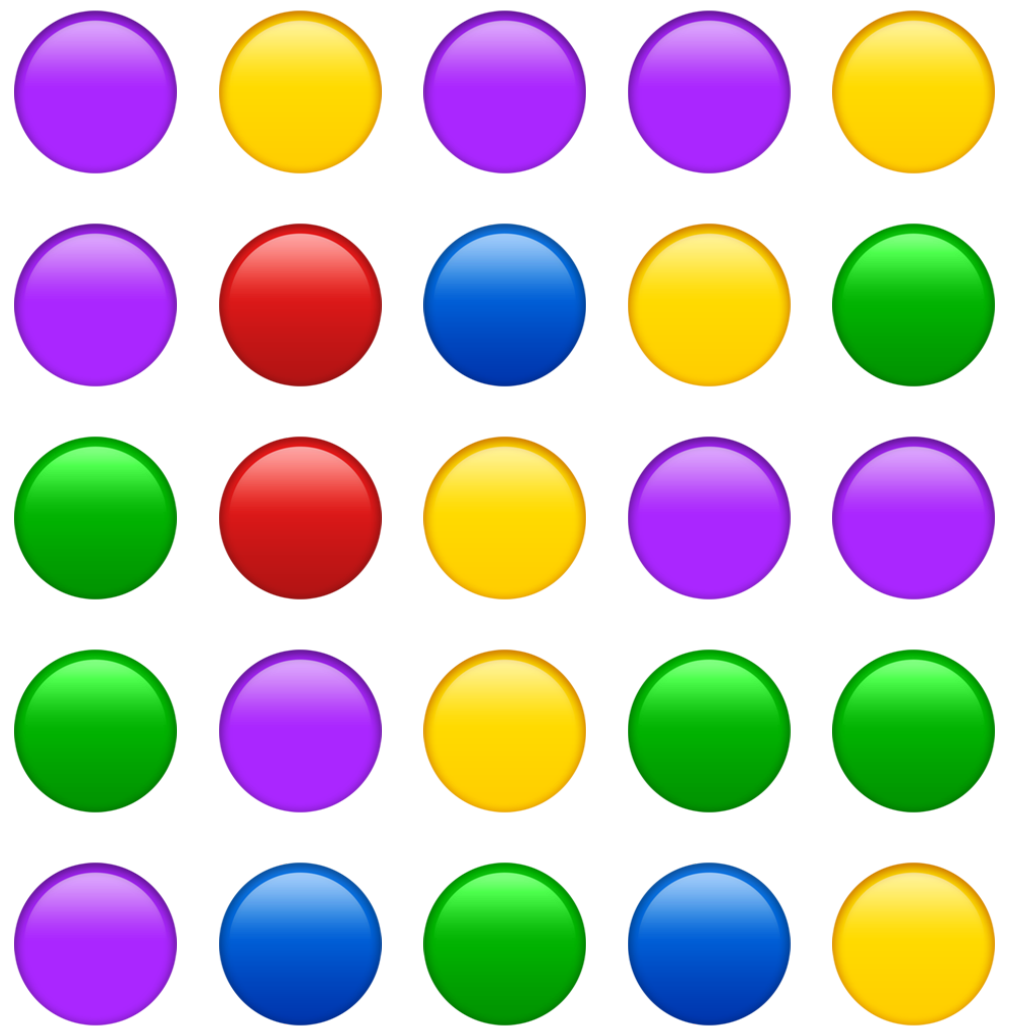

Playtesting Match 3 Games with PPO

Master's thesis in Mathematical Sciences

Stanislaw Malec

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Playtesting Match 3 Games with PPO

Stanislaw Malec

Department of Mathematical Sciences
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Playtesting Match 3 Games with PPO

Stanislaw Malec

© Stanislaw Malec, 2023.

Supervisor:
Morteza Haghir Chehreghani, Department of Computer Science (Chalmers)

Industrial Supervisors:
Joakim Bergdahl, SEED (Electronic Arts)
Alessandro Sestini, SEED (Electronic Arts)
Linus Gisslén, SEED (Electronic Arts)

Examiner:
Adam Andersson, Department of Mathematical Sciences (Chalmers)

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Playtesting Match 3 Games with PPO

Stanislaw Malec
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The training of proximal policy optimization agents with action masking on stochastic match-3 environments is explored in this thesis. A performant, feature-rich match-3 simulator is developed, and experiments demonstrate improved performance over a random policy on both seen and unseen levels. Furthermore, the best generalization performance is achieved when training is done by sampling levels from a subset of levels.

Keywords: reinforcement learning, match-3

Acknowledgements

I would like to express many thanks to my supervisor Joakim Bergdahl whose unwavering support and guidance were instrumental in the completion of this endeavor. I would also like to thank Morteza, Linus, and Alessandro for invaluable feedback and insights.

Stanislaw Malec, Gothenburg, 2023-10-16

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Match-3 . . . 1
- 1.2 Research Question . . . 2
- 1.3 Related Work . . . 3
- 1.4 Purpose . . . 3
- 1.5 Motivation . . . 3
- 1.6 Electronic Arts . . . 4

2 Background 5
- 2.1 Artificial Neural Networks . . . 5
  - 2.1.1 Loss Function . . . 6
  - 2.1.2 Backpropagation . . . 6
  - 2.1.3 Gradient Descent . . . 7
  - 2.1.4 Learning Rate . . . 8
- 2.2 Convolutional Neural Networks . . . 9
- 2.3 Reinforcement Learning . . . 11
  - 2.3.1 Core Elements of Reinforcement Learning . . . 12
  - 2.3.2 Markov Decision Processes . . . 12
  - 2.3.3 Policy and Value Functions . . . 12
  - 2.3.4 Advantage Function . . . 13
- 2.4 Policy Gradient Methods . . . 13
  - 2.4.1 REINFORCE . . . 14
  - 2.4.2 Proximal Policy Optimization (PPO) . . . 14
- 2.5 Actor-Critic Architecture . . . 15
- 2.6 Action Masking . . . 16
- 2.7 Evaluation Metrics . . . 16

3 Implementation 17
- 3.1 Simulator . . . 17
  - 3.1.1 Simulator with Best of Both Worlds: C++ and Python . . . 17
  - 3.1.2 Reproducibility . . . 17

3.1.3 Interoperability between the Simulator and Existing Match-3 Games . . . 18
3.2 Environment . . . 18
3.2.1 Visualizing Trajectories . . . 18
3.2.2 Color Blocks . . . 19
3.2.3 Powerups . . . 19
3.2.4 Powerup Combinations . . . 20
3.2.5 Obstacles . . . 20
3.2.6 Objectives . . . 21
3.2.7 Gravity and Cascading . . . 21
3.2.8 Swap Limit . . . 21
3.3 State Space . . . 21
3.4 Action Space . . . 22
3.5 Reward Function . . . 23
3.6 PPO Agent . . . 24
3.7 Hardware . . . 25

4 Results **27**
4.1 Experiments . . . 27
4.1.1 Experimental Setup . . . 27
4.1.2 Stability of Training . . . 27
4.1.3 Training on Single Levels . . . 28
4.1.4 Training on Subset {6,7,10,25} Sequentially . . . 31
4.1.5 Training on Subset {27,29,47,51} Sequentially . . . 33
4.1.6 Training on Subset {6,7,10,25} with Uniform Sampling . . . 34
4.1.7 Training on Subset {27,29,47,51} with Uniform Sampling . . . 34
4.1.8 Training on All Levels with Uniform Sampling . . . 35
4.1.9 Benchmark Comparison . . . 36
4.2 Discussion . . . 38
4.2.1 Training on Single Levels . . . 38
4.2.2 Training on Subsets Sequentially . . . 38
4.2.3 Training on Subsets with Uniform Sampling . . . 38
4.2.4 Training Steps . . . 38
4.2.5 Action masking . . . 38
4.2.6 Problematic Levels . . . 39
4.2.7 Ethical Considerations . . . 40

5 Conclusion **41**
5.1 Future Work . . . 41

Bibliography **43**

List of Figures

1.1 Screenshots of two popular match-3 games. **Left:** Bejeweled. **Right:** Candy Crush Saga. 2

2.1 Example of a neural network with an input layer, two hidden layers, and an output layer. 5
2.2 Gradient descent for different learning rates. 8
2.3 High level comparison of the three main paradigms in machine learning. 11
2.4 Agent-environment Interaction in reinforcement learning. 12
2.5 Illustration of clipping of the objective for (left) positive advantages and (right) negative advantages [15]. 15

3.1 Overview of setup and how the simulator communicates with Python. 17
3.2 Overview of how levels are loaded into the simulator. 18
3.3 PPO network architecture 24

4.1 This plot illustrates the stability and reproducibility of training of training on a single level. 28
4.2 Four distinct models are trained independently on individual levels. The average trajectory length of each model is evaluated on its respective training level. Top-left: L6. Top-right: L7. Bottom-left: L10. Bottom-right: L25. 29
4.3 Four distinct models are trained independently on individual levels. The average success rate of each model is evaluated on its respective training level. Top-left: L6. Top-right: L7. Bottom-left: L10. Bottom-right: L25. 30
4.4 Comparison of average trajectory length between individually trained models. Top-left: L6. Top-right: L7. Bottom-left: L10. Bottom-right: L25. 31
4.5 Model $M_{6,7,10,25}$ is evaluated on its training levels 6,7,10,25. 31
4.6 Model $M_{6,7,10,25}$ is evaluated on all levels. 32
4.7 Model $M_{27,29,47,51}$ is evaluated on its training levels 27,29,47,51. 33
4.8 Model $M_{27,29,47,51}$ is evaluated on all levels. 33
4.9 Model $M_{\sim 6,7,10,25}$ is evaluated on all levels. 34
4.10 Model $M_{\sim 27,29,47,51}$ is evaluated on all levels. 34
4.11 Model $M_{\sim \text{all}}$ is evaluated on all levels. 35
4.12 Color-block configurations that lead to matches. 39

(a) G_1 39
(b) G_2 39
(c) G_3 39

List of Tables

3.1 Visualization of color blocks. 19
3.2 Visualization of powerups blocks. 20
3.3 Visualization of powerups combinations. 20
3.4 Visualization of obstacles. 21
3.5 Visualization of observation channels. 22

4.1 Average trajectory lengths of all models and a random agent. 36
4.2 Percentage difference in average trajectory length between all models and random agent. 37

1

Introduction

The popularity of video games has continued to rise every year, but the development of video games remains time-consuming and complex. Play-testing, the process of playing the game to ensure suitable difficulty and quality of experience, is a challenging aspect of game development because humans have to play through the game and provide feedback manually. One approach to play-testing that has gained traction in recent years is using reinforcement learning (RL) algorithms. RL is a subfield of machine learning that deals with learning optimal behavior in an environment. In this thesis, we investigate the feasibility of applying RL to a common video game category called match-3 which, in its simplest form, is a type of puzzle game in which three or more tiles of the same type are matched to score points.

1.1 Match-3

A match-3 game consists of levels that contain a $m \times n$ grid (typically 10×10, ± 5), where the player can swap adjacent cells to match up three in a row/column. The objective is to remove the required amount of specified block types in the level mission. In addition, many games expand upon the simple matching of three mechanics by introducing block types with different functions: horizontal/vertical blasters, TNT, and crates are examples of commonly found objects in popular match-3 games. Two popular match-3 games are shown in Figure 1.1.

