LEVEL GENERATION USING GENETIC ALGORITHMS AND DIFFICULTY TESTING USING REINFORCEMENT LEARNING IN MATCH-3 GAME

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SAMET ALP DUKKANCI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2021

Approval of the thesis:

LEVEL GENERATION USING GENETIC ALGORITHMS AND DIFFICULTY TESTING USING REINFORCEMENT LEARNING IN MATCH-3 GAME

submitted by **SAMET ALP DUKKANCI** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Ferda Nur Alpaslan
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Mehmet Volkan Atalay
Computer Engineering, METU _____

Prof. Dr. Ferda Nur Alpaslan
Computer Engineering, METU _____

Prof. Dr. İlyas Çiçekli
Computer Engineering, Hacettepe University _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Samet Alp Dukkancı

Signature :

ABSTRACT

LEVEL GENERATION USING GENETIC ALGORITHMS AND DIFFICULTY TESTING USING REINFORCEMENT LEARNING IN MATCH-3 GAME

Dukkancı, Samet Alp
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Ferda Nur Alpaslan

September 2021, 56 pages

The gaming industry is an enormous one that includes game development, art, and marketing. It has grown faster in recent years, especially in the mobile gaming area. Competition in mobile gaming increased in such a way that it brings us some challenges like quick prototyping, automation, minimum viable products, and so on. Level generation is the most important issue in the development period because unique and well-adjusted difficulty for a level is generally tested by humans many times for one level to assure that the level is ready to be added and it is a time consuming process. This thesis aims to come through those issues by the proposed automated level generation for a match-3 game using genetic algorithms and testing all generated levels using reinforcement algorithms to minimize the time consumption for a level designer. This helps game developers easily and quickly generate as many levels as needed.

Keywords: reinforcement learning, genetic algorithms, level design, match-3 games

ÖZ

EŞLEŞTİRME OYUNLARINDA GENETİK ALGORİTMA İLE SEVİYE ÜRETİMİ VE TAKVİYELİ ÖĞRENME İLE SEVİYE ZORLUK BELİRLEMESİ

Dukkancı, Samet Alp
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ferda Nur Alpaslan

Eylül 2021 , 56 sayfa

Oyun endüstrisi zaman geçtikçe büyümüş ve şu an çok büyük bir endüstriye dönüşmüştür. Oyun endüstrinin içeriğinde oyun geliştirme, sanat, ve bu oyunların piyasaya çıkartırılması da dahil olmak üzere çok fazla disiplinlerin bulunduğu bir endüstridir. Mobil oyun sektörü de oyun endüstrinin payı oldukça büyüyen bir parçasıdır. Mobil oyun sektöründe odaklanan konular genel olarak hızlıca bir prototip çıkarmak, otomasyon ve minimum ihtiyaçlar kullanılarak uygulanabilir bir ürün çıkarmaktır. Mobil oyun sektöründe tutunmak için bu bahsedilen işleri her zaman başarılı bir şekilde devam ettirmek ve oluşacak tüm hataları ve zaman ihtiyacı gerektiren işleri minimize etmek gerekmektedir. Oyun üretim hattında zamandan çalan ve doğruluğundan emin olmanın riskli olduğu en problemli alanlardan biri üretilen oyun için oyun seviyeleri oluşturmaktır. Tüm oyun seviyeleri insanların keyif alabileceği bir zorluk seviyesinde ve dizaynında olması gerekmektedir. Bu seviye üretiminin en güvenli yolu bir oyun seviye üreticisinin elinden çıkmasıdır. Seviye üreticisi oyun için gereken seviyeyi kendi tasarlar, belirler ve bir çok kez oynarak test ederek oyunun zorluğunu bu

sistemle belirler. Ancak, bu yöntem çok fazla kaynak kullanmakta ve bu nedenle de çok zaman almaktadır. Bu tezdeki çalışma bu problemleri ortadan kaldırmayı hedeflemektedir. Bu hedefe ulaşmak için de oyun seviyesi üretiminde genetik algoritmalar ve üretilen oyun seviyelerinin zorluk seviyesinin ayarlanmasında da takviyeli öğrenme algoritmaları kullanılmıştır. Böylece zorluğu belirlenmiş oyun seviyesi üretimi çok daha kısa sürede yapılıp daha otomasyon bir hale getirilmiştir.

Anahtar Kelimeler: takviyeli öğrenme, genetik algoritmalar, oyun seviyesi dizaynı, eşleştirme oyunları

To my family and loved ones

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Dr. Ferda Nur Alpaslan for guiding me and learn more with her thoughtful comments and recommendations. Our meetings were vital in inspiring me to think from multiple perspectives to form a comprehensive and objective critique.

I also would like to thank Özgür Alan. He also helped and supported me and give new ideas about some aspects of this study with his guidance by brainstorming questions on reinforcement learning, algorithms as much as my supervisor.

I appreciate my friends Artun, Hazan, İsmail for always giving ideas and motivating me to learn more. I am very grateful to have friends I can depend on. I am glad we had each other and could go through this experience together. I also would like to thank my friends, Cansu, Ozan, Volkan, and Serkan who always stood by me and supported me to overcome all my difficulties.

I could not go on without my cousins Merve, Özge, and Zuhal providing me so much love and their support. Even though we can see each other for a long time, our phone calls made me feel I am loved. Our love and friendship will be forever.

I am thankful to my family for always being by my side and comforting me while I was going through tough times. I couldn't do anything without them. They made me believe in myself and that I could succeed. When I need them, they lived with me and always offered help no matter the subject. Our family supports us in our worst times and celebrates in our bests; everything we are is because of them. I have a wonderful shelter, which is my family. I have a wonderful relationship with my sister Esra and her husband Kadir; this makes me feel that I'm lucky enough that I will love and be loved for the rest of my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

NPC	Non-player Character
GA	Genetic Algorithm
FSM	Finite-state Machines
MCTS	Monte Carlo Tree Search
CNN	Convolutional Neural Network
A2C	Advantage Actor Critic
PCG	Procedural Content Generation
PCGML	Procedural Content Generation via Machine Learning
PPO	Proximal Policy Optimization
ANN	Artificial Neural Network
2D	Two Dimensional
3D	Three Dimensional
MDP	Markov Decision Process
TD	Temporal Difference
TRPO	Trust Region Policy Optimization
VPG	Vanilla Policy Gradient
ReLU	Rectified Linear Unit
MDP	Markov Decision Process

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

The video game industry has been the largest entertainment industry since 2019 according to their global revenue [1]. This industry is increasing every year and growth is much more than other industries. The mobile game industry is one of its sections where the growth rate is high as well. Therefore, working on a better game development pipeline has a great impact on the game industry.

Game mechanics usually depend on a person's ability, critical thinking, and quick responses. That's why it is a difficult and time-consuming task to develop, achieve and create an end-product. However, while trying to create a game that is loved by the majority of players, game creation will be prone to evolving a complex task because a game may contain a non-player character or a game may contain puzzles to solve. Creating a good non-player character and challenging puzzles are needed to be developed by a professional level designer. These games confront this problem more likely. To cope with those issues without a level designer, traditional methods can be used such as an NPC with random actions, rule-based actions where there is no usage of machine learning-included algorithms and methods.

Since the demand in the video game industry is great and increasing, video games developers also use different approaches. The most prominent one is using Machine Learning methods. Reinforcement learning is one of the basic machine learning paradigms which is very suitable to use in video games. More generalized, successful, and adaptive solutions are obtained compared with the traditional methods. Generating and testing new levels using genetic algorithms and machine learning is

another novelty and the solution for a task like this.

