.

The OpenAI Gymnasium simulator gives Pong single-frame observations of 3-dimensional tensors of size $(210, 160, 3)$, which if spread to 1-dimension gives $100,800$. To make the observations smaller and easier to learn from, we used a pre-processing trick from [3] to convert the $100,800$ dimensions into 6400 dimensions (the input size to our network). This technique down-samples and crops each frame, leaving only the essential information we need to learn from.

Finally, we utilized the Adam optimization algorithm [4] as it offers a simple state-of-the art method for performing gradient steps.

V. Results

In this section, we present the results of two experiments aimed at training a reinforcement learning agent to play Atari Pong. We evaluate the performance of each experiment, and highlight the key findings from our analysis.

A. *Experimental Setup*

Before diving into the results, we briefly describe the experimental setup, including the hardware and software used, as well as the training and evaluation procedures.

1) **Hardware and software** - We conducted our experiments on a machine with an Intel Core i9 CPU with 16 GB of RAM. The software stack included MacOS 12.6, Python 3.11, and PyTorch.
2) **Training procedure** - For each configuration change the agent was trained for $20,000$ max episodes, with each episode consisting of a parameterized trajectory length. For simplicity the policy parameters were updated using a single-trajectory. Future work could explore doing updates using the average of a multi-trajectory batch.
3) **Evaluation procedure** - The performance of the trained agents was evaluated by measuring the win-rate of the agent following the learned policy over 100 games.

B. Constant Hyperparameters

This section lists the hyperparameter values that remained constant throughout all experiments.

1) Discount factor $(\gamma) = 0.99$
2) Learning rate $(\alpha) = 0.0001$
3) Max episodes $= 20,000$
4) Input dimension size $= 6400$
5) Hidden dimension size $= 200$
6) Output dimension size $= 2$

Future experiments could investigate changing these hyperparameters and seeing their effect on learning performance.

C. Experiment 1 - Max Steps per Episode

In our first experiment we chose to modulate the max steps per episode (i.e. trajectory length) hyperparameter because it affects training time, and it is advantageous to find a low number of steps while still maintaining high performance. We trained two agents using the reward-to-go algorithm with no baseline subtraction. The first agent had max steps per episode set to 1000 (Agent 1), and the second agent had max steps per episode set to 5000 (Agent 2).

Agent 1 trained for 12 hours and 29 minutes to reach $20,000$ epochs, and Agent 2 trained for 38 hours to reach $20,000$ epochs (nearly three times as long as Agent 1 due to 5 times more steps it needed to complete an episode).

Overall Agent 2 greatly outperformed Agent 1, winning 95% of games on average, where Agent 2 only won 28% of games on average. At the end of training, over the last 100 games Agent 2's total reward was about 8 times higher on average over Agent 1, which is reflected in the higher win-rate (see figure 2).

We also visualized learned weights of two neurons for Agent 1 and Agent 2 (see figure 3). White pixels are positive weights, and black pixels are negative weights. On the left of each neuron you can see an thin line of alternating gray and white pixels, indicating where the computer paddle moves. You can also see alternating gray and white diagonal lines indicating the ball in the center of the screen moving between each side. For the agent's paddle on the right side of each neuron, Agent 2's weights have slightly more structure than Agent 1's

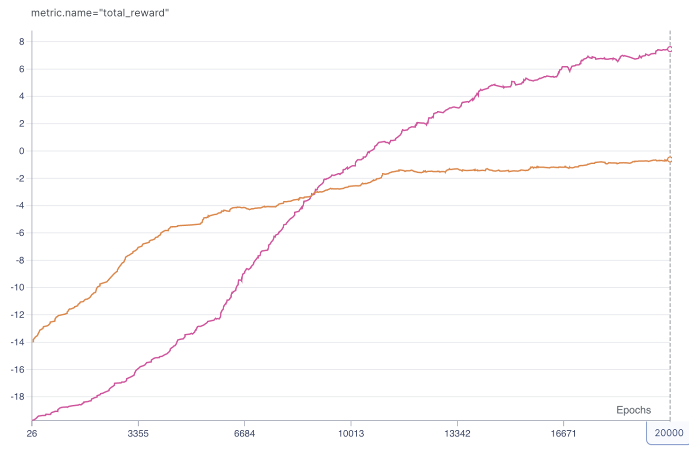

Fig. 2. Experiment 1 - Total reward curve over $20,000$ epochs. Agent 1 is orange, and Agent 2 is pink. Each point is the smoothed average total reward over 100 epochs. Agent 1's average total reward never gets above 0, likely because it does not have enough steps to learn to deflect multiple return hits from the computer player.

weights, which is likely due to the increased steps Agent 2 was able to learn from to deflect the ball.