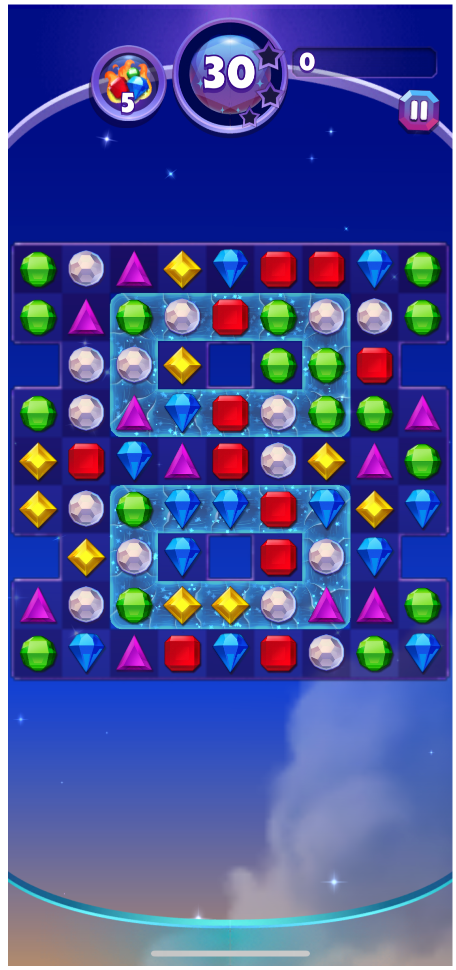

Figure 1.1: Screenshots of two popular match-3 games. **Left:** Bejeweled. **Right:** Candy Crush Saga.

Once a player matches blocks, they are removed causing new blocks to fall down from the top. This is where stochasticity comes into play: the new blocks are (usually uniformly) sampled from a distribution, making it computationally difficult to determine the sequence of swaps that leads to the completion of a level. For any given state in a level, there are many possible swaps leading to combinatorial explosion, and the resulting state from a swap can not be predicted with certainty. Guala et al. [1] showed that Candy Crush is NP-Hard by reducing the game from 1-in-3 positive 3SAT.

1.2 Research Question

This work attempts to answer the question whether it is possible to estimate level difficulty through Reinforcement Learning agents.

1.3 Related Work

Mnih et al [2] established the foundation of applying deep learning to RL problems by achieving high performance in a diverse set of atari games from just visual information in 2013. A combination of Deep Q-Networks and experience replay surpassed the performance of previous RL algorithms. However, DQNs suffer from instability during training and limited generalization capabilities, generally failing on unseen environments.

A common issue with RL agents is their tendency to overfit. Cobbe et al. [3] aim to address the issue assessing generalization ability in RL models by introducing a platformer game benchmark that generates levels procedurally. This thesis takes inspiration from this work by randomizing the seed for every trajectory.

There are a number of works that investigate applying RL match-3 games. Poromaa [4] used monte-carlo tree search on Candy Crush and found that MCTS could predict average human success rate better a small group of people attempting a levels 50 times. However, the downside of MCTS is long inference time. Shin et al. [5] train an advantage actor-critic (A2C) agent to play a match-3 game and achieve performance within 5% of margin to human performance. However, generalization is limited because the agent is trained on levels that have the same size and obstacles. In this thesis, models are evaluated on 29 levels with a diverse set of layouts and obstacles.

1.4 Purpose

Improving automated play-testing methods can have a significant impact on quality and success of video games. The purpose of this work is exploring applying PPO to match-3 games to accelerate the playesting process, improve games, and reduce game development costs.

1.5 Motivation

Match-3 games such as Bejeweled or Candy Crush represent a novel challenge for RL algorithms due to the vast state space and inherent randomness. The motivation for researching methods to solve match-3 games is twofold. First, mastering match-3 games can push the boundaries of current RL methods. Second, generalizable strategies might emerge. The methods developed in AlphaGo [6] to defeat the world chamption in the game of Go have been transferred and adapted to many applications including protein folding [7]. By researching methods for solving match-3 games, new techniques across diverse problem domains might emerge.

1.6 Electronic Arts

This thesis is carried out at SEED (Search for Extraordinary Experiences Division), a research division of Electronic Arts with a focus on applied research in the domain of interactive entertainment. SEED aims to drive advancements not just within EA but also contribute to the evolution of the global gaming industry.

2

Background

The following sections of this thesis describe machine learning, the main elements of reinforcement learning, and the Proximal Policy Optimization (PPO) algorithm.

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) are mathematical models inspired by the structure and function of the brain. ANNs consist of interconnected neurons organized into layers, where the output of one layer is fed into the next layer until the final layer produces the predicted output of the model. A simple neural network is presented in Figure 2.1.

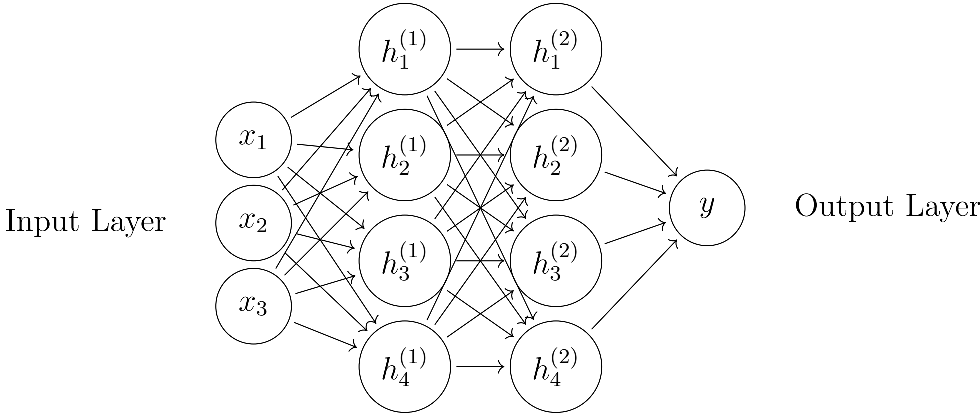

Figure 2.1: Example of a neural network with an input layer, two hidden layers, and an output layer.

The input layer is a vector $\boldsymbol{x} \in \mathbb{R}^n$, directly connecting to the input data. The input is fed to hidden layers $h^{(i)}$, which are parameterized by a weight matrices $W^{(i)}$ and bias vectors $\boldsymbol{b}^{(i)}$. A sequence of transition functions $f^{(i)}$ between layers i and $i-1$ can describe the flow, where each function is defined as:

$$f^{(i)}(h^{(i-1)}) = \sigma^{(i)}(W^{(i)}h^{(i-1)} + \boldsymbol{b}^{(i)}) \tag{2.1}$$

where σ is an activation function that introduces non-linearity. Without an appropriate activation function, the neural network would only be able to learn linear

relationships and would not be any more effective than simple methods like least squares. The success of neural networks is partly due to their ability to learn highly non-linear relationships, which activation functions enable.

One commonly used activation function is the rectified linear unit (ReLU), which is defined as:

$$\sigma_{ReLU}(x) = \max\{0, x\} \tag{2.2}$$

for some $x \in \mathbb{R}$. ReLU outputs the input directly if it is positive, which allows learning linear patterns. Otherwise, it outputs zero, which allows the modeling of non-linear relationships.

We can thus represent an ANN as a composition of functions:

$$f(\boldsymbol{x}) = (f^{(m)} \circ f^{(m-1)} \circ \cdots \circ f^{(0)})(\boldsymbol{x}) = \boldsymbol{y}. \tag{2.3}$$

2.1.1 Loss Function

Mean Square Error (MSE) is a differentiable loss function that is commonly used to optimize ANNs and can be expressed as:

$$L_{MSE}(\theta) = \frac{1}{n}\sum_{i=1}^{n}(f_\theta(\mathbf{x}_i) - \boldsymbol{y}_i)^2 \tag{2.4}$$

Given a set of samples $\boldsymbol{x}_i \in \mathbb{R}^n$ and their associated labels $\boldsymbol{y}_i \in \mathbb{R}$, a neural network defines a mapping $f_\theta : \mathbb{R}^n \mapsto \mathbb{R}$ (Equation 2.1) parameterized by $\theta = \{W^{(1)}, \boldsymbol{b}^{(1)}, \ldots, W^{(m)}, \boldsymbol{b}^{(m)}\}$ which represents all weights and biases. The goal is to adjust parameters θ to minimize the MSE between predictions $f_\theta(\boldsymbol{x_i})$ and target outputs $\boldsymbol{y}_i$.

2.1.2 Backpropagation

Rumelhart et al. [8] introduced backpropagation to train neural networks in the 1980s. Backpropagation allows the training of deep (many layers) neural networks by using the chain rule to compute the gradient of the loss function with respect to the weights and biases of the network. The gradient is used to adjust the parameters in the direction that minimizes the difference between the predicted output $f_\theta(\boldsymbol{x_i})$ and actual output $\boldsymbol{y}_i$.

Backpropagation works in two stages. It starts with forward propagation where a sample $\boldsymbol{x} \in X$ is passed into Equation 2.1 to produce $\boldsymbol{y}_{\text{prediction}}$, which is then used to compute the loss L. Once the loss is computed, the gradients of the error with respect to weights and biases are propagated backward through the network.

For a given weight $W_{ij}^{(i)}$, the partial derivative of the loss L with respect to $W_{ij}^{(i)}$ is computed using the chain rule:

$$\frac{\partial L}{\partial W_{ij}^{(i)}} = \frac{\partial L}{\partial f^{(i)}(h^{(i-1)})} \frac{\partial f^{(i)}(h^{(i-1)})}{\partial W_{ij}^{(i)}}. \tag{2.5}$$

The first partial derivative $\frac{\partial L}{\partial f^{(i)}(h^{(i-1)})}$ is the derivative of the loss function with respect to the output of ith layer. This derivative can be computed directly if $f^{(i)}$ is the output layer, and recursively for the hidden layers using the chain rule from the output layer back to the current layer (which is where the name "backpropagation" comes from). The second partial derivative, $\frac{\partial f^{(i)}(h^{(i-1)})}{\partial W_{ij}^{(i)}}$, is the derivative of the output of the ith layer with respect to its weight $W_{ij}^{(i)}$.

2.1.3 Gradient Descent

Gradient descent uses the gradients from backpropagation to update the network parameters θ in the direction that minimizes the loss function. There are three main variants: batch, mini-batch, and stochastic gradient descent.

Batch Gradient Descent

Batch gradient descent computes the gradients of the loss function for the entire training set, subsequently taking their average:

$$\theta = \theta - \eta \nabla_\theta \left[\frac{1}{N} \sum_{i=1}^{N} L(\theta; \boldsymbol{x}_i, \boldsymbol{y}_i) \right] \tag{2.6}$$

where η is the learning rate and N is the number of samples. This approach provides the most accurate update direction but is often impractical for large datasets.

Stochastic Gradient Descent

Stochastic Gradient descent (SGD) addresses the computational issue of batch gradient descent by updating the parameters θ after a single random sample $\boldsymbol{x} \sim \mathrm{Unif}(X)$ at each step:

$$\theta = \theta - \eta \nabla_\theta L(\theta; \boldsymbol{x}, \boldsymbol{y}). \tag{2.7}$$

SGD enables faster convergence. However, random sampling can cause the loss function to fluctuate, leading to less optimal convergence points.

Mini-batch Gradient Descent

A middle ground between batch gradient descent and SGD is mini-batch gradient descent, which updates the parameters based on a randomly sampled subset $B \subseteq X$

at each step:

$$\theta = \theta - \eta \nabla_\theta \left[\frac{1}{B} \sum_{i=1}^{B} L(\theta; \boldsymbol{x}_i, \boldsymbol{y}_i) \right]. \tag{2.8}$$

2.1.4 Learning Rate

The magnitude of the step taken in gradient descent is called the learning rate. It is important to set an appropriate learning rate to succeed in training. Figure 2.2 shows three different learning rates. It will take a long time to converge or get stuck on a local minimum if it is too small. If it is too large, it may never converge.

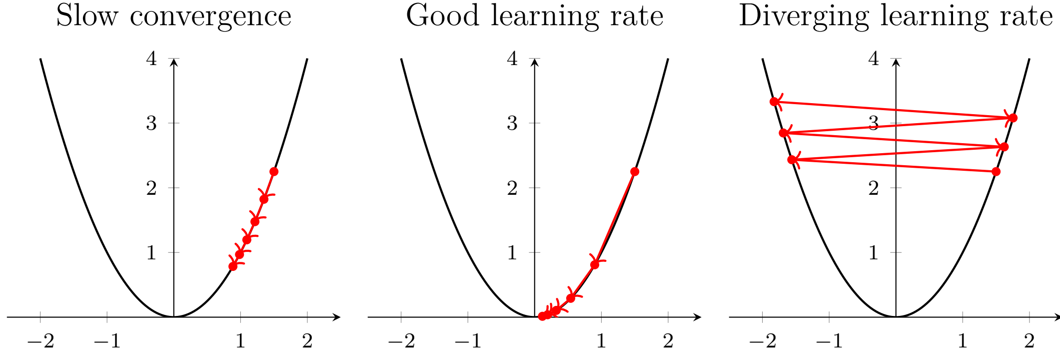

Figure 2.2: Gradient descent for different learning rates.