There are three types of match-3 games. One is swiping one item with another to match three same items in horizontal or vertical. The second one is choosing an item and swiping to neighbor items that have the same color as the first item. The third one is tapping one item that contains neighbor items with the same color recursively. Studies generally focused on the first and third types of games since it is the most classic one and it is very simple to create a well-defined environment for them. The second type of match-3 game is used in this thesis.

In this study, the focus is on a match-3 game, called "Zombie Blast". The problem is creating new levels for this game as fast as possible and ensuring the difficulty of the levels. The need to solve this problem is an automated level generation producing reliable results. The motivation is that generating levels for this game and testing them autonomously leads to a decrease in time spent for level design.

In Zombie Blast, there is a grid where 5 different colored items are aligned and you need to match more than 3 items by linking them which can be done starting on an item and sliding it to the other consecutive same colored items where consecutive items may appear in all 8 directions. Every level must be finished somehow, whether it is won or lost. For Zombie Blast, if you clear all obstacles, you won. However, if you exceed the number of moves given at the beginning of the level, you lost. Designing new levels and testing their difficulty for a game like this may take a long time. A level designer has to be sure about its difficulty and the uniqueness of the level to add the new level into the live product. This pipelining is very time-consuming and it may still have some missing parts in terms of testing which is used to find the difficulty of the level. To cope with these problems, in this study, I tried some solutions to reduce the time spent for generating new levels for this game by using genetic algorithms and testing these levels' difficulty by machine learning-based methods.

1.2 Proposed Methods and Models

There are several candidate algorithms to generate new levels for a game autonomously. Genetic algorithm is one of the best ones for Zombie Blast like games where we need

a few levels designed before creating new ones. New levels can be generated as a mixture of old and newly generated ones with some changes. The aim is to solve the level generation problem by using simple but effective algorithms and to design new levels in a shorter time.

After new levels are generated, they need to be tested to set their difficulty. Machine learning methods are used to test these levels. The bottleneck of the study is to determine the difficulty of the generated levels by using machine learning. Two reinforcement learning algorithms are chosen for this purpose. Furthermore, some modifications are applied to these algorithms to get better results for this type of game.

1.3 Contributions and Novelties

The contributions of this thesis are as follows:

- Creating a pipeline where level generation and setting the difficulty of the levels are done automatically. This makes generating ready-to-add levels into an end-product possible.
- Comparison of algorithms that are evaluated and compared each other more detailed for a match-3 game.

1.4 The Outline of the Thesis

- Related work is explained in Chapter 2
- Genetic algorithms and reinforcement learning algorithms used in this study are provided as the background of this study in Chapter 3
- Models and methods used in details such as environment, implementation, and different approaches on methods are explained in Chapter 4.
- Experimental results are shown in Chapter 5.

- A summary of the research results and a discussion on future works can be found in Chapter 6.

CHAPTER 2

RELATED WORK

2.1 Genetic Algorithms

Procedural content generation (PCG) techniques are used for game content for many reasons such as game aesthetics, game levels, automating different parts of game development. Most PCG algorithms need to be developed or tailored to each game specifically. In contrast, Procedural Content Generation via Machine Learning uses machine learning to generate content from existing content. This type of PCG technique is more effective to create unique levels. Autoencoders used with evolutionary algorithms are greater for level generation. In a study[2], they use a multi-channel approach to represent content in full fidelity, and they compare standard and variational autoencoders. They also update the values of the hidden layer of trained autoencoders to find levels with desired properties. In addition, they use a multi-population evolutionary algorithm that contains patch-wise crossover and latent variable mutation operators, which can retain some interesting features of original levels while inserting less predictable variations. Their experimental results showed that the proposed method can create diverse levels. Besides, they found out that variational autoencoders are prone to generate more complex levels than vanilla autoencoders.

Level generation is easy for a match-3 game to generate using genetic algorithms. As an example, a paper is analyzed about a level generation for a platform game using genetic algorithms[3]. They use this algorithm for the game called Prince of Persia. It is a very known platform video game where levels have limited specific objects to create one except visuals that do not affect the gameplay. Therefore, levels are divided into those limited parts such as path, wall, obstacle. The combination of these parts

may construct a valid level. They use a fitness function which can also be called the objective function. It helps to know how much the solution is close to the aim wanted to be achieved. There are simple algorithms to run genetic algorithms and they are explained in Chapter 4. This paper uses the fitness function to achieve a better design solution.

2.2 Machine Learning

Creating a new level and testing to find out its difficulty is crucial for the level-based game development process. Building a level is a complex task because of the need of keeping users interested in the game continuously. It gets harder to do when the difficulty of the game is not optimal for users. When a level is ready, the difficulty of the level is calculated by playtesting by a level designer, using finite-state machines(FSM), Monte Carlo tree search (MCTS), and convolutional neural networks (CNNs). However, either these methods are time-consuming or large data-sets are needed. In order to get rid of these disadvantages, researchers use strategic plays via reinforcement learning in a 3 match game[4]. The Strategy agent, advantage actor-critic(A2C), takes the current state of the board and 5 strategic plays which are predefined with different strategies depending on the goals of a level. The agent chooses the move to apply among the highest rewarded moves after evaluation of these 5 moves. Human players, random agents, rule-based agents, and strategy agents play all levels more than 100 times to compare the average moves to complete a level. The difficulty of a level is measured by the number of moves to finish the level. The Strategy Agent performed 15.75% better than the Rule-Based Agent and performed within a 5% margin of human performance on the most complex mission in the experiment. This shows that there is no need for actual user data for playtesting thanks to giving five predefined strategic options in A2C available. This helps level designers and developers evaluate the difficulty of a level without data collection and spending time on training.

Solving a generic problem rather than a specific one is another challenge. The strongest programs are based on a combination of sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions that have been refined by

human experts over several decades. AlphaGo Zero program recently achieved superhuman performance in the game of Go by reinforcement learning from self-play[5]. AlphaZero replaces the handcrafted knowledge and domain-specific augmentations used in traditional game-playing programs with deep neural networks, a general-purpose reinforcement learning algorithm, and a general-purpose tree search algorithm. At a high level, it uses a modified Monte-Carlo Tree Search (MCTS) backed by a deep residual neural network (or "ResNet"). AlphaZero plays against itself, with each side choosing moves selected by MCTS. The results of these self-play games are used to continually improve the ResNet. The self-play and ResNet training occur in parallel, each improving the other. In AlphaGo Zero, self-play games were generated by the best player from all previous iterations. After each iteration of training, the performance of the new player was measured against the best player; if the new player won by a margin of 55%, then it replaces the best player. On the other hand, AlphaZero simply maintains a single neural network that is updated continually rather than waiting for an iteration to complete. Self-play games are always generated by using the latest parameters for this neural network. AlphaZero has a generic reinforcement learning and search algorithm originally devised for the game of Go that achieved superior results within a few hours, by searching as many positions, given no domain knowledge except the rules of chess. Furthermore, the same algorithm was applied without modification to the more challenging game of shogi, again outperforming state-of-the-art programs within a few hours.

A similar study also uses PPO and action masking[6] for a quicker and better strategy. A match-3 game called "Lily's Garden" has the same game dynamics as the game in this thesis. Training data-set has 5 levels and test data has 6 levels. Best results are obtained for easy levels where the goal is collecting colored tiles at a constant value. As the strategy they used resetting while training a game like this giving the move in the game is not adequate to make the network learn, as first episodes may not be able to do valid actions in that limited move. Also, not giving an upper limit for resetting the level can make the learning process worse since when the number of moves is increasing, rewarding the useless or bad states can be prevented. To make values of different colors equal, they switch colors in observation space and feed the network with switched layers of color representations. This strategy helps the

network perform better at all levels. The most important part of this work is adding action masking for the PPO algorithm. They use 2 types of action masking; hard-masking and soft-masking. Hard-masking limits the agent to make an invalid action by masking logits of policy distribution by changing invalid actions' value to $-\infty$. The second type of masking, soft masking, is adding a layer on observation space where valid actions have the value 1 and invalid 1s are 0s. Softmasking using color switching at the beginning of the episode outperformed all other methods. However, there are some not well-defined tests on action-masking. The first problem with action masking is training an agent with hard masking and changing logits' to $-\infty$ affect the network badly and fill the action distribution with $-\infty$. Therefore, there are no left plausible actions in the distribution. Therefore, it is not different from selecting a random action. In this thesis, we used a different approach for creating hard masking which is explained in detailed in Chapter 4.