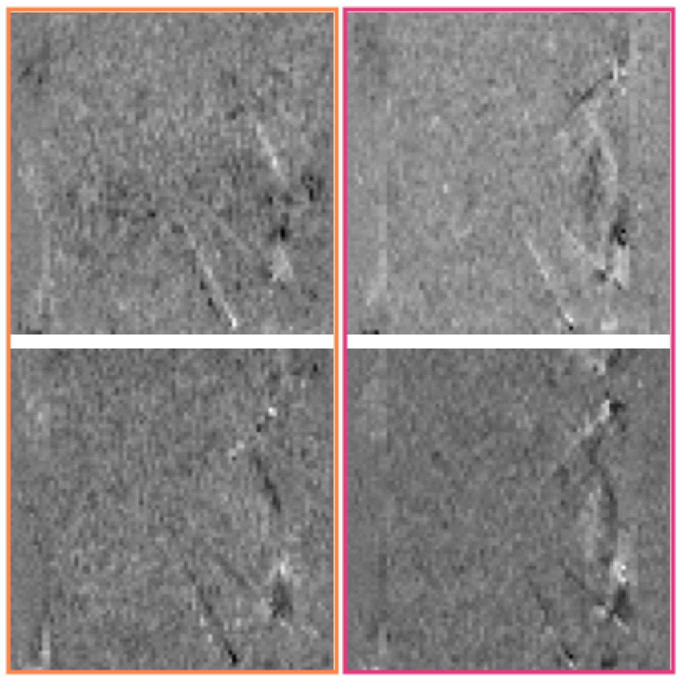

Fig. 3. Experiment 1 - Learned weights of two neurons. Agent 1 is the left column outlined in orange, and Agent 2 in the right column outlined in pink. White pixels are positive weights, and black pixels are negative weights.

Table I summarizes the statistics of training all agents over the $20,000$ epochs. You can see the higher loss variance and total reward variance in Agent 2 over Agent 1, which is reflected in how much longer it took to converge. However in the end that variance payed off since it gave the agent more experience, which translated to a higher win rate. We conclude from this experiment that having a larger max steps per episode is advantageous at the cost of taking more time to train.

TABLE I
TRAINING LOSS AND TOTAL REWARD

Agent	Metric	Mean	Variance	STD[a]	SEM[b]
1	loss	-130.51	64041.91	253.07	1.79
2	loss	-231.95	165097.00	406.32	2.87
3	loss	-9.52	263.11	16.22	0.11
1	total reward	-3.95	21.45	4.63	0.03
2	total reward	-3.54	119.93	10.95	0.08
3	total reward	0.19	80.65	8.98	0.06

[a]Standard deviation
[b]Standard error of the mean

D. Experiment 2 - Baseline Subtraction

The second experiment trained an additional agent (Agent 3) using the reward-to-go algorithm with baseline subtraction. The baseline subtraction set $b(s_t) = mean(R_t(\tau))$ from equation (8). We compared the results of Agent 3 with Agent 2 from the previous experiment to see if performance improved by using baseline subtraction.

Agent 3 trained for 35 hours to reach $20,000$ epochs, and Agent 2 trained for 38 hours to reach $20,000$ epochs. Thus Agent 3 was about 3 hours faster (8%) than Agent 2, likely because of the decrease in gradient variance from baseline subtraction. By reducing the variance the learning algorithm converges more quickly because updates to the policy parameters are more stable and less susceptible to random fluctuations. Figure 4 shows the gradient estimate difference between Agent 2 and Agent 3.

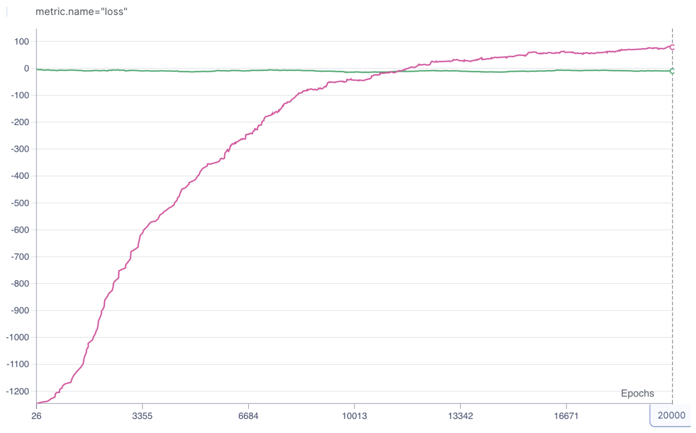

Fig. 4. Experiment 2 - Smoothed average over 100 epochs of the gradient estimate ("loss") of Agent 2 and Agent 3 during training. Agent 2 is pink and Agent 3 is green. You can visibly see the greatly decreased variance in Agent 3's gradient signal as a result of subtracting the baseline from the reward-to-go.