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of artificial neural network that works well for extracting features from image data [9]. In general, a convolution is an operation that takes an input function a and convolves it with a function b (referred to as filter/kernel) to produce a new function. Using certain filters, convolutions can extract information such as low frequencies in an audio signal. A general form of convolution can be described as:

$$(a * b)(t) = \int_{-\infty}^{\infty} a(\tau) \cdot b(t - \tau) \, d\tau. \tag{2.9}$$

Convolution is the core building block of a CNN. Convolutional layers consist of a set of filters with a small receptive field. Each filter is convolved across the width and height of the input in equidistant steps (referred to as stride), computing the dot product between entries of the filter and the input. This operation can be described as follows:

$$F_{i,j} = \sum_{m} \sum_{n} I_{i+m,j+n} \cdot W_{m,n} \tag{2.10}$$

where F is the output feature map, I is the input image, W is the filter weights. A convolutional layer often includes multiple filters, extracting different information, which helps improve performance.

Filters of size 3×3 are typically used in CNNs. This leads to detecting low-level features such as edge detection in initial layers. As we move deeper into the network, the low-level features are combined to form high-level abstract representations. This enables learning hierarchical structures. Additionally, because the filters are applied across the entire input, the learned representations are translation invariant [10].

A convolution layer can have multiple filters, each producing its own output feature map after sliding over the input. These output feature maps are then stacked along the last dimension, where the size of this dimension is equal to the number of filters. For example, applying n different filters ($3 \times 3 \times 3$) to a rgb image will produce output with n feature maps. The following convolutional layer would thus require $3 \times 3 \times n$ filters. Applying m such filters results in m new feature maps.

Pooling Layers

Pooling layers often follow convolutional layers to reduce the number of parameters, lowering computational costs and preventing overfitting. By aggregating neighboring regions, it makes subsequent layers more efficient. Additionally, it helps with feature extraction as it selects the most dominant signals. The pooling operation can be described as follows:

$$P_{i,j} = \max_{m,n} I_{i+m,j+n} \tag{2.11}$$

where $P_{i,j}$ is the output of the pooling layer, $I_{i+m,j+n}$ is the input, and N is the neighborhood defined by the pooling window size.

2.3 Reinforcement Learning

Reinforcement learning (RL) is a method to solve control problems. More specifically, RL focuses on learning a policy that maximizes the cumulative reward over time. The development of reinforcement learning was partly inspired by studies of animal behavior by Skinner [11] in the 1950s. Skinner observed that animals, such as rats and pigeons, could learn through trial and error by receiving rewards or punishments based on their actions.

Work on developing RL began in the 1980s when early algorithms like Q-learning got introduced by Watkins et al. [12]. The first algorithms were primarily designed for small toy problems and were limited by the computational power available at the time. In the 1990s, researchers explored using neural networks to handle more complex problems. A breakthrough occurred in 2013, when Mnih et al. [2] combined a convolutional neural network with Q-learning to surpass human performance in Atari games. They demonstrated that RL can be used to learn from high dimensional input (just raw pixel data) and solve complex problems.

Figure 2.3 shows an overview of supervised learning, reinforcement learning, and unsupervised learning. In contrast to traditional machine learning, where a labeled dataset is necessary to learn a mapping from observations to actions, reinforcement learning algorithms learn optimal control by trial and error from interacting with the environment. By receiving rewards and punishments depending on the current state and action taken, the algorithm tries to maximize the cumulative reward over time. Reinforcement learning also differs from unsupervised learning, which deals with finding patterns in unlabeled data without feedback.

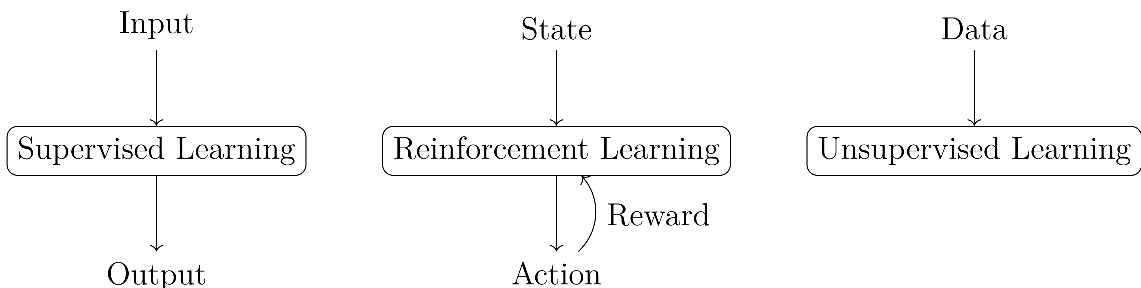

Figure 2.3: High level comparison of the three main paradigms in machine learning.

2.3.1 Core Elements of Reinforcement Learning

The agent and the environment are the core elements of RL and are shown in Figure 2.4. The agent is a decision-making entity that takes actions based on the current state. At each timestep $t \in \mathbb{N}$, the agent is given a state $s_t \in \mathcal{S}$ and selects some action $a_t \in \mathcal{A}(s_t)$.

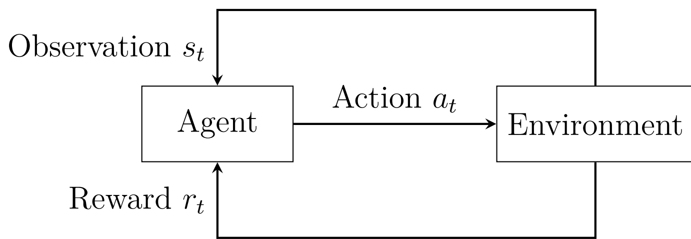

Figure 2.4: Agent-environment Interaction in reinforcement learning.

2.3.2 Markov Decision Processes

The most common framework for modeling the interaction between an agent and its environment in RL is the Markov Decision Process (MDP). An MDP models the dynamics with the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where:

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the action space.
- $\mathcal{P} : (\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \mapsto [0, 1]$ is the state transition probability function. Specifically, $\mathcal{P}(s'|s, a)$ is the probability of transitioning to state s' from state s and taking action a.
- $\mathcal{R} : (\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \mapsto \mathbb{R}$ is the reward function, which returns the immediate reward upon transitioning from state s to s' after taking action a.
- $\gamma \in [0, 1]$ is a discount factor that balances the importance of immediate and future rewards.

Additionally, MDPs have the Markov property, which states that the future state depends only on the present state and action. This assumption simplifies the modeling since the algorithm can ignore irrelevant details from the past.

2.3.3 Policy and Value Functions

The policy, denoted by π, is the RL agent's strategy to determine which action to take based on the current state. Formally, a policy is the mapping from states to probabilities of selecting each possible action. Policies can be either deterministic or stochastic. Deterministic policies select a single action with certainty $\pi(s) = a$, whereas stochastic policies form a distribution over actions $a \sim \pi(a|s)$ from which an action is sampled.

Value functions, on the other hand, estimate how good a given state or action is conditioned on following a policy π in terms of expected future rewards. Value functions are broken down into two types: state-value functions $V^\pi(s)$ and action-value functions $Q^\pi(s,a)$.

The state-value function $V^\pi(s)$ is the expected return when starting from state s and following policy π until termination:

$$V^\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^t r_{t+k+1} \middle| S_t = s\right] \tag{2.12}$$

where G_t is the return following time step t. If the environment is not episodic, e.g., $T = \infty$, then the discount factor must satisfy < 1 for convergence.

The action-value function $Q^\pi(s,a)$ is the expected return when starting in state s, taking action a, and following policy π:

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]. \tag{2.13}$$

2.3.4 Advantage Function

The advantage function measures how much better action a is compared to the average action from state s.

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{2.14}$$

2.4 Policy Gradient Methods

Policy gradient methods optimize a parameterized policy directly. Let θ represent the parameters of policy $\pi(a|s;\theta)$. Policy gradient methods aim to find θ that maximizes the expected return from the start distribution. To evaluate the approximate value of a policy, we define $J(\theta)$ as the expected value of the sum of rewards:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\sum_t \mathcal{R}(s_t, a_t)\right] \tag{2.15}$$

where $\tau = s_1, a_1, \ldots, s_T, a_T$ is a trajectory sampled from the trajectory distribution

$$p_\theta = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t). \tag{2.16}$$

We can now estimate $J(\theta)$ by sampling trajectories from policy π_θ:

$$J(\theta) \approx \frac{1}{N} \sum_i \sum_t \mathcal{R}(s_{i,t}, a_{i,t}) \tag{2.17}$$

where $s_{i,t}$ and $a_{i,t}$ are from the ith sample at time t. To improve our policy π_θ, we compute the derivative of $J(\theta)$ and update θ. We refer to [13] for derivation details:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t, s_t) \left[\sum_t^T \mathcal{R}(s_t, a_t)\right]\right] \tag{2.18}$$

2.4.1 REINFORCE

The REINFORCE algorithm is a simple policy gradient algorithm that adjusts the policy parameters in the direction of increasing expected return [14]. It samples trajectories τ under policy π_θ and uses Equation 2.18 to update parameters θ using gradient ascent.

Algorithm 1 REINFORCE

```
1: for each episode do
2:     Generate an episode τ following π_θ
3:     for each step of the episode t = 1, ..., T do
4:         G_t ← return from step t
5:         θ ← θ + η∇_θ J(θ)
6:     end for
7: end for
```

REINFORCE suffers from high variance due to the randomness of sampling trajectories. The return G_t is influenced by subsequent actions and returns, resulting in different returns even from the same state. One way to improve REINFORCE is using a baseline. If V_π is subtracted from the expected return G_t, the return is effectively normalized to reflect how much better or worse the actual return is compared to what was expected (i.e., the baseline). This reduces variance in the returns unrelated to the action, consequently reducing variance in the gradient estimate $\nabla_\theta J(\theta)$

2.4.2 Proximal Policy Optimization (PPO)

Schulman et al. introduced the policy gradient algorithm PPO in 2017 [15], which has proven to be robust and efficient [16]. In the REINFORCE algorithm, the policy updates are made from a single trajectory τ, resulting in noisy and sample-inefficient learning. PPO addresses noisy learning by training on a batch of sampled trajectories. Updating parameters after a batch results in a more accurate estimate of the policy gradient, which reduces variance and allows parallel sampling of trajectories. Additionally, the objective function in PPO is clipped, which reduces how much the policy can improve compared, which makes training more stable.

The PPO objective function is designed to penalize changes in the policy that deviate too far from the current policy:

$$J_{\text{clip}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{2.19}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}(a_t|s_t)}} \tag{2.20}$$

where $\hat{A}_t$ an estimate of the advantage function, ϵ is a hyperparameter that controls the size of policy change, and $r_t(\theta)$ is the probability ratio between the old and new policy.

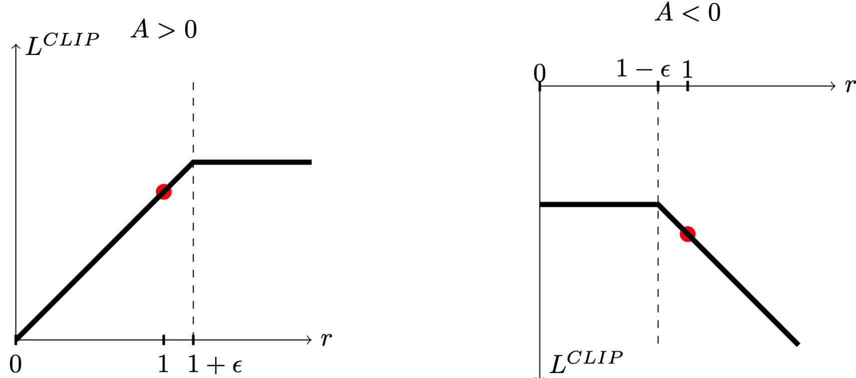

Figure 2.5: Illustration of clipping of the objective for (left) positive advantages and (right) negative advantages [15].