CHAPTER 3

BACKGROUND

3.1 Genetic Algorithms

Genetic algorithms are created by John Holland et al [7, 8]. Genetic algorithms are self-explanatory because these algorithms inspired by biological evaluation imitating chromosomes' behavior. There is a general procedure explained[9] by following these steps:

- Defining the objectives or cost functions
- Choosing a fitness function or limiting with a selection criterion
- Generating a population
- Iterating the evaluation cycle with individuals in population and creating a new population by using genetic operations
- Resolving results to get the solution

A simple representational schema is adopted to encode the problem. Strings can be used for encoding and decoding by defining characters for the problem.

3.1.1 Genetic Operations

There are three main genetic operators and they are explained below:

- **Crossover:** Exchanging parts with the same objective or mechanics between two individuals like chromosome crossover and it helps creating new different

individuals. For example, if we represent a problem with zeros and ones and there are two individuals which are encoded respectively 001010 and 110001. After a crossover operation which is swapping first 3 numbers and last 3 numbers in bulks, the result may be 001001 or 110010. Crossover operation can be more complex and changing dynamically with a probability to generate more different individuals to avoid generate same ones after a few steps.

- **Mutation:** After crossover operation, there can be a mutation at the new individual. Randomly selected characters from strings can be changed with different characters to mutate strings. Mutation can be done by changing characters or a set of characters as a larger gene part. This will increase the diversity in the population and help us to not stuck at a local optimum diversity.
- **Selecting the fittest:** After creating a new population, some of them are not good or not working for the solution we want. Therefore, some of them are selected and those selected better individuals are moved to next generation to create best optimum population and the bad ones are got rid of at the point they created.

3.2 Machine Learning

Machine learning is a part of artificial intelligence that focuses on data. Statistical learning and optimization methods are used to get better accuracy on some tasks that humans do such as detection and recognition of an object, speaking a language, creating new things like songs, paintings, playing games professionally, driving cars, solving real-life problems.

Machine learning has three main concepts as defined by UC Berkeley[10]:

a. A decision process:

General usage of machine learning is about classification or prediction. In order to do that, the learning system tries to produce best results from an input by trying to find a pattern in the data.

b. An error function:

An error function is used for finding how good the result the learning system produced compared to the results that is known that they are true values if there is any.

c. An updating or optimization process:

The algorithm runs to get better results comparing the true results. The learning system tries to find a better pattern by changing its values to get more accuracy on the task and optimize the all results by repeating this steps many times to converge its values, i.e. weights, and try to produce better results.

There are three paradigms in the machine learning depending on what feedback, signal, or reward is received while the learning is processed:

Supervised Learning:

The learning process runs under supervision. Data are tagged or labeled for this purpose. Two main problems using supervised learning are classification and regression. Classification is about finding a discrete result such as recognizing which animal is in an image. There are certain discrete classes, groups and the algorithm tries to find the best fit/output for the input. The second one, regression, is in continuous space and it tries to find a result that fit right or approximately. For example, trying to predict the value of a price of an object by using factors that affect its price for such an object like a computer, a house, or a car.

Unsupervised Learning:

Labeled data is generally prepared by humans and in some tasks it is hard or not possible to label the data. Hence, researchers tries to get good results without labeled data from an algorithm. In order to do that, they force the algorithm to create a pattern in data given without labels by choosing features and grouping data with those values. Since there is no true value for results, it may difficult to be sure how good the algorithm works for the task, but it may not always be a problem.

3.2.1 Artificial Neural Networks

ANNs are systems which have layers that have units called node. There are input and output layers. Input layer is the representation of the problem. Output layer is the representation of the result. There are one or more hidden layers between the input and output layers. Each layer is connected the layers above and below it. The connections have weights which represents the connecting strength. Nodes are the representation of neurons of a human's brain. Nodes get values from nodes that are in the previous layer and transmit their values to other nodes that in the following layer when activated. Learning in ANN is updating the connection weights to find out the best fitting pattern.

3.2.2 Convolutional Neural Networks

Convolutional neural networks are neural networks specialized for 2D inputs. Besides 2D, it can be used for 1 or 3 dimensional inputs such as images with different channels for each pixel. There is an array of weights which is 2 or 3 dimensional according to input dimension and smaller than the input, called kernel. It traverses on the input by multiplying element-wise overlapped values of the input with its own. It transfers the calculated result to the next layer. This is why it is called "convolutional" and it tries to create and keep features of an image while processing these operations. Thanks to convolution, features are not lost after shrinking the image by filter and it helps to process image efficiently and faster than primitive methods and predicts with more accuracy.

3.3 Reinforcement Learning

The learning process is in an environment with an agent which is the part of the learning process and the agent make an action in the environment to finish a goal. After that, there is a reward or punishment system according to the action or depending on the algorithm used. the agent tries to get more rewards and avoiding to be punished by changing its weights to get a great construction of the task. At the end, the agent

predicts the best action to get the highest reward and achieve the objective.

Reinforcement learning algorithms are goal-oriented algorithms. They have a goal to achieve or a value to maximize at the end of an episode. Those algorithms aims to teach a task to an agent by penalizing them if the agent makes a wrong choice, and giving reward if its decision is a good one. These algorithms surpass humans and also champions in games. The success of reinforcement algorithms increase rapidly and they are now able to play Atari games and also some 3D games that compete with humans.

The related definitions and terms are explained below:

Agent: Agent is the part of the learning in reinforcement learning and they take actions in an environment. For instance, player in a game is the agent or a robot which tries to learn to walk is the agent.

Action: A single action is represented with **a** and it is an element of the set of all actions denoted by capital **A**. Action is taken by an agent to change the situation and getting know what to do by taking feedback at the end of the action taken. Action is usually needed to be determined before, it may discrete or continuous, but it must be finite like number of actions for a discrete action space or a range for a continuous action space. For example, in a platform game, jumping, going forward or backward, shooting, going left or right, or changing weapon are all actions that an agent may take in a game.

Discount Factor: Discount factor is expressed with the letter gamma γ. It is multiplied by future rewards that will be given and it usually smaller than 1 because we want to decrease the effect of future rewards. For instance, if there is an immediate reward at time step 4, then the reward from this step will be multiplied by γ 4 times and this decrease the value of the reward at time step 4. If γ is 0, then the agent only learn from the immediate rewards and if γ is 1, then the agent take all sum of future rewards and process all of actions.

Environment: Environment returns the reward, next state after taking and analyzing the agent's action and current state. It checks and controls the impact and the reaction of the impact.

State: States can change when an agent take an action and it goes into a different state as a result. Set of states are denoted by capital **S**, whereas current state and next state are usually represented by **s** and **s'** respectively. State is where the agent is and all other things and their relation between them. For example, in a grid representation a state contains obstacles, enemies, rewards at some points at column and row indexes.

P(s'ls, a): the probability of that the next state is s' when the agent is in current state s and takes the action a.