Table I summarizes the loss variance and total reward variance between Agent 2 and Agent 3. Agent 2 had a loss variance of 165097.00, and Agent 3 had a loss variance of 263.11. A nearly 200% reduction from the baseline subtraction! Similarly Agent 2 had a total reward variance of 119.93, and Agent 3 had a total reward variance of 80.65, a roughly 40% reduction.

Overall both Agent 2 and Agent 3 achieved superior performance over the Pong computer, but Agent 3 achieved it in a shorter amount of time. This is likely due to the smaller amount of gradient variance achieved by subtracting the baseline. A future experiment could compare performance of Agent 3 to other agents trained with different baseline functions.

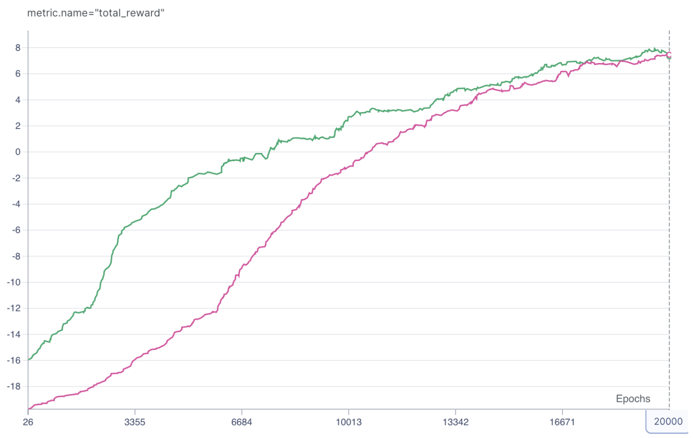

Fig. 5. Experiment 2 - Smoothed average over 100 epochs of the total reward of Agent 2 and Agent 3 during training. Agent 2 is pink and Agent 3 is green. Both converge to the same total reward, but Agent 3 does it in less time likely because of decreased gradient variance.

VI. CONCLUSION

In closing, we have conducted an investigation on training a reinforcement learning agent to proficiently play Atari Pong utilizing a policy gradient algorithm with deep neural networks. Our research encompasses two primary experiments: the first comparing agents trained with varying trajectory lengths, while the second contrasts agents trained with reward-to-go gradients to those with reward-to-go baseline subtracted gradients.

After training three agents for a combined total of over 85 hours, we achieved superior performance against the Pong computer player. Our findings indicate that trajectory length plays a significant role in Pong, with longer trajectories demonstrating a positive correlation with higher maximum scores against the computer. Additionally, we have corroborated that incorporating baseline subtraction effectively reduces gradient variance, ultimately leading to accelerated training times and achieves equivalent performance levels compared to solely just using reward-to-go.

Future research may explore the optimization of additional hyperparameters, such as batch size and learning rate, to further enhance the performance of the agents. Moreover, the incorporation of more advanced neural network architectures, such as convolutional neural networks for processing game frames, could be investigated. Additionally, examining the potential benefits of utilizing more sophisticated policy gradient algorithms, such as Proximal Policy Optimization (PPO) [9], might yield valuable insights and improvements.

VII. Contributions and Release

The authors grant permission for this report to be posted publicly. All algorithms for this project were implemented from scratch with Python and PyTorch using the references listed below. The code repository is on GitHub at https://github.com/camargo/dmu-project, and includes a video in the README of our best Pong agent trained in this study. It also includes the models so experiments can be replicated.

References

[1] Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. "Openai gym." arXiv preprint arXiv:1606.01540 (2016).

[2] Graesser, Laura, and Wah Loon Keng. Foundations of deep reinforcement learning. Addison-Wesley Professional, 2019.

[3] Karpathy, Andrej. "Deep Reinforcement Learning: Pong from Pixels." *Andrej Karpathy Blog*, 31 May 2016, https://karpathy.github.io/2016/05/31/rl/.

[4] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[5] Kochenderfer, Mykel J., Tim A. Wheeler, and Kyle H. Wray. Algorithms for decision making. MIT press, 2022.

[6] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[7] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65, no. 6 (1958): 386.

[8] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." nature 323, no. 6088 (1986): 533-536.

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

[10] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[11] Sutton, Richard S., David McAllester, Satinder Singh, and Yishay Mansour. "Policy gradient methods for reinforcement learning with function approximation." Advances in neural information processing systems 12 (1999).

[12] Williams, Ronald J. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." Reinforcement learning (1992): 5-32.