The probability ratio is clipped to ensure smaller changes, and is showcased in Figure 2.5.

2.5 Actor-Critic Architecture

PPO is implemented using an actor-critic architecture. The architecture consists of two components, the actor and critic. The actor is a neural network that represents the policy π_θ, and the critic is a neural network that evaluates the actions picked by the actor. The critic offers feedback to the actor's decisions which are used to during policy updates. Actor-critic architecture uses an advantage function where the critic estimates the advantage A^π providing an estimate of how good an action is in comparison to the average. The actor then uses the advantage to improve its policy, making positive actions (which are better than average) more probable and negative actions less probable.

2.6 Action Masking

Each state has a different set of valid actions. To conform to the RL framework, the action spaces are combined into a single action space [17]. This allows applying RL algorithms but creates the problem of sampling invalid actions. An action sampled from the entire discrete action distribution will likely be invalid. For example, the agent that Vinyals et al. [18] created to play StarCraft II has approximately 10^{26} choices at each time step, but only a small subset of which is valid. In our simulator, there are 220 possible actions for each step, but most are invalid (e.g., the swap does not result in a match or one block is not moveable).

Invalid action masking is a method that masks out invalid actions in the output (last) layer by setting the logits to $-\infty$ (or a large negative number in practice). Given a masking function $\mathcal{M} : \mathbb{R} \mapsto \mathbb{R}$ that masks with $-\infty$ if invalid and leaves argument unchanged otherwise, we can calculate the re-normalized policy π'_θ as follows [19]:

$$\begin{aligned}
\pi'_\theta(\cdot|s_0) &= \text{softmax}(\mathcal{M}([l_0, l_1, l_2])) && (2.21)\\
&= \text{softmax}([l_0, -10^{10}, l_2]) && (2.22)\\
&= [\pi'_\theta(a_0|s_0), \lim_{x \to 0} x, \pi'_\theta(a_2|s_0)] && (2.23)\\
&= [0.5, 0, 0.5]. && (2.24)
\end{aligned}$$

After re-normalization during the sampling phase, the invalid action policy π'_θ is substituted into the objective function. This results in the policy learning to avoid invalid actions. Further, substituting π'_θ into an objective function results in a valid gradient update. The masking function $\mathcal{M}$ outputs either a large negative constant or the original value, both of which are differentiable for all actions and states.

2.7 Evaluation Metrics

We use two metrics to evaluate agents playing match-3 games. The first metric is the average length of trajectories of a given level:

$$\text{average trajectory length} = \frac{\sum_{\tau_i}^{N} \text{length}(\tau_i)}{N} \tag{2.25}$$

The second metric is the win rate, which uses the win cutoff set by the level designer:

$$\text{win rate} = \frac{\sum_{\tau_i}^{N} \text{IsWin}(\tau_i)}{N} \tag{2.26}$$

where IsWin is a simple indicator function.

3

Implementation

In this chapter, we present a detailed overview of the simulator and how the agent interacts with it.

3.1 Simulator

3.1.1 Simulator with Best of Both Worlds: C++ and Python

A feature-rich match-3 simulator is developed without unnecessary components such as audio and animations. Taking inspiration from Lanctot et al. [20], the simulator is written in C++ with an exposed API to Python through the pybind11 project[1]. The architecture of the simulator is optimized for speed. The API is exposed to Python for ease of use. This setup combines the speed of C++ and the ease of use and vast machine learning ecosystem of Python. The setup is illustrated in Figure 3.1

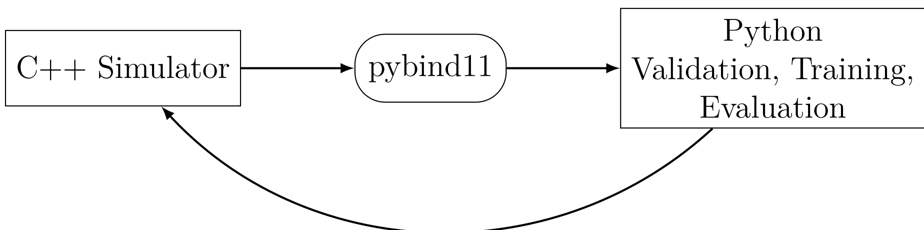

Figure 3.1: Overview of setup and how the simulator communicates with Python.

3.1.2 Reproducibility

To ensure the reproducibility of trajectories, the simulator uses a pseudorandom number generator (PRNG) with a seed. Specifically, `mt19937` is used for every action in which the simulator uses randomness. This makes trajectories deterministic, which is particularly useful for identifying and debugging issues during training and testing.

[1] `https://github.com/pybind/pybind11`

3.1.3 Interoperability between the Simulator and Existing Match-3 Games

Match-3 games typically store level data in JSON files. Each file can store various properties such as game grid dimensions, level objects, and their position, objectives, and swap limit. To load a level into the simulator, the level file(s) are read into Python, which calls the simulator API to set up every object.

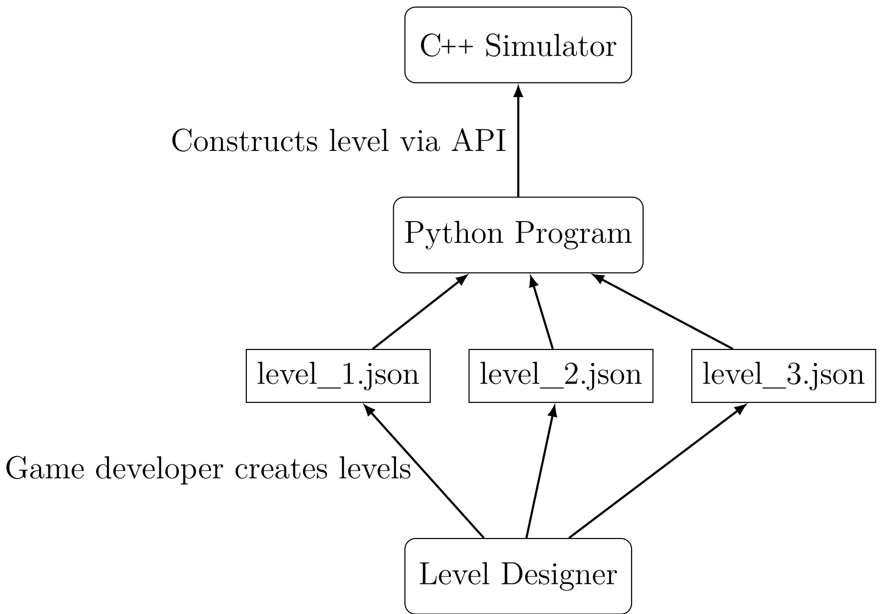

Figure 3.2: Overview of how levels are loaded into the simulator.

3.2 Environment

The environment in a match 3 game consists of a $m \times n$ grid of objects. As described earlier in chapter 1, the core game mechanic is to swap adjacent objects to create matches. When a match occurs, damage is applied to any adjacent obstacles, and the matched objects are cleared, allowing new objects to fall from the top. These matches can also result in the creation of powerup objects which have special effects when activated.

3.2.1 Visualizing Trajectories

The simulator provides convenient methods for analyzing trajectories through visual inspection by printing to standard output using emojis to represent game objects. In addition, ANSI escape codes are used to highlight specific actions on the board, such as which objects are being swapped, by adding a white color background. This visualization method provides a concise representation of the game state and actions, which is useful for inspecting agent behavior and verifying that the simulator works correctly.

Printing the game to standard output provides simplicity and ease of use. Compared to writing a GUI program, which can be complex, printing to standard output require no external dependencies and is portable across more platforms and operating systems. The benefit of portability makes it easy to integrate with existing machine learning pipelines. The simulator's output can also be piped into a file for comparison.

The simulator was used across three major operating systems: macOS Ventura, Ubuntu 20.04, and Windows 10. Any terminal supporting UTF-8 encoding is compatible.

3.2.2 Color Blocks

The simulator supports six color blocks, but only four or five are used in the levels. Typically, only four color blocks are used, but five may be used to increase the difficulty of creating powerups. Color blocks and their visual representation are shown in Table 3.1.

Name	**Representation**
Color 1	🔴
Color 2	🟢
Color 3	🔵
Color 4	🟠
Color 5	🟡
Color 6	🟣

Table 3.1: Visualization of color blocks.

3.2.3 Powerups

Powerups can strategically manipulate the game board and complete objectives faster. These powerups can be generated by matching blocks in specific patterns such as L or T shapes. The powerups in the simulator are showcased in Table 3.2. The match patterns are checked up to rotations and and reflections across the game grid.

A match-3 game typically supports activation of a powerup by either (1) clicking on it, (2) swapping it with any adjacent moveable tile, or (3) indirectly through the action of another powerup. We observed that activation by clicking is not substantially different, which made us omit support for it in the simulator. This allowed simplifying the action space.

Name	Icon	Match Pattern	Effect
Horizontal Blaster	↔️	●●●●	Applies damage to a row
Vertical Blaster	↕️	● ● ● ●	Applies damage to a column
TNT	🧨	●●● ● ●	Applies damage to surroundings within a 2-block radius.
Helicopter	🚁	●● ●●	Applies damage to four surrounding blocks and a random objective

Table 3.2: Visualization of powerups blocks.

3.2.4 Powerup Combinations

Combining adjacent powerups yields even stronger effects which provides players additional strategic options. Levels can be designed that require using powerup combinations to succeed which forces players to think strategically. Powerup combinations and their effects are shown in Table 3.3.

Combination	Effect
(↔️/↕️) + (↔️/↕️)	Applies damage to row+column
🚁 + 🚁	Launches 3x 🚁
🧨 + 🧨	Applies damage to surroundings within a 4-block radius.
🧨 + ↔️/↕️	Launches 3x ↔️ and 3x ↕️
🚁 + (↔️/↕️/🧨)	Helicopter transports and activates the second powerup at a random objective location

Table 3.3: Visualization of powerups combinations.

3.2.5 Obstacles

Players have to clear obstacles to complete a level. A blocker with 1-9 health points (HP) is the most basic obstacle. A blocker can be damaged in two ways. They can (1) take soft damage, which is surrounding damage from a match, or (2) hard damage from a powerup. Obstacles can be configured to take any combination of these damage modes.

Additionally, there are eggs, hearts, mailboxes, and grass. In contrast to crates, eggs and hearts can be swapped and are subject to gravity. Eggs and hearts have one and two health points, respectively. Mailboxes are obstacles that have to be hit with damage a specified number of times, after which the object remains on the board but has no further effect. Another type of obstacle is grass, which is specified on a

layer underneath the game board. Grass has 1-2 HP and can only be cleared by a direct hit to the grid position (e.g., no adjacent damage).

Name	**Visualization**	**Static**	**Takes Soft Damage**	**Health Points**
Egg	🥚	No	Yes	1
Heart	❤️💔	No	Yes	1-2
Blocker	🚧	Yes	Yes	1-9
Mailbox	📫	Yes	Yes	Z^+
Grass	🟩	Yes	No	1-2

Table 3.4: Visualization of obstacles.

3.2.6 Objectives

The goal in a level is to clear a set of objectives. Let $\mathcal{O}$ be the objective space which consists of tuples (c, t) where $c \in \mathbb{N}$ represents the count and $t \in \mathcal{T}$ represents the objective event type. For example, if a level requires a player to remove 10 "Eggs", then the objective for the level can be represented by the tuple $(10, "Egg")$ in the objective space $\mathcal{O}$.

3.2.7 Gravity and Cascading

Gravity moves tiles downwards to fill the empty spaces when a match is made or a powerup wipes objects. Gravity in the game also supports diagonal falling. If the space below a tile is occupied, but there is space available diagonally below, the tile will move there.