Reward: Reward is a result of an agent's action is given by the environment according to current state and action. Reward can be positive or negative depending on whether the action is good or bad by calculating the distance to the goal by the environment. For instance, if a player get coins while playing it gets positive reward, but if it is attacked by an enemy, player gets negative rewards, i.e. be punished.

R(s, a): the reward function that return the reward value of the action.

G(t) or R(t): total future rewards at state s_t

Episode: All states and actions from initial state to terminate state.

Policy: Policy is represented as π and it evaluates current state in order to choose the next action for the agent. It tries to get the best action with the highest reward based on the current state.

Markov Decision Process(MDP): Every state has a relation with their previous state with the action executed and reward taken in that previous state just before going to the new state. No need to know what happened before the previous state to make the transition because Markov Property assumes that the previous state s' keep the old information from previous ones. This assumption results in the Markov Decision Process and it can be written as like below:

$$s' = s'(s, a, r) \tag{3.1}$$

Value function: The value function V tells us the expected overall value of a state s_t

under the policy π and the formula is:

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\} \tag{3.2}$$

Q function: The Q function tells us value of a state s_t and an action a_t under the policy π and formula can be written as below:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t|s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\} \tag{3.3}$$

The relation between Value and Q functions can be written as:

$$V^{\pi}(s) = max_a(Q^{\pi}(s, \pi(s))) \tag{3.4}$$

Advantage Function: Advantage function helps us to know how good the specific action a taken at state s by taking the difference of Q-Value of the state s and action a and Value that gives the average of actions at state s.

$$A(s,a) = Q(s,a) - V(s) \tag{3.5}$$

Bellman Equation: Bellman equation shows us the relation between a state and the future ones. The current value is the sum of current state's reward and discounted value at the next step with policy π.

$$V(s) = max_a(R(s,a) + \gamma \sum_{s' \in S} (s' P(s,a,s') V(s')) \tag{3.6}$$

$V(s)$ is considered as the maximum value of the action from all possible states and all actions pairs. This is just for one action and it can be written as a Q Value since it gets the value of a state s and action a:

$$Q(s,a) = R(s,a) + \gamma \sum_{(s' \in S} (s' P(s,a,s') V(s')) \tag{3.7}$$

Since $V(s)$ is defined as the maximum value of $Q(s,a)$, some changes in the equation result this optimum bellman equation:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} (s' P(s,a,s') max_{a'} Q^*(s',a')) \tag{3.8}$$

Temporal Difference: Temporal difference is an approach to learning by bootstrapping and do updates from the current estimate of the value function.

Policy Iteration: From randomly selecting a policy π, policy iteration takes place as follows:

1. Evaluate the policy at state s selected in order to find the value function of the policy and this step can be called "policy evaluation".
2. Find a new improved policy according to previous policy and this step can be called "policy improvement".
3. Iterate 2 steps until finding the optimal policy.

Value Iteration: Instead of selecting a policy randomly, we select a value function randomly at the beginning. Then, it takes place as follows:

1. Find a new improved value function based on previous value function by maximizing the value.
2. Iterate step until reaching the optimal value function.
3. When optimal function is found, one can derive the optimal policy from the value function as in Bellman equation.

Exploration & Exploitation: Reinforcement learning is a different from other machine learning paradigms since data is collected while training the agent by giving the current state according to the action the agent makes. The agent needs to find better states to maximize the reward by choosing different actions as much as it can by wandering around new states and performing actions. This is called "exploration".

However, if the agent always explore their environment, they can not improve their overall reward and stuck at average. From their explored states, agent should exploit action-state pairs that are better for increasing the reward. This is called "exploitation". The trade-off between these two should be in a balance to find the best exploited path by using many explored paths.

Model-based & Model-free Approaches: There are 2 types of techniques used in order to optimize the policy by the agent. It can be explained by giving an example from a game. Checkers can be an example for this. Before playing the game, you can produce values of all states, state-transition probabilities or state-action pairs and calculate those values. After calculation is done, you can act and play according to choose an action at a state that get the most valuable reward by checking calculations produced before. This is called Model-based approach because it is learned from a deterministic model created, all information was ready to exploit. On the other hand, in a model-free approach, the agent learns by playing the game randomly using trial-and-error and explores new states and retrieves the information. The main difference between model-free and model-based approaches is the first one construct the environment but the latter has no information about the environment and it tries to learn the environment.

On-Policy & Off-Policy: Reinforcement learning algorithms generates a policy for the agent to decide the action to take in each state. On-Policy algorithms have goal to improve the policy at every step and try to find the optimal policy. In other words, an on-policy algorithm concerns about the policy. However, off-policy algorithms don't focus on the policy. For example, Q-Learning is an off-policy algorithm that store all Q-Values and update them by using Bellman equation and the agent choose the optimal action to get the maximum reward from the state and the exploitation increases.

3.3.1 Policy Optimization Algorithms

The main goal of policy optimization problems is to reach the optimal policy by increasing the probabilities that gives positive reward and decreasing the probabilities that gives negative reward, called Vanilla policy gradient[11] and VPG equation is

below:

$$\nabla J(\theta) = E_{\pi_\theta}[\nabla_\theta log_{\pi_\theta}(a_t|s_t)\hat{A}_t] \tag{3.9}$$

Major problem of vanilla policy gradient is the high variance and this problem is solved by using Advantage function at equation (3.5). Advantage function indicates that how good the action taken by subtracting the average action at state s. This helps agent to know the effect of action better and decrease the variance in the gradient. Thanks to advantage function, a safer change is made for transition from old policy to new policy. However, the advantage function is not good enough to find optimal policy because of noisy estimations it creates. The reason is that the gradient ascent is estimated by line search and if the step size is big and the direction is wrong, it can be irreversible and stuck at a bad local optimum. Solution may not be reducing the size because convergence can take long time.

3.3.1.1 Trust Region Policy Optimization

A simple policy gradient computes the first derivative and better works for flatter surfaces. However, if the surface have many curves, an action may produce a worse result. If the result is far away from the optimal, it is very hard to read just the approximator to a better state from a bad situation. If our steps are too small, getting to a better place may take too long time.

Trust region stands out from other optimization methods such as gradient descent. For gradient descent, a direction is determined first and then the update is applied with a step size. However, in the trust region, maximum step size is determined first before selecting a direction. In this determined range, an optimal point is being searched to choose the direction with the best result. This optimization technique is introduced[12] as trust region policy optimization(TRPO) and it bring the term of trust region.

3.3.1.2 Proximal Policy Optimization Algorithms

Proximal Policy Optimization(PPO) has the same stability and reliability like TRPO and is easy to implement by using vanilla policy gradient and it has a better performance than many methods. There are two major contributions that Proximal Policy Optimization[13] provided.

The first modification is **the clipped surrogate objective**. The main problem of the policy optimizations is updating policy for improving the stability of the update by limiting the change of the policy at every step. We presented the vanilla policy gradient equation at (3.9). VPG uses the log probability of the actions to follow the effects of actions. A different function is introduced[14] that is the ratio of the probabilities with action a under the current policy divided by the probabilities with action a old policy:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{3.10}$$

- if $r_t(\theta)$ is greater than 1, then the action is more probable for the current policy than old one.

- if $r_t(\theta)$ is between 0 and 1, then the action is less probable for the current policy than old one.

If we replace the log probabilities at vanilla policy gradient equation with this function $r_t(\theta)$, we can obtain the objective function of TRPO:

$$L_{TRPO}(\theta) = \hat{E}_t[log_{\pi_\theta}(a_t|s_t)\hat{A}_t] = \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t] = \hat{E}_t[r_t(\theta)\hat{A}_t] \tag{3.11}$$

This gives the information about how much an action will affect the policy but depending on the policy, the change may still be huge for one update on the policy. Instead of tweaks and optimizations from TRPO, PPO adds a clipping surrogate to avoid those big steps and also it does not need to calculate more complex equations.