Additionally, a check for further matches is made once the board settles due to gravity. This step is called cascading and results in more action on the board. Powerups are commonly created as a result which helps the player.

3.2.8 Swap Limit

A key parameter that level designers consider is the swap limit. Each level defines a swap limit, the maximum number of swaps for a given level. Game-level designers target around 25 swaps for every level but adjust it up or down to ensure players are engaged.

3.3 State Space

Since the features are categorical, the simulator uses a one-hot encoding scheme to represent the state space. Because the game supports level grids up to 11×11, the

observation channels are also 11×11. However, the levels usually have varying grid dimensions. To allow the agent to play any level, the observations are unified and padded with zeros so they are all 11×11.

Another consideration is the number of channels used to represent the state space. A concise representation of state space can improve performance as it simplifies learning [13]. However, reducing the state space can also result in losing important information, leading to worse performance. Thus, a balance has to be struck. We take inspiration from Kristensen et al. [21] by representing state space using layers corresponding to attributes. This approach makes observations more dense and allows better generalization. Furthermore, instead of introducing a new channel for every new obstacle, which the agent would have to learn from scratch, the agent can take advantage of its prior training on the common attributes.

Observation Layer	**Objects**
0	Random color
1	Random color
2	Random color
3	Random color
4	Random color
5	Random color
6	↔️
7	↕️
8	🧨
9	🚁
10	↔️ OR ↕️ OR 🧨 OR 🚁
11	🟩
12	🥚 OR 💔 OR 🚧 OR 📭

Table 3.5: Visualization of observation channels.

Additionally, the channels for color blocks are shuffled to prevent overfitting and allow the agent to generalize across levels that use different color-block subsets. The shuffling uses `std::shuffle` along with the `mt19937` PRNG described in subsection 3.1.2 to ensure determinism.

3.4 Action Space

There are 220 possible swaps in an 11×11 game grid. The action space is, therefore, the discrete integer space $[1, ..., 220]$, where the swaps are enumerated by all the horizontal swaps (proceeding from left to right, top to bottom) followed by all the vertical swaps (also progressing left to right, top to bottom).

Proposition 1. *Total swaps* $= m(n-1) + n(m-1)$, *where* n *and* m *are width and height, respectively.*

A $m \times n$ grid can be modeled as an undirected graph. Let each cell correspond to a vertex, and let edges connect adjacent cells (horizontal or vertical neighbors). The total number of swaps is thus the sum of all horizontal and vertical edges. There are $n-1$ edges in the first row, and m rows in total, hence $m(n-1)$ horizontal edges. Similarly, there are $m-1$ vertical edges in each column, and with n columns in total, this results in $n(m-1)$ vertical edges. The total number of edges (swaps) is therefore given by $m(n-1) + n(m-1)$.

The board can reach a state where there are no legal swaps available. The game enforces an invariant that asserts that there must always be at least one legal move at any given state. To implement this invariant in the simulator, a check for legal moves is made after the gravity and cascade loop. If no moves are available, the board is shuffled as follows until a move is available. Five coordinates (x, y) where $x \sim \text{Unif}\{\text{width}\}$ and $y \sim \text{Unif}\{\text{height}\}$ are sampled, and if there is a color block on the sampled coordinate, replace it with a new sampled color block from the uniform distribution of colors specified in the level. The shuffling process is repeated until at least one move becomes available.

3.5 Reward Function

The design of the reward function can significantly affect learning. There are two main classes of reward functions, sparse and dense. Silver et al [6] successfully used a sparse reward function that rewarded either $+1$ or -1 at the end of each episode. With a large amount of self-play, an agent can build up a value function that predicts probabilities of intermediate states despite not receiving any immediate rewards. However, the agent only receives a signal at the end of an episode, requiring a large amount of sampling to converge.

Dense reward functions can enable faster and more stable learning. By getting immediate feedback, the agent can adapt to the environment faster. We use the following reward function:

1. An action is performed in the game and the reward, $+1$ for every single damage hit to obstacles, is returned. This reward is normalized by dividing it by the maximum possible score.
2. A penalty proportional to the reciprocal number of allowed swaps is subtracted from the reward.
3. If the game level is solved, a bonus of 10 and an additional bonus proportional to how fast the level is solved is added to encourage the agent to solve the level as quickly as possible:

$$\text{bonus} = 10 + \left(1 - \frac{\text{steps}}{\text{max_swaps}}\right) \tag{3.1}$$

4. If the agent does not complete the level within the specified number of steps, the reward is set to -5.

3.6 PPO Agent

```
(pi_features_extractor): CustomCNN(
  (cnn): Sequential(
    (0): Conv2d(13, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(1, 1), stride=(1, 1))
    (3): ReLU()
    (4): Flatten(start_dim=1, end_dim=-1))
  (linear): Sequential(
    (0): Linear(in_features=5184, out_features=128, bias=True)
    (1): ReLU()))
(vf_features_extractor): CustomCNN(
  (cnn): Sequential(
    (0): Conv2d(13, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(1, 1), stride=(1, 1))
    (3): ReLU()
    (4): Flatten(start_dim=1, end_dim=-1))
  (linear): Sequential(
    (0): Linear(in_features=5184, out_features=128, bias=True)
    (1): ReLU()))
(mlp_extractor): MlpExtractor(
  (policy_net): Sequential(
    (0): Linear(in_features=128, out_features=64, bias=True)
    (1): Tanh()
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): Tanh())
  (value_net): Sequential(
    (0): Linear(in_features=128, out_features=64, bias=True)
    (1): Tanh()
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): Tanh()))
(action_net): Linear(in_features=64, out_features=220, bias=True)
(value_net): Linear(in_features=64, out_features=1, bias=True)
```

Figure 3.3: PPO network architecture

The PPO algorithm is used for training the agents due to its sample efficiency, stability, and wide adaptation. The model network architecture used in experiments is shown in Figure 3.3. The network consists of the actor and critic networks that share the same architecture but not the weights. Both the actor and critic use a CNN feature extractor. Further, the following parameters are used:

- Learning rate: 3e-4
- Batch size: 64

- γ: 0.99
- clip range: 0.2

3.7 Hardware

A PC running Ubuntu 20.04 with the following hardware is used for training:

1. GPU: Nvidia RTX 3080
2. CPU: AMD Ryzen 5600X
3. RAM: 16GB

4

Results

This chapter presents the results from training PPO models on individual levels and subsets. Models are evaluated on unseen levels to determine generalization ability. Finally, the models are compared against a random agent.

4.1 Experiments

4.1.1 Experimental Setup

To investigate if PPO can solve match-3 levels, a variety of models are trained and compared. The models are evaluated on 29 levels with a diverse set of board layouts (from 7×7 to 11×11) and obstacles. Additionally, each level is seeded with random number to prevent overfitting. The first experiment trains a model repeatedly to establish stability of training on a single model. A model is stable if it can repeatedly converge. Next, models are trained on individual levels to determine if they can master individual levels. Following this, training is done on a subset of levels with two different strategies: sequential and random sampling during training.

4.1.2 Stability of Training

Model M_6 is trained on a single level repeatedly. Figure 4.1 shows the min, max, and average trajectory lengths on level 6, suggesting stable reproducibility under repeated training. Three models were trained to 1M steps and all three models converged.

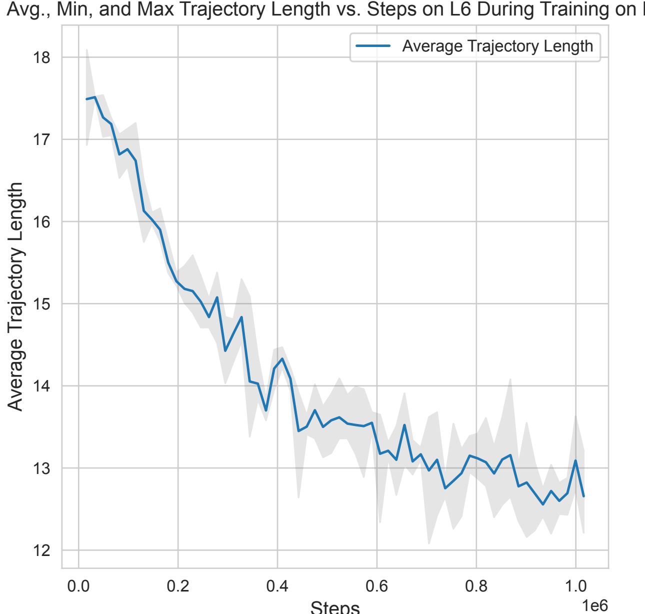

Figure 4.1: This plot illustrates the stability and reproducibility of training of training on a single level.

4.1.3 Training on Single Levels

Four models are trained on 1M steps on individual levels and are compared against each other. Figure 4.2 and Figure 4.3 show the average trajectory length and success rate, respectively. Figure 4.4 compares the models, with each subgraph highlighting the corresponding training level.

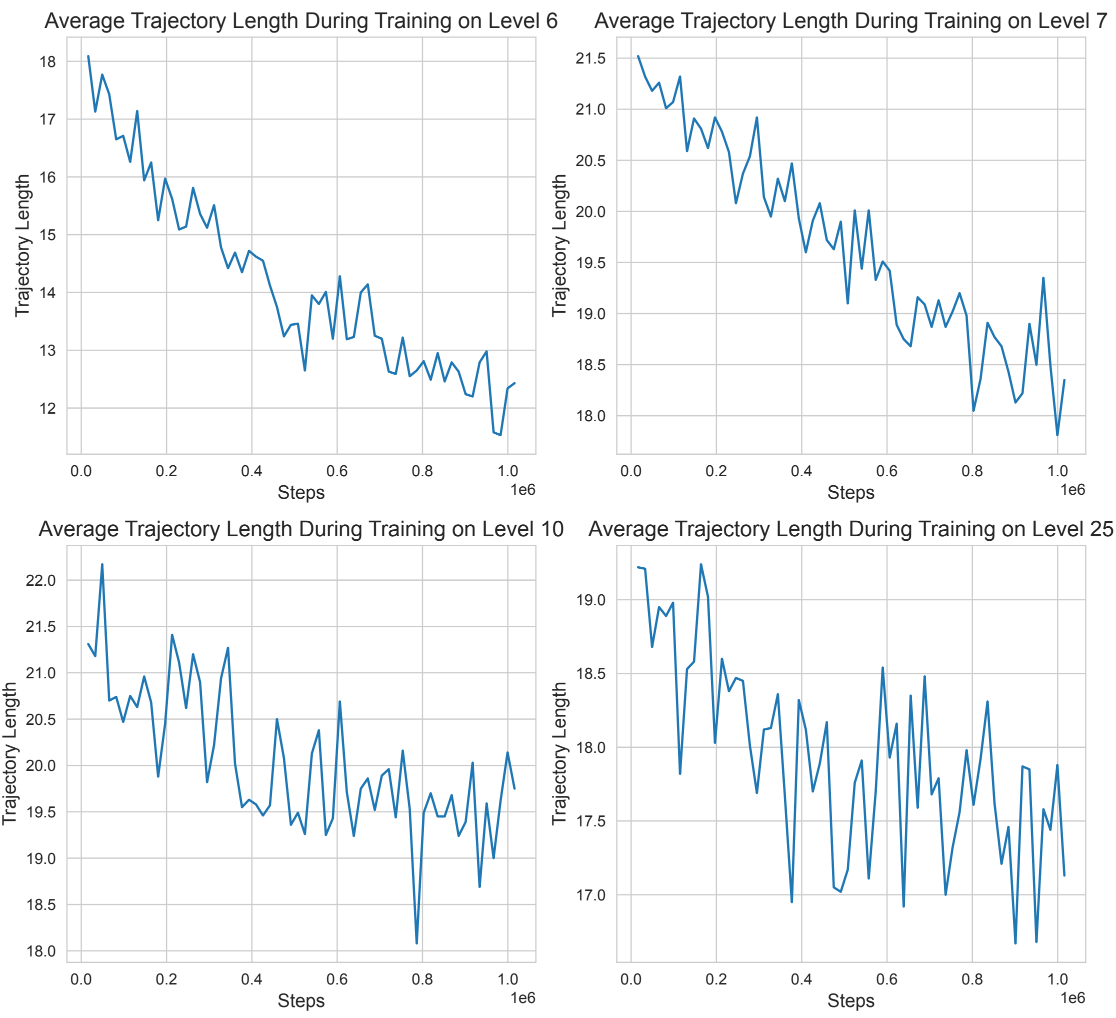

Figure 4.2: Four distinct models are trained independently on individual levels. The average trajectory length of each model is evaluated on its respective training level. Top-left: L6. Top-right: L7. Bottom-left: L10. Bottom-right: L25.

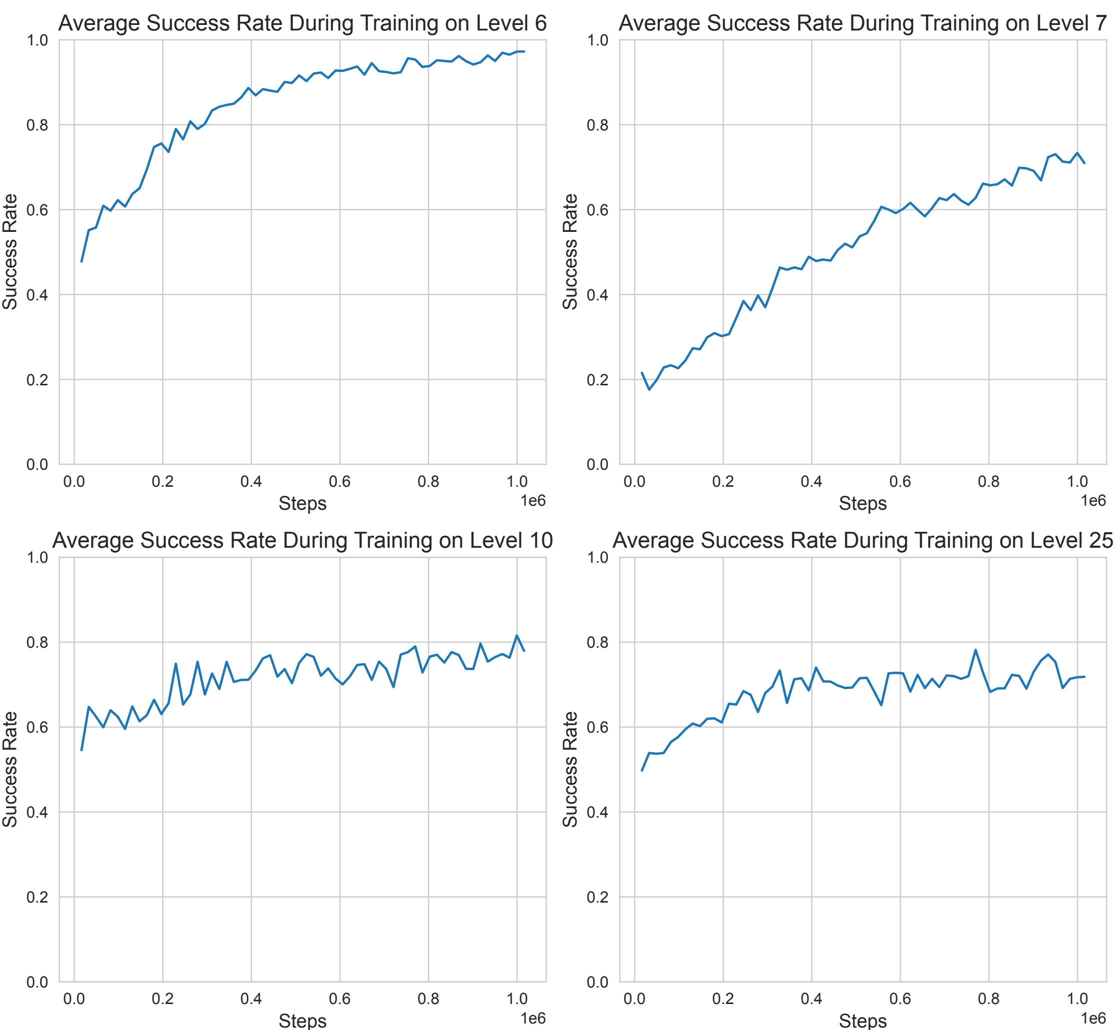

Figure 4.3: Four distinct models are trained independently on individual levels. The average success rate of each model is evaluated on its respective training level. Top-left: L6. Top-right: L7. Bottom-left: L10. Bottom-right: L25.

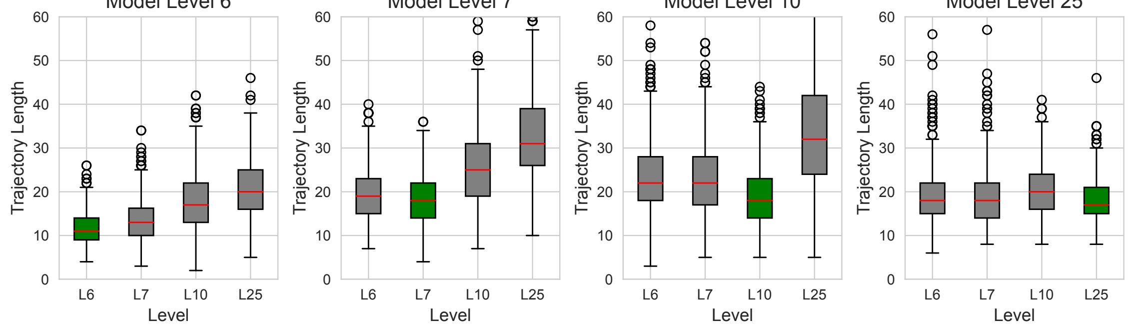

Figure 4.4: Comparison of average trajectory length between individually trained models. Top-left: L6. Top-right: L7. Bottom-left: L10. Bottom-right: L25.

4.1.4 Training on Subset {6,7,10,25} Sequentially

The model $M_{6,7,10,25}$ is trained 1M steps on levels 6,7,10,25 sequentially, for a total of 4M steps.

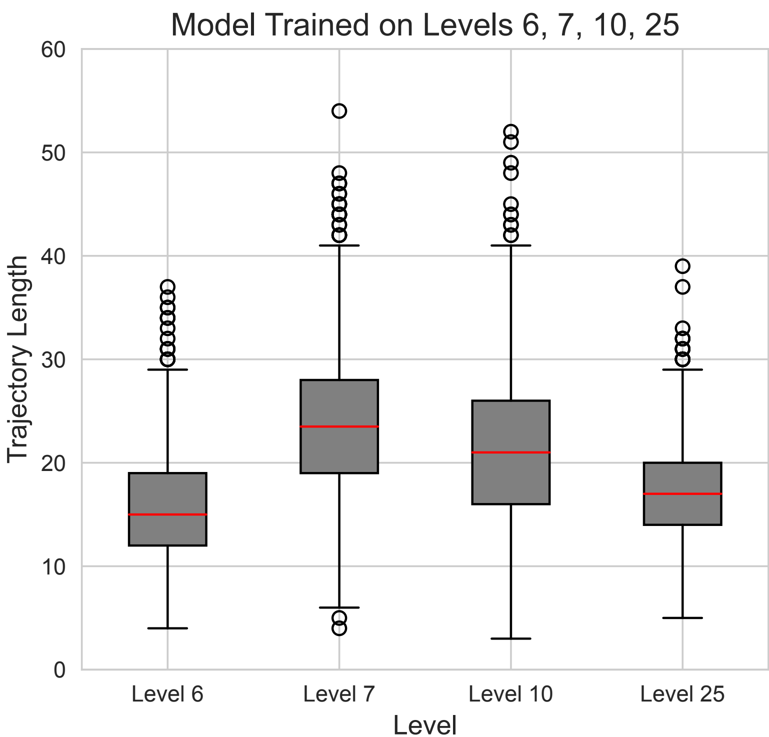

Figure 4.5: Model $M_{6,7,10,25}$ is evaluated on its training levels 6,7,10,25.

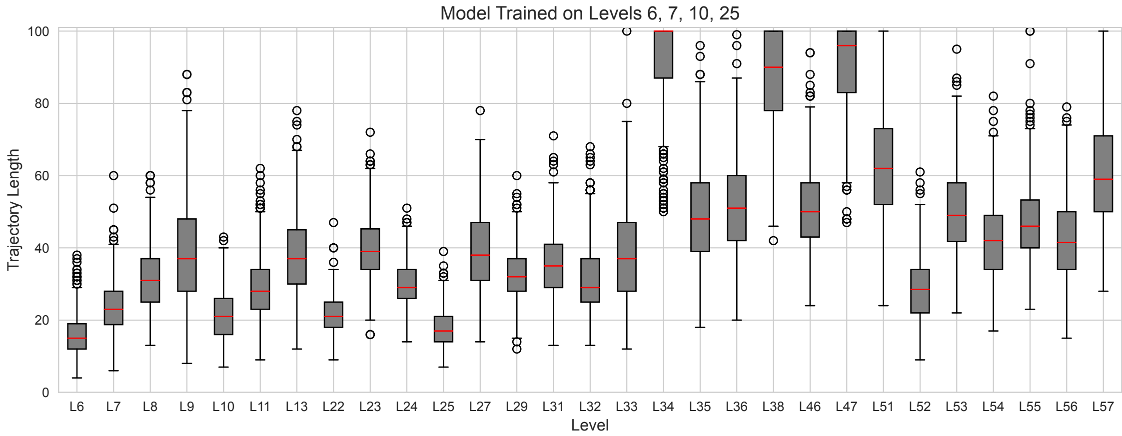

Figure 4.6: Model $M_{6,7,10,25}$ is evaluated on all levels.

4.1.5 Training on Subset {27,29,47,51} Sequentially

Similarly, model $M_{27,29,47,51}$ is trained 1M steps on levels 27,29,47,51 for a total of 4M steps.

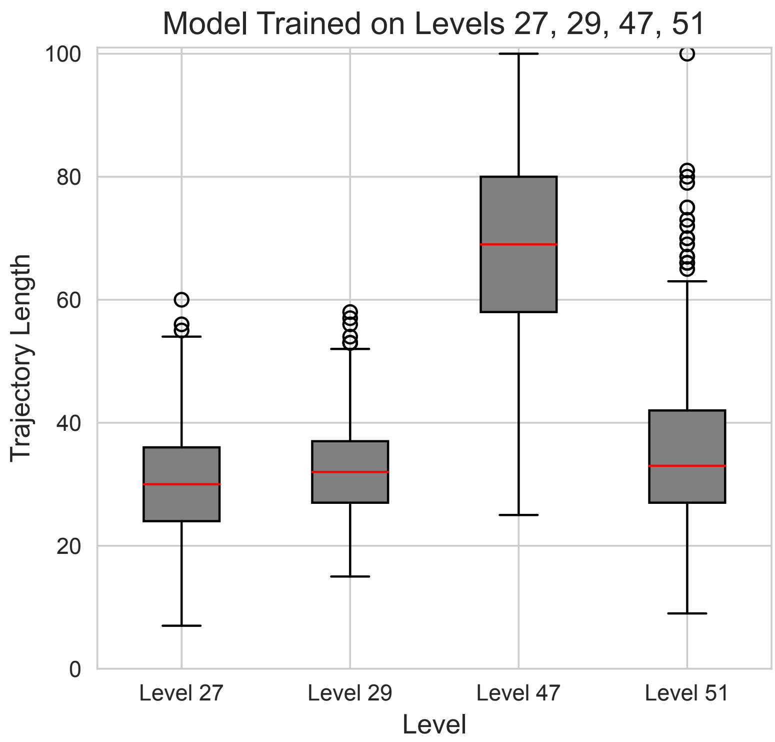

Figure 4.7: Model $M_{27,29,47,51}$ is evaluated on its training levels 27,29,47,51.