The clipped surrogate objective is:

$$L_{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{3.12}$$

The major change is limiting the $r_t(\theta)$ by clipping it between $(1-\epsilon, 1+\epsilon)$ where ϵ is usually 0.2 to get a good results. If the A_t is positive where A is the advantage value for that state-action pair, then $(1+\epsilon)$ is the limit but only limiting the positive side and vice versa.

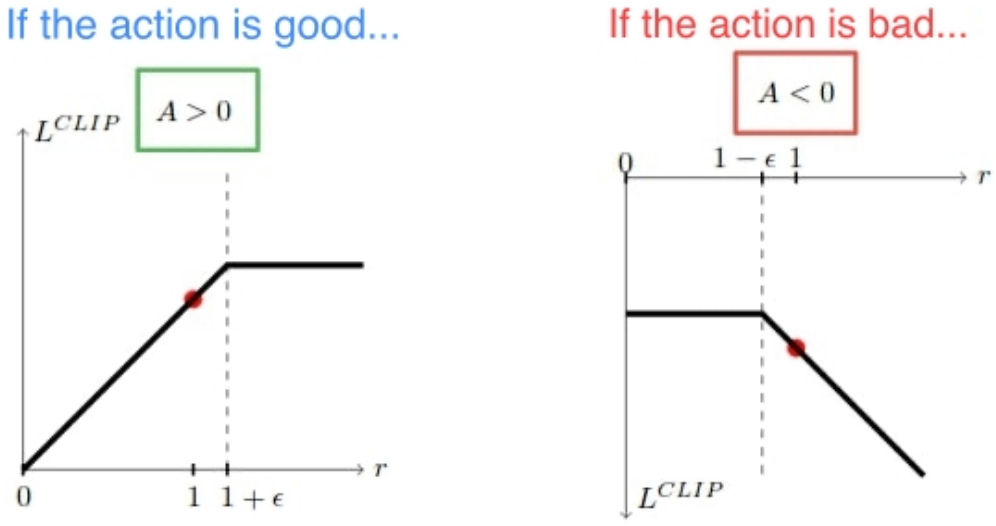

Figure 3.1: Conditions of Advantage Value [1]

The left side tells us that the action is better in the current policy and the result will be good after the action. However, the right one is the opposite and the action is bad for the agent to take and the result will be worse because of this action. Policy updates are used to converge to the more probable weights for the network when an action is taken and $A > 0$. If $A < 0$, then policy update will avoid to take this action after. Thanks to clipping, big changes are limited and this helps to keep a steady policy update. Even if the action returns a positive or negative reward which is large and affect the policy update, this is prevented by clipping because this will not be accurate since it is a local approximation for the policy. For the right side of the right diagram in 3.1, the negative value of L^{CLIP} is not limited by the clipping and that is

[1] Figure retrieved from https://stackoverflow.com/questions/46422845/what-is-the-way-to-understand-proximal-policy-optimization-algorithm-in-rl

because of providing a recover for the policy by making less probable the action since that action makes policy worse.

The second modification is **multiple epochs for policy updating**. This gives the opportunity to approximator to run multiple epochs on samples while vanilla policy gradient methods cannot do without damaging the learning since they take large steps. Running multiple epochs on samples provide getting more information from samples as well and this decreases the inefficiency for samples.

3.3.1.3 Advantage Actor Critic

Actor-critic algorithms[15] have 2 separate approximators networks. First one learns to choose better actions and the second one criticize by taking the action that actor network chose and the current environment and return the Q-Value that is the maximum reward that can be taken by the action at state s. Actor chooses better actions and the critic one also evaluates those actions with the state and they update their weights during training. The important thing is that those updates are not at the end of the episode but at every step, unlike policy gradients.

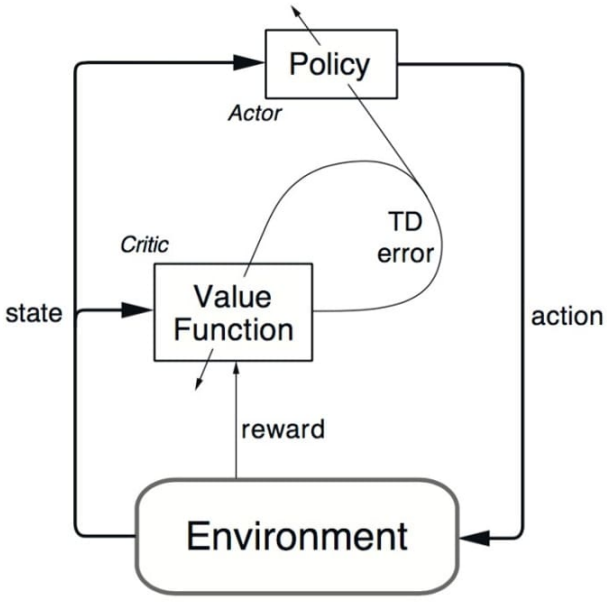

Figure 3.2: Advantage Actor-Critic Algorithm [2]

[2] https://kargarisaac.medium.com/rl-series-a2c-and-a3c-in-pytorch-6e9edf5c8788

Advantage Actor Critic3.2 is the version of that the critic network that gets the same input but returns the advantage values instead of Q-Value. Thanks to Advantage values, we can know how much an action is better than other actions that are valid at the state s. Hence, the high variance of policy approximators can be decreased by using advantage values instead of using just Q-Value.

3.4 Game Environment

Zombie Blast is a hybrid of casual and role-playing match-3 game. There is a 7x7 grid and the items on the grid with 5 different colors. Also, there is an upper section that our hero stands and shoot zombies coming from the right, in a wavely manner. Upper section has connection that each item has a different attack on zombies. However, even if there are different types of attacks, if you clean the grid from obstacles by doing a match close to obstacles, zombies will be defeated as well at the end of the level. The game is restricted by a number of moves you are able to taken in a game. The objective is that you need to clear the grid before you exceed the move given or you lose. For this reason, the focus is only on the grid part of the game.

There are some obstacles we try to break and clean from the grid. Those are stones and bubbles. If you do a match near to a stone, stone will be destroyed or broken into weaker stone depending on its state. Most powerful state can be destroyed by doing matches 3 times next to a stone. Bubbles are popped and cleaned when they are used during a match since they contains an item in it.

Doing a match is very easy and there is a simple difference than other classic match-3 games. You need to touch an item to start a match and continue by swiping your finger to next same colored item. Valid neighbors are left, right, up, down and also diagonal items. You need to match more than 2 items to create a valid match. There can be 3 different results of the matches you did if they pass some thresholds and they are called "combo". Those are:

Figure 3.3: Game scene of Zombie Blast by SNG Studios[3]

- If the number of matched items is equal or greater than 5 items, a two-way (horizontal or vertical) rocket is spawned at the position of the last matched item. Those rockets can be used to clear a row or column and they will be activated when they are used in a match independently from the color of the match.

- If the number of matched items is equal or greater than 8 items, a four-way (horizontal and vertical rocket) rocket is spawned at the position of the last matched item. This rocket can be used to clear a row and also a column and they will be activated when they are used in a match independently from the color of the match.

[3] https://play.google.com/store/apps/details?id=com.sngict.survivors.zombies&hl=en&gl=US

- If the number of matched items is equal or greater than 10, a magnet with a color is spawned at the position of the last item of the match. This colored magnet can be used to clear items with the same color with itself. A magnet will be activated when it is used in a match dependent of the color of the match.

Besides of combos, there are also more options when there is a consecutive combos on the grid. Those are:

- if two two-way rockets are next to each other, then they are merged and create a four-way rocket and activated.
- if a two-way rocket and four-way rocket are next to each other, then they are merged and create 9 four-way rockets around them and itself and they are activated.
- if a two-way or four-way rocket and a magnet are next to each other, then they are merged. All items with the same color with the magnet's color are turned to those rockets and they are activated.
- if two magnets are next to each other, then they are merged and clear all tiles at every position at the grid once.

CHAPTER 4

PROPOSED WORK

4.1 Genetic Algorithms for Level Generation

First of all, population initialization is needed to start generating levels. The population is retrieved from existing levels of Zombie Blast. Levels are encoded as letters for the game. Encoded levels are also used for the representation of an individual of the population. An individual is also called a chromosome. Letters used for encoding are called genes. The phenotype of a level is shown in the game as a grid and images. A genotype is a description with strings to decode from letters. An example of an encoded level is shown in Table 4.1 and its decoded version is shown in Figure 4.1. The next step is to apply genetic operations on individuals of the current population to create new individuals. Lastly, new individuals are selected and added to the new population. After reaching termination criteria, new levels are created and they can be transferred directly to the game since the game also uses the same encoded version for a level to decode.

Table 4.1: Encoded Version, Genotype of a Level

B	-	-	f	-	-	0
-	B	-	-	-	1	-
-	-	B	h	2	-	-
*	*	-	B	-	*	*
-	-	2	B	B	-	-
-	1	ic	*	-	B	-
0	-	-	*	-	-	B

Figure 4.1: Phenotype of a level

Using genetic algorithms to generate a new level is very efficient since all states are deterministic and can be created by using simple language. Grid is represented with characters and a nested array is used for encoding a level for creating an individual for the genetic algorithm. The alphabet has simple characters for different tile representations. Those are explained below:

- "-" is used for a random colored tile
- "_" is used for empty tile but can be used if obstacles above are cleared, tiles can fall there
- "*" is used for a block or out of grid tiles that cannot be used at that level
- "a" is used for green hand item
- "b" is used for red heart item
- "c" is used for blue eye item
- "d" is used for purple brain item
- "e" is used for yellow bullet item
- "f" is used for horizontal two-way rocket item

- "g" is used for vertical two-way rocket item
- "h" is used for four-way rocket item
- "ia", "ib", "ic", "id", "ie" are used to generate a magnet with the color they contain the letter from which are respectively green, red, blue, purple, yellow magnets
- "0", "1", "2" are used for stones that have different strength levels. "2" is weakest and broken after one match next to it. Others' stronger one and two more than "2", respectively "1" and "0".
- "B" is used for bubbles that contain a tile and popped when you use this item in a match.

Grid is represented as a collection of these letters in 7 arrays of arrays with the length of 7. Fitness function is not used. A selection criterion is used instead because level design aesthetic is the most important thing in level design. Having many levels with some symmetrical shapes and good designs can support the genetic algorithm if types of selection criteria are specific to those level concepts, new generated levels will be also great and different from old ones as well. To keep the symmetry of grids, genes are selected as a larger gene, and crossover operations are applied to these genes by combining appropriate genes together. 5 different crossover operations split the grid and doing other genetic operations at those parts. These 5 operations are:

- Half of the grid by cutting vertical in the middle. Since the grid has 7 columns, the left one has 4 columns and the other right one has 3 columns and vice versa.
- Half of the grid by cutting horizontal in the middle. Since the grid has 7 rows, the upper one has 4 rows and the other bottom one has 3 rows and vice versa.
- Dividing the grid into 4 square is the other separation method used. One of them has 4 columns and rows and the others 4x3, 3x4, and 3x3. The position of the section with the 4x4 is changeable with a probability of 1/4 like the first separation methods. An example of parts is shown in Figure 4.2.

- The other separation method is dividing the grid to generate the largest plus sign with the middle column and middle row and the remaining parts are united to be a different section. An example of parts is shown in Figure 4.3.
- The other separation method is dividing the grid to generate the largest X sign with the middle column and middle row and the remaining parts are united to be a different section. An example of parts is shown in Figure 4.4.

Figure 4.2: Partition Example for a Crossover Operation

Figure 4.3: Partition Example 2 for a Crossover Operation

To generate more unique levels with these 5 crossover operations, mutation operations used are:

- Using the same part for another level at a different position. For example, generated parts can be divided into 4 parts as explained as the third item of

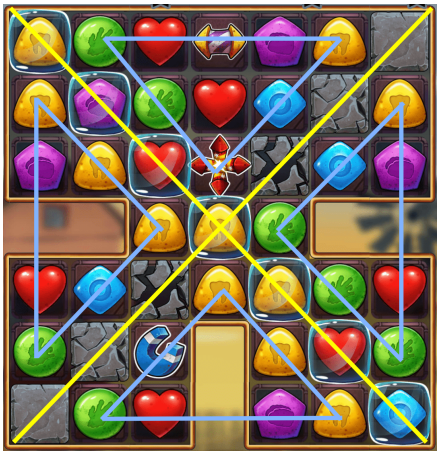

Figure 4.4: Partition Example 3 for a Crossover Operation

crossover operation. Let's say those parts are 4x4, 4x3, 3x4, 3x3 and positions are respectively upper left, upper right, bottom left, bottom right as shown in Figure 4.2. While generating parts and changing those parts, this order can also be changed to increase diversity and we can put those parts 3x3, 3x4, 4x3, 4x4 respectively into a new individual.

- Besides changing the position of parts, the rotation of a part is also used if it is applicable. The rotation can be local or global like rotating only the part itself or rotating the whole parts by rotating some of them clockwise and some of them counterclockwise and putting those parts into a 7x7 grid to create a proper grid for a level.

- While changing a part and doing a crossover operation, the alphabet can be used like if there is a stone obstacle in a part, this part can change its obstacle type into bubble obstacles, empty cells to avoid using the same obstacle types in different levels and it helps to generate fun, different and unique more levels.

For example, in Figure 4.5, the top half of the first level is taken, flipped over the X-axis, and place at the bottom of the new level. Stones are changed to bubbles as well. The bottom half of the second level is taken and place at the top of the new level. Bubbles are changed to empty cells and stones are changed to stronger ones. Other results for generated levels are shown in Figure 4.6

Figure 4.5: An example of crossover & mutation from 2 levels to generate new level.

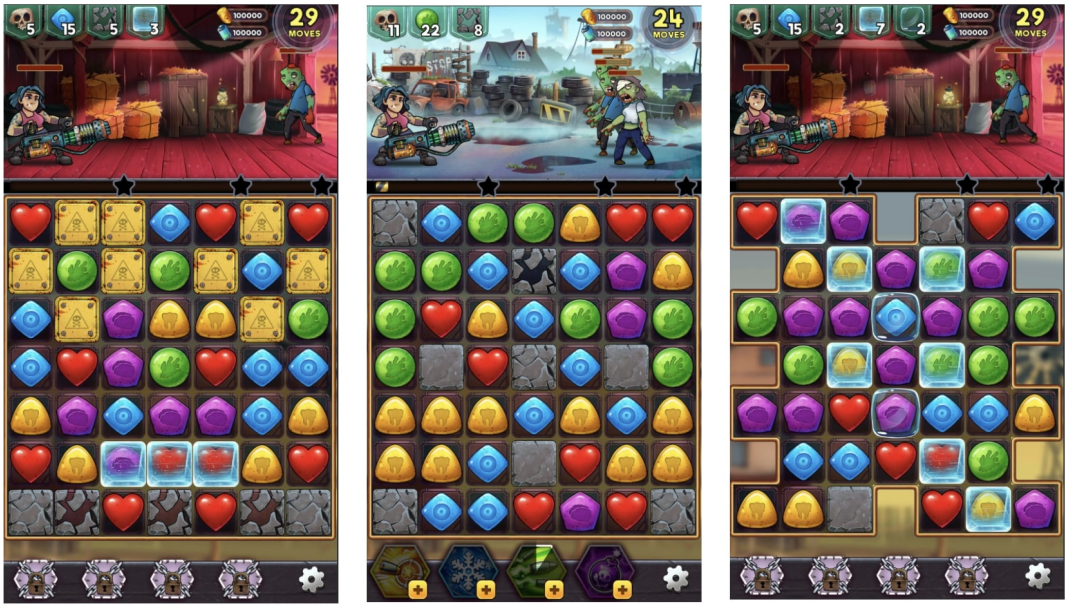

Figure 4.6: Generated levels.