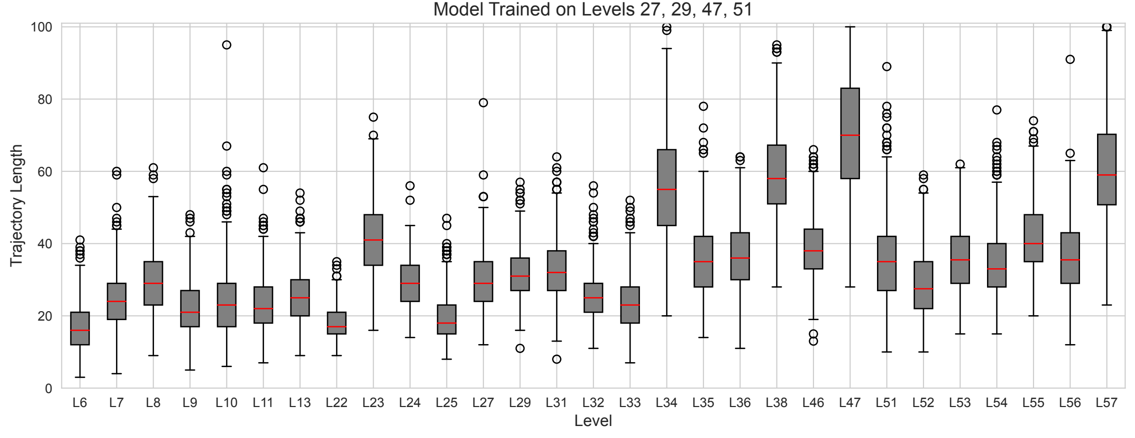

Figure 4.8: Model $M_{27,29,47,51}$ is evaluated on all levels.

4.1.6 Training on Subset {6,7,10,25} with Uniform Sampling

Model $M_{\sim 6,7,10,25}$ is trained by sampling a new level uniformly from 6,7,10,25 for each new trajectory during training.

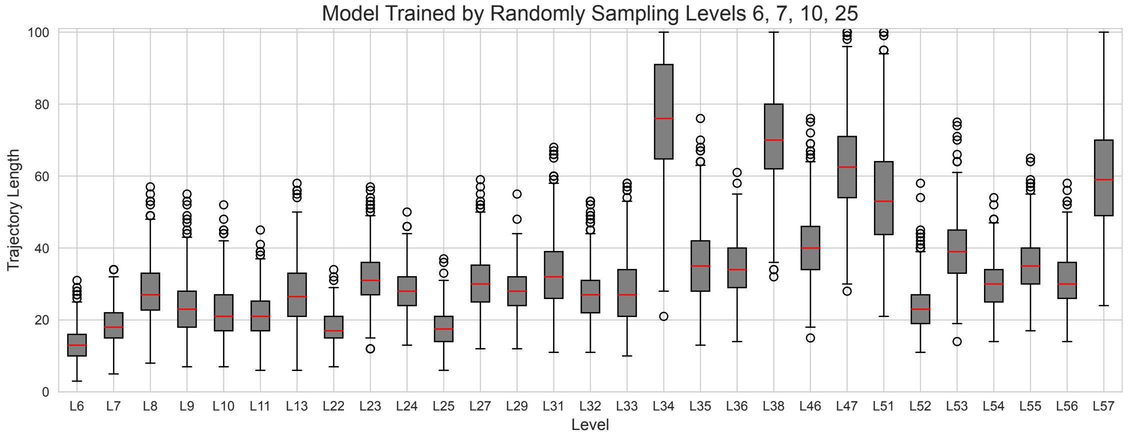

Figure 4.9: Model $M_{\sim 6,7,10,25}$ is evaluated on all levels.

4.1.7 Training on Subset {27,29,47,51} with Uniform Sampling

Model $M_{\sim 27,29,47,51}$ is trained by sampling a new level uniformly from 27,29,47,51 for each new trajectory during training.

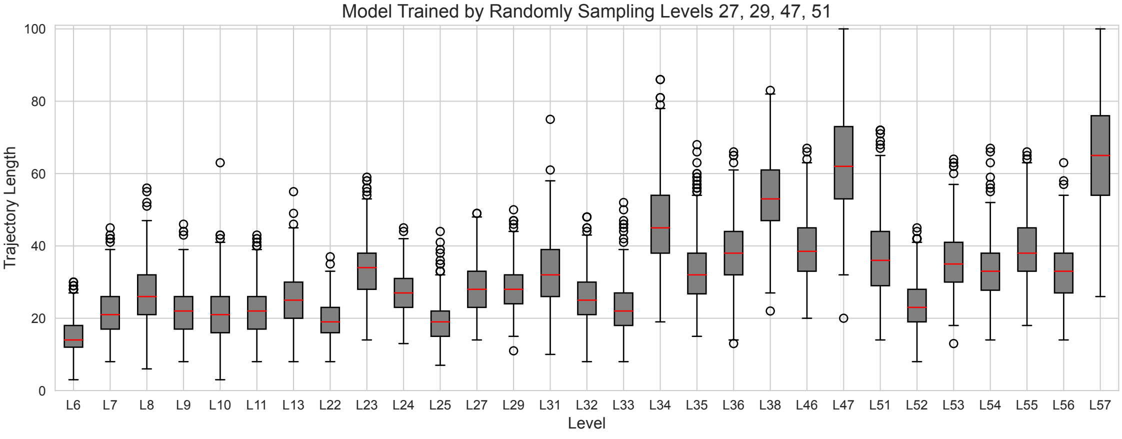

Figure 4.10: Model $M_{\sim 27,29,47,51}$ is evaluated on all levels.

4.1.8 Training on All Levels with Uniform Sampling

Model M_{all} is trained by sampling a level from all available levels for each new trajectory during training.

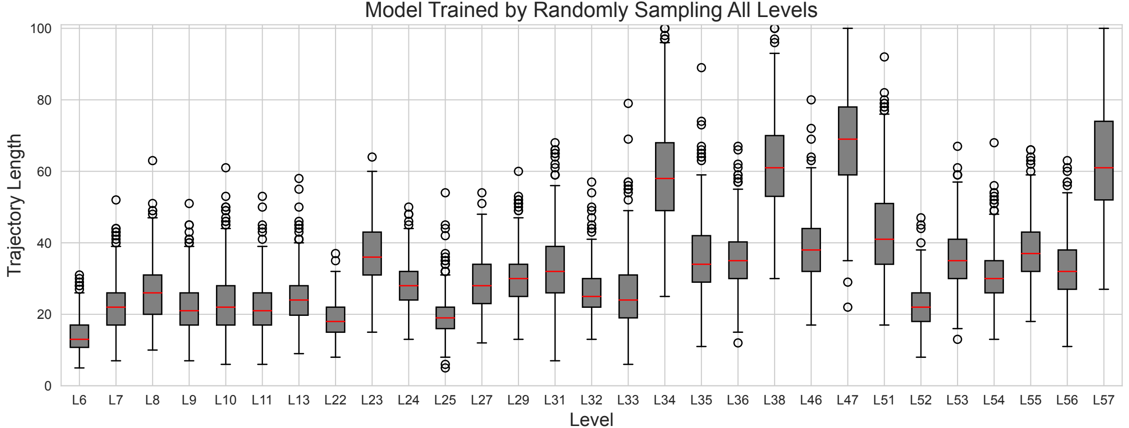

Figure 4.11: Model $M_{\sim\text{all}}$ is evaluated on all levels.

4.1.9 Benchmark Comparison

Table 4.1 presents the average trajectory length of all models across all levels. Table 4.2 presents the percentage difference in performance to a random agent.

	M_6	$M_{6,7,10,25}$	$M_{27,29,47,51}$	$M_{\sim 6,7,10,25}$	$M_{\sim 27,29,47,51}$	$M_{\sim all}$	$M_{random\ agent}$
L6	**11.7**	15.8	16.7	13.6	15.0	14.1	23.5
L7	20.0	23.7	24.5	**18.6**	21.8	22.4	35.0
L8	31.3	32.1	29.0	28.2	26.6	**25.9**	37.9
L9	33.9	38.3	**21.9**	23.6	**21.9**	22.0	28.7
L10	23.3	21.7	24.3	22.2	**21.2**	23.3	26.4
L11	22.3	28.8	23.4	**21.5**	22.3	21.9	37.4
L13	29.7	37.4	25.5	27.3	25.5	**24.5**	38.4
L22	19.7	21.5	18.0	**17.7**	19.7	18.7	32.1
L23	**30.1**	40.0	41.8	31.7	33.7	36.6	70.2
L24	27.7	29.7	29.2	28.1	**27.0**	28.5	32.2
L25	19.6	**17.7**	19.7	**17.7**	19.2	19.6	23.2
L27	35.1	39.5	29.9	30.6	**28.1**	28.5	42.3
L29	**28.0**	32.6	32.1	28.4	28.1	30.4	41.8
L31	**31.6**	35.4	32.6	32.7	33.1	33.3	47.5
L32	28.4	31.5	**25.7**	27.4	**25.7**	26.2	37.4
L33	36.1	38.2	24.1	28.5	**22.8**	25.1	28.4
L34	92.2	92.6	55.9	76.2	**46.6**	59.6	68.0
L35	43.5	49.1	36.0	35.9	**33.5**	35.7	49.2
L36	37.8	51.3	36.9	**34.7**	38.1	35.4	53.7
L38	84.7	86.9	59.3	71.0	**54.2**	62.0	80.9
L46	40.3	50.6	38.9	40.5	39.3	**38.4**	49.2
L47	83.5	90.2	70.4	63.8	**63.1**	69.3	89.1
L51	57.6	62.6	**36.2**	54.7	37.6	43.1	59.5
L52	**22.0**	28.9	28.9	23.7	23.9	22.5	36.5
L53	45.3	49.7	**35.8**	39.2	36.0	35.9	51.7
L54	30.9	42.2	34.7	**30.1**	33.3	30.9	55.9
L55	35.8	47.2	41.6	**35.2**	39.1	37.8	57.7
L56	32.1	42.4	36.2	**31.1**	32.8	32.7	48.6
L57	61.1	60.9	61.0	**60.4**	66.1	63.3	74.9
Avg.	37.8	42.7	34.1	34.3	32.3	33.4	46.8

Table 4.1: Average trajectory lengths of all models and a random agent.

	M_6	$M_{6,7,10,25}$	$M_{27,29,47,51}$	$M_{\sim 6,7,10,25}$	$M_{\sim 27,29,47,51}$	$M_{\sim all}$
L6	-50.2	-32.8	-28.9	-42.1	-36.2	-40.0
L7	-42.9	-32.3	-30.0	-46.9	-37.7	-36.0
L8	-17.4	-15.3	-23.5	-25.6	-29.8	-31.7
L9	18.1	33.4	-23.7	-17.8	-23.7	-23.3
L10	-11.7	-17.8	-8.0	-15.9	-19.7	-11.7
L11	-40.4	-23.0	-37.4	-42.5	-40.4	-41.4
L13	-22.7	-2.6	-33.6	-28.9	-33.6	-36.2
L22	-38.6	-33.0	-43.9	-44.9	-38.6	-41.7
L23	-57.1	-43.0	-40.5	-54.8	-52.0	-47.9
L24	-14.0	-7.8	-9.3	-12.7	-16.1	-11.5
L25	-15.5	-23.7	-15.1	-23.7	-17.2	-15.5
L27	-17.0	-6.6	-29.3	-27.7	-33.6	-32.6
L29	-33.0	-22.0	-23.2	-32.1	-32.8	-27.3
L31	-33.5	-25.5	-31.4	-31.2	-30.3	-29.9
L32	-24.1	-15.8	-31.3	-26.7	-31.3	-29.9
L33	27.1	34.5	-15.1	0.4	-19.7	-11.6
L34	35.6	36.2	-17.8	12.1	-31.5	-12.4
L35	-11.6	-0.2	-26.8	-27.0	-31.9	-27.4
L36	-29.6	-4.5	-31.3	-35.4	-29.1	-34.1
L38	4.7	7.4	-26.7	-12.2	-33.0	-23.4
L46	-18.1	2.8	-20.9	-17.7	-20.1	-22.0
L47	-6.3	1.2	-21.0	-28.4	-29.2	-22.2
L51	-3.2	5.2	-39.2	-8.1	-36.8	-27.6
L52	-39.7	-20.8	-20.8	-35.1	-34.5	-38.4
L53	-12.4	-3.9	-30.8	-24.2	-30.4	-30.6
L54	-44.7	-24.5	-37.9	-46.2	-40.4	-44.7
L55	-38.0	-18.2	-27.9	-39.0	-32.2	-34.5
L56	-34.0	-12.8	-25.5	-36.0	-32.5	-32.7
L57	-18.4	-18.7	-18.6	-19.4	-11.7	-15.5
Average	-20.3	-9.8	-26.5	-27.2	-30.6	-28.7

Table 4.2: Percentage difference in average trajectory length between all models and random agent.

4.2 Discussion

4.2.1 Training on Single Levels

Model M_6 is trained solely on level 6 and, unsurprisingly, performs best on level 6. Interestingly, model M_6 also beat all other models in levels 23, 29, 31, and 52. Upon inspection of the levels, they share the common characteristic of having a smaller square board size of 7×7. This suggests that the model learned not only to play well on 7×7, but also to generalize to other levels. Level 6 contained only crates and regular blocks, whereas levels 23, 29, 31, and 52 combined included all the remaining obstacles.

4.2.2 Training on Subsets Sequentially

Training on subsets sequentially demonstrated catastrophic forgetting [22], which results in performance degradation on previously trained levels. Models $M_{6,7,10,25}$ and $M_{27,29,47,51}$ mainly performed well on their last trained levels, 25 and 51, respectively, while other models not trained on these subsets performed better. For example, M_6 beats $M_{6,7,10,25}$ by 3.7 swaps on level 7. An alternative approach was explored upon identifying catastrophic forgetting, which samples levels for every trajectory during training.

4.2.3 Training on Subsets with Uniform Sampling

Models trained by sampling from a subset of levels outperformed their sequential counterparts on average from Table 4.1. By constantly rotating through the levels, the models continuously refine the learning from each level resulting in better retention. However, model $M_{\sim \text{all}}$ despite sampling from all levels, did not have superior performance across all levels. A possible explanation for this may be that the agent did not train long enough.

4.2.4 Training Steps

From Figure 4.3 it can be observed that performance starts to plateau around one million steps. Consequently, all models are trained to one million steps. Performance could be improved with longer training durations, but given the time constraints of this thesis, it is left as future work.

4.2.5 Action masking

Action masking allows the agent to converge quickly compared to other methods. Prior to using action masking, experiments were done with assigning negative penalties for invalid actions, and it did not work well. The agent would not converge most of the times.

4.2.6 Problematic Levels

Half of the models performed worse than random policy on levels L33 and L34 from Table 4.2. Upon inspection, these two levels have narrow starting areas with color blocks, with the rest being filled with eggs/hearts. Obstacles like eggs/hearts are subject to gravity, which may result in the obstacles further narrowing down the areas occupied by color blocks at the beginning of an episode. Since our simulator implementation only resamples color-blocks, entering states with no further legal actions is possible. Trajectories that enter states with no legal actions (even after resampling color blocks) terminate after the hard cutoff of 100 steps in our implementation.

There are different ways of dealing with the problem of no available legal moves. As mentioned in chapter 3, our simulator implementation only resamples color blocks. Resampling color blocks is a simple method that works well for most levels. However, resampling only works under the following proposition:

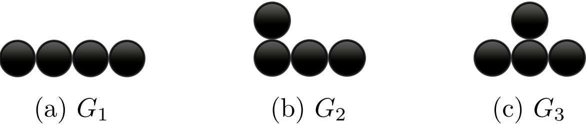

Figure 4.12: Color-block configurations that lead to matches.

Proposition 2. *Let G denote a match-3 grid. Assume only color blocks are subject to gravity and resampling. Let C be the set of all possible configurations of G. For any configuration $c \in C$, a legal move is possible if and only if there exists a color-block sub-configuration s within c that is congruent to G_1, G_2, or G_3 up to rotations, reflections, and translations. Additionally, every block in s must be reachable by gravity from a block spawn point.*

Proof. ($\Rightarrow$) If a move exists, two adjacent blocks can be swapped to form a match of three blocks of the same color. This can only occur in the following cases:

- four color-blocks are aligned,
- three color-blocks are aligned, and a fourth color block extends from the middle,
- three color-blocks are aligned, and a fourth color block extends from one of the ends.

These cases correspond to G_1, G_2, and G_3 up to rotations, reflections, and translations.
($\Leftarrow$) If a color-block sub-configuration congruent to G_1, G_2, G_3 exists, then a move exists. Given any such sub-configuration, there exists a color permutation (reachable by resampling) that results in being able to perform a swap to align three blocks of the same color. Additionally, gravity will cause blocks to fall and replenish empty spaces caused by the removal of matched blocks. $\square$

Proposition 2 can be used to verify the playability of levels, ensuring that there is always a legal move. In practice, the shapes G_1, G_2, and G_3 can be checked for by scanning across the match-3 grid. For simple vertical downwards gravity, each color block must have an uninterrupted (by obstacles) path to a block spawner. For more complex gravity mechanics that, for example, allow diagonal falling, each color-block in the configurations mentioned above must also have an uninterrupted path to a block-spawner. Such a path must exist in the space that starts from the block and grows by two blocks horizontally for every upward step.

Additionally, it can aid in automatic match-3 level generation by verifying levels directly rather than sampling sufficient trajectories to ascertain playability.

4.2.7 Ethical Considerations

Training agents to playtest match-3 games raises the issue of impact on employment. Instead of replacing level designers, such tools accelerate their workflow and improve the overall game. Instead of focusing on repetitive tasks such as manual testing during level creation, the creator can receive instant feedback and focus on the creative aspects of level creation.

5

Conclusion

This thesis explores improving play-testing by building a performant, extensive match-3 game simulator and training PPO agents with action masking to learn and generalize on levels. To answer the original research question, whether it is possible to estimate level difficulty through Reinforcement Learning agents, the results show that RL agents can be trained to estimate difficulty. Agents are trained with different strategies and are compared against each other. The best model was trained on a subset of four diverse levels that were sampled uniformly during training. Action masking allowed all models to converge and perform better than an random agent across all levels (with a few addressed exceptions). Results show that models are able to generalize well on unseen levels with different layouts and obstacles, suggesting that logically ORing obstacles into a single layer works well. While the model performed well, it is difficult to draw conclusions about how the performance relates to humans without real playtesting data.

5.1 Future Work

Comparing agent performance to human play-data could give further insights into how well the agents are performing. Additionally, a comparison against classical hard-coded AI would be useful to determine whether trained agents demonstrate more adaptable gameplay strategies.

Bibliography

[1] L. Gualà, S. Leucci, and E. Natale, *Bejeweled, Candy Crush and other Match-Three Games are (NP-)Hard*, arXiv:1403.5830 [cs], Mar. 2014. DOI: 10.48550/arXiv.1403.5830. [Online]. Available: http://arxiv.org/abs/1403.5830 (visited on 05/30/2023).

[2] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, *Playing Atari with Deep Reinforcement Learning*, arXiv:1312.5602 [cs], Dec. 2013. DOI: 10.48550/arXiv.1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602 (visited on 01/25/2023).

[3] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, *Quantifying Generalization in Reinforcement Learning*, arXiv:1812.02341 [cs, stat], Jul. 2019. [Online]. Available: http://arxiv.org/abs/1812.02341 (visited on 02/09/2023).

[4] E. R. Poromaa, *Crushing Candy Crush : Predicting Human Success Rate in a Mobile Game using Monte-Carlo Tree Search*, eng. 2017. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-206595 (visited on 06/06/2023).

[5] Y. Shin, J. Kim, K. Jin, and Y. B. Kim, "Playtesting in Match 3 Game Using Strategic Plays via Reinforcement Learning," *IEEE Access*, vol. 8, pp. 51 593–51 600, 2020, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2980380.

[6] D. Silver, J. Schrittwieser, K. Simonyan, *et al.*, "Mastering the game of Go without human knowledge," en, *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017, Number: 7676 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/nature24270. [Online]. Available: https://www.nature.com/articles/nature24270 (visited on 05/02/2023).

[7] J. Jumper, R. Evans, A. Pritzel, *et al.*, "Highly accurate protein structure prediction with AlphaFold," en, *Nature*, vol. 596, no. 7873, pp. 583–589, Aug. 2021, Number: 7873 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2. [Online]. Available: https://www.nature.com/articles/s41586-021-03819-2 (visited on 05/30/2023).

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," en, *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, Number: 6088 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/323533a0. [Online]. Available: https://www.nature.com/articles/323533a0 (visited on 05/09/2023).

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information*

Processing Systems, vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (visited on 05/16/2023).

[10] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, *Image-to-Image Translation with Conditional Adversarial Networks*, arXiv:1611.07004 [cs], Nov. 2018. DOI: 10.48550/arXiv.1611.07004. [Online]. Available: http://arxiv.org/abs/1611.07004 (visited on 05/19/2023).

[11] B. F. Skinner, "Are theories of learning necessary?" *Psychological Review*, vol. 57, pp. 193–216, 1950, Place: US Publisher: American Psychological Association, ISSN: 1939-1471. DOI: 10.1037/h0054367.

[12] C. J. C. H. Watkins and P. Dayan, "Q-learning," en, *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992698. [Online]. Available: https://doi.org/10.1007/BF00992698 (visited on 05/07/2023).

[13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, English, second edition. Cambridge, Mass: A Bradford Book, Mar. 1998, ISBN: 978-0-262-19398-6.

[14] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," en, *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992696. [Online]. Available: https://doi.org/10.1007/BF00992696 (visited on 05/20/2023).

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal Policy Optimization Algorithms*, arXiv:1707.06347 [cs], Aug. 2017. DOI: 10.48550/arXiv.1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347 (visited on 05/21/2023).

[16] C. Yu, A. Velu, E. Vinitsky, *et al.*, "The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games," en, *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 611–24 624, Dec. 2022. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/hash/9c1535a02f0ce079433344e14d910597-Abstract-Datasets_and_Benchmarks.html (visited on 05/21/2023).

[17] O. Vinyals, T. Ewalds, S. Bartunov, *et al.*, *StarCraft II: A New Challenge for Reinforcement Learning*, arXiv:1708.04782 [cs], Aug. 2017. [Online]. Available: http://arxiv.org/abs/1708.04782 (visited on 05/21/2023).

[18] O. Vinyals, I. Babuschkin, W. M. Czarnecki, *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," en, *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 2019, Number: 7782 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. [Online]. Available: https://www.nature.com/articles/s41586-019-1724-z (visited on 05/21/2023).

[19] S. Huang and S. Ontañón, "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms," *The International FLAIRS Conference Proceedings*, vol. 35, May 2022, arXiv:2006.14171 [cs, stat], ISSN: 2334-0762. DOI: 10.32473/flairs.v35i.130584. [Online]. Available: http://arxiv.org/abs/2006.14171 (visited on 05/21/2023).

[20] M. Lanctot, E. Lockhart, J.-B. Lespiau, *et al.*, *OpenSpiel: A Framework for Reinforcement Learning in Games*, arXiv:1908.09453 [cs], Sep. 2020. DOI: 10.48550/arXiv.1908.09453. [Online]. Available: http://arxiv.org/abs/1908.09453 (visited on 04/28/2023).

[21] J. T. Kristensen and P. Burelli, "Strategies for Using Proximal Policy Optimization in Mobile Puzzle Games," in *International Conference on the Foundations of Digital Games*, arXiv:2007.01542 [cs], Sep. 2020, pp. 1–10. DOI: 10.1145/3402942.3402944. [Online]. Available: http://arxiv.org/abs/2007.01542 (visited on 05/01/2023).

[22] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, Mar. 2017, Publisher: Proceedings of the National Academy of Sciences. DOI: 10.1073/pnas.1611835114. [Online]. Available: https://www.pnas.org/doi/10.1073/pnas.1611835114 (visited on 06/01/2023).