Whether a generated level can be finished or not is not tested here because invalid level generation is probably impossible to create and the main contribution of this study is testing the difficulty of a level. Therefore, the purpose of genetic algorithms used in this study is to generate only good-looking unique levels.

4.2 Game Solver

To be able to set the difficulty for a level, the minimum number of moves needed to finish a level must be known. If the minimum number of moves is chosen as the given moves for a level to finish, the difficulty would be the hardest. If the given number of moves is increased, the game would be easier to be completed. Therefore, solving a level as fast as possible means that the hardest solution is found. To achieve the fastest solving, a game solver that is implemented by using reinforcement learning algorithms is used. There are some helpers to create a clean, reliable solver. Stable-baselines[16] library is used to implement reinforcement learning algorithms. To create an interaction between the game and algorithms, a custom gym environment[17] is built. The gym environment needs observation and action space to run. It is explained first and the network is shown in detail. After basic implementations and interaction are done, changes in implementation and observation for different strategies are explained.

4.2.1 Observation and Action Space Representation

There are different approaches for the representation of the input and action types. Atari game problems are usually getting the input directly from the screen that means the input is retrieved from images of the game. Besides, a new representation can be generated by hand choosing important parts of the observation and create the environment only using simple and necessary features. For action space, there can be discrete or continuous action sets depending on the problem but they must be finite and initialized at the beginning.

4.2.1.1 Observation Space

Observation space means a set of values gives information about the environment state that the agent can retrieve information from. Observation space for this problem is generated to get only the necessary features to feed the network as inputs. The game is implemented in Python language without visuals to get rid of the overhead of

visuals, animations, and other resources spent unnecessarily.

The observation space is a box-shaped one with a width and height of 7 and the depth with the number of channels 18. Width and height represent the grid positions and all layers of depth represent a feature from the grid. Those layers are shown in Figure 4.7:

Figure 4.7: An instance of a game scene and a level with its representation in channels in the right. The last 6 layers are included in specific experiments

- First 5 layers are for default items that create matches and each layer has one color. For instance, layer 2 is for items that are red heart. In this layer 2, if there is an item that is the red heart then this unit of the layer will be 1 and if there are other items, not the red item, the unit will be zero.
- Layers 6,7,8,9 are for combo items. Layer 6 is for the horizontal rocket, layer 7 is for the vertical rocket, layer 8 is for the four-way rocket and layer 9 is for the magnet. Magnet color may vary, it only keeps the information whether it is a magnet or not.
- Layer 10 is to know if the cell is playable/valid or not. Some of the cells are not used for that level and those cells are marked on layer 10.
- Layer 11 is for obstacles that contain a colored item and needed to be used in a match to be destroyed.

- Layer 12 is for obstacles that do not contain a colored item or another thing and only themselves. This layer is those obstacles that are destroyed when you do a match next to them.
- Layer 13 is for action masking and it is explained in subsection 4.2.4.
- Layer 14 is for another type of action masking and it is also explained in subsection 4.2.4.

4.2.1.2 Action Space

Table 4.2: Action space distribution

0	**1**	**2**	**3**	**4**	**5**	**6**
7	**8**	**9**	**10**	**11**	**12**	**13**
14	**15**	**16**	**17**	**18**	**19**	**20**
21	**22**	**23**	**24**	**25**	**26**	**27**
28	**29**	**30**	**31**	**32**	**33**	**34**
35	**36**	**37**	**38**	**39**	**40**	**41**
42	**43**	**44**	**45**	**46**	**47**	**48**

Action space means choosing which distinct action to apply from a finite action set. Action space has discrete actions and those actions are determined by considering the game and its gameplay. To make a valid action, you need to swipe more than 2 same-colored items or with combos to make an action. Action space is very huge and it cannot be initialized if we would like to keep all item positions that create a valid action. Hence, I decided that actions have two points which are its start position and end position. However, selecting items by checking start and end position is also a problem because it can be different as well. There can be two paths for a match that is the shortest or longest path. There is a problem again to find a deterministic way to represent an action since the agent may not select all items they can get to make a strategy. Therefore, action space remained simple by using only the endpoint of the action because the most important strategy in this game is to end a combo next to

another combo and create a bigger explosion. Since combos are spawned at the end of the action we keep action information by keeping only the endpoint of the action. Selecting items with the endpoint is selecting the longest path that its endpoint can be the position of the end of the action. We simplified action space and give the network the most important information to make strategies because there is no difference to make different paths if it will be the longest and the end-point is determined before. Therefore, The number of actions is 49 and they are shown in Table 4.2.

4.2.2 The Network Architecture

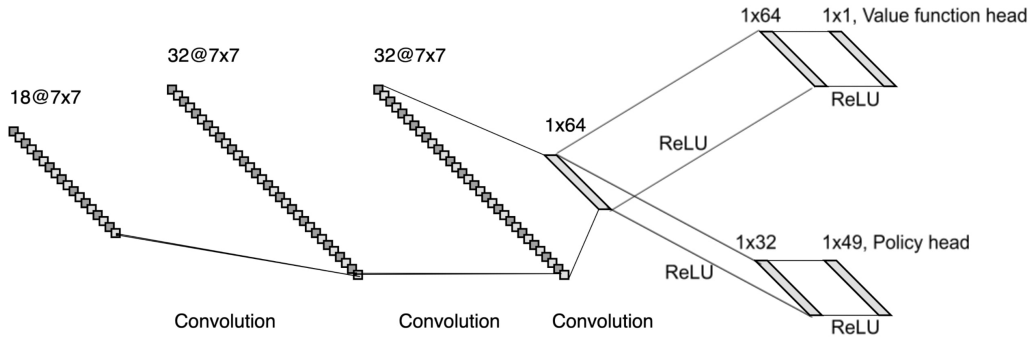

Figure 4.8: The network architecture of the agent[1]

A custom convolutional neural network is used for feature extraction from the input and they are fed into the value function and policy heads as shown in Figure 4.8. The input size is 18x7x7 with a 1x1 kernel size and the filter size is 32 for 2 convolutional layers that use Rectified Linear Unit(ReLU) activations. These features are fed into shared fully connected 64 layers. It forwarded to actor and critic heads that are 64 layers for value function head and 32 layers for policy head. The output of the policy head is values of all action, 49 output units, where the input is state s and the value function returns only the value with 1 output unit.

[1] Architecture is created on http://alexlenail.me/NN-SVG/LeNet.html

4.2.3 Methods and Algorithms

Many rule-based agents are implemented to compare the agent learned by reinforcement learning algorithms. The strategies that the rule base agent uses are as follows:

- Random Player: Choose a random tile from the grid as the endpoint of the match.
- Rule-based Player 1: Choose the longest match they found in a state.
- Rule-based Player 2: Choose the match that clears most obstacles or decreases their value.

The results obtained from human players are also used for comparison with our agent. Those comparisons give more accurate values to determine the success of the agent. The agent has a CNN for feature extraction because CNN is better for finding a relation between nearby pixels and for our input in this study it can be easier to find relation nearby tiles and that is the most crucial thing for this case. Game-play is about finding good relations nearby tiles. After CNN extracts the features, we tested these features with two different reinforcement learning algorithms, Advantage Actor-Critic and Proximal Policy Optimization. These two algorithms are the ones mostly used for match-3 games. Different hyper-parameters are used and tested for each algorithm to get the best results.

The reward function is another vital part of a reinforcement learning algorithm. Many reward functions are tried to find the optimal formula. Main reward functions given in Table 4.3.

In the above formulas, c_{goal} is the number of decreasing or cleaning an obstacle where $n_{obstacle\_count}$ is the total count of obstacle in a level and this dynamically changes based on level. The last variable is $c_{finished}$ and it is 1 if all obstacles are cleared and 0 if obstacles are remaining after a specific number of moves exceeded. The limit for the number of moves is 150 for one episode. If this number is exceeded, it is interpreted as the agent failed at passing the level. Choosing the limit is critical as well because if the limit is low then the agent cannot learn from the episode. The reason

Table 4.3: Reward Functions

Reward Function 1	$r = c_{goal}/n_{obstacle\_count} + c_{finished} - 0.1$
Reward Function 2	$r = c_{goal} + c_{finished} - 0.1$
Reward Function 3	$r = (c_{goal}/\|c_{goal}\|) * c_{goal} * c_{goal}/n_{obstacle\_count} + c_{finished} - 0.1$
Reward Function 4	$r = c_{goal} + 10c_{finished} - 1$
Reward Function 5	$r = (c_{goal}/\|c_{goal}\|) * c_{goal} * c_{goal} + 10c_{finished} - 1$

is that the first episodes of training may go wrong because the knowledge will be not enough and the agent will not be able to adjust its weights in a short time and cannot determine whether those actions are good or bad since exploration will be limited. Furthermore, sampling before training with a low limit may also affect training badly at the beginning since it may end up at a bad local optimum. On the other hand, selecting a very high limit can also affect training in a negative way because the agent cannot be sure if an action is good enough to spend all other useless actions and states. This causes the agent to increase the probability of bad action state pairs or decrease the probability of a good action since the time spent on an episode is too long that the agent cannot calculate the values of the actions accurately.

4.2.4 Strategies

The strategies are tested with rule-based agents, random players, and also human players to find out the best and optimal solution for the agent in a match-3 game.

Color Shuffling

All levels have an initial fixed state and it is special to that level. However, the agent can be biased to initial states and therefore it will memorize those initial states and give more priority to those states than it should be. This may damage training and agent may not work well for levels that they have never seen before. To overcome the problem, a technique that helps give all colors the same priority is needed. This can be solved by shuffling colors which means the first 5 layers of observation are assigned to items according to their color and changing those layers at every episode.

Limiting the Length of Episode

Every level must be finished somehow, whether it is won or lost. For Zombie Blast, if you clear all obstacles, you won. However, if you exceed the number of moves given at the beginning of the level, you lost. Since the aim of this study is to find the minimum number to complete the level, a random number cannot be given to training the agent. Therefore, four different numbers(20, 50, 100, 150) are tested to know the number needed for a level approximately. Furthermore, this experiment shows us the importance of the length of an episode. Longer episodes that should have been completed earlier time may affect training negatively. Shorter episodes can also affect training negatively because the length of the episode is too short for the agent to learn from it.

Hard Masking

The most common problem encountered while creating a better agent is the fact that many actions from action space are not valid to select in a random state because there must be at least 3 same-colored items next to each other and selecting an action that does not result in 3 same-colored consecutive items causes the state not changed after this action. In most states, more than half of the actions are invalid and if the agent chooses invalid actions more and more it would affect training negatively. The negative reward may be large to avoid those invalid actions, but it can also destruct values of weights and a more biased agent would be trained. The action masking is used so that the agent cannot choose an invalid action and choose an action from valid actions at a state. Forcing the agent to choose only valid actions from the set of actions is called hard masking. Hard masking helps to choose better actions and get more rewards in one episode. To mask invalid actions and create a hard mask, the logits of policy distribution are changed where valid actions stay the same and invalid actions are set zero.

Soft Masking

Another masking technique is soft masking. It is similar to hard masking but it allows the agent to select invalid actions. Soft masking provides a new layer in observation input that assigns 1's to valid actions grid tiles and 0's to invalid actions. This helps

the agent choose more valid actions and learn in a shorter time. Also, there is another soft masking called soft masking v2. New action masks can be added according to the color of the item and their valid actions. The number of soft action mask layers for soft masking v2 is five since the number of different colors for items is five. These five layers are added into observation input as well.

CHAPTER 5

EXPERIMENTS

5.1 Dataset

Among the first 25 levels of the game, 4 of them are used for training and 4 are used for testing. As shown in Figure 5.1, on the left, levels 18, 6, 8, and 15 are used for training. On the right, levels 11, 10, 9, and 23 are used for testing. These levels are selected randomly and used in this thesis as the dataset. The number of a level does not mean anything. There is no relation between numbers and difficulty. Higher numbers do not mean harder levels. The difficulty of a level is independent of its number. The numbers are just for notation. To avoid confusion, level numbers are denoted as 0, 1, 2, 3 for training levels and 4, 5, 6, 7 for testing levels.

Figure 5.1: Training and testing levels.

All next charts have the same titles for the X and Y axes. The values of the X-axis

mean the number of episodes passed. The values of the Y-axis mean the average of the last 20 plays' move count of a level. All charts in the experiment section are generated by Weights & Biases [18].

5.2 Advantage Actor Critic vs Proximal Policy Optimization

I tried different values for hyper-parameter tuning to achieve the best performances for A2C and PPO. There are different types of hyperparameters of A2C and PPO. Some of them are mutual. Their meanings and values tested are:

learning_rate: Value that controls the amount of change of weights in the neural network model. Tested values are 1e-03, 1e-04, 5e-05, 1e-05.

nsteps: The number of steps to run for each environment per update. Tested values are 5, 32, 128, 256, 512, 1024, 2048.

gae_lambda: Factor for trade-off of bias vs variance. Tested values are 0.1, 0.3, 0.5, 0.7, 0.9.

ent_coef: Entropy coefficient for the loss calculation. Tested values are 0.001, 0.01, 0.1, 0.

v_coef: Value function coefficient for the loss calculation. Tested values are 0, 0.1, 0.2, 0.5, 0.9.

n_envs: The number of environments. Tested values are 1, 2, 4, 8, 16.

nbatches: Minibatch size. Tested values are 8, 16, 32, 64.

nepochs: Number of epochs when optimizing the surrogate loss. Tested values are 10, 20, 30.

Best performances of the values of hyperparameters are shown in Table 5.1 and Table 5.2. Comparison with best results from hyperparameter tests for A2C and PPO is shown in Figure 5.2. PPO passed A2C on most of the levels. The gap between A2C and PPO is large for levels 1, 2, and 3. The PPO agent can complete levels in fewer moves. Batch size and the number of epochs help the PPO agent update policy

more stable. The Clipping also helps PPO perform better at approximation for policy because huge updates are restricted by the clipping.

Table 5.1: A2C hyperparameter values

	learning_rate	nsteps	gae_lambda	ent_coef	v_coef	n_envs
A2C	5e-05	5	0.5	0	0.5	8

Table 5.2: PPO hyperparameter values

	learning_rate	nsteps	nbatches	nepochs	gae_lambda	ent_coef	v_coef	n_envs
PPO	5e-05	1024	32	20	0.5	0	0.5	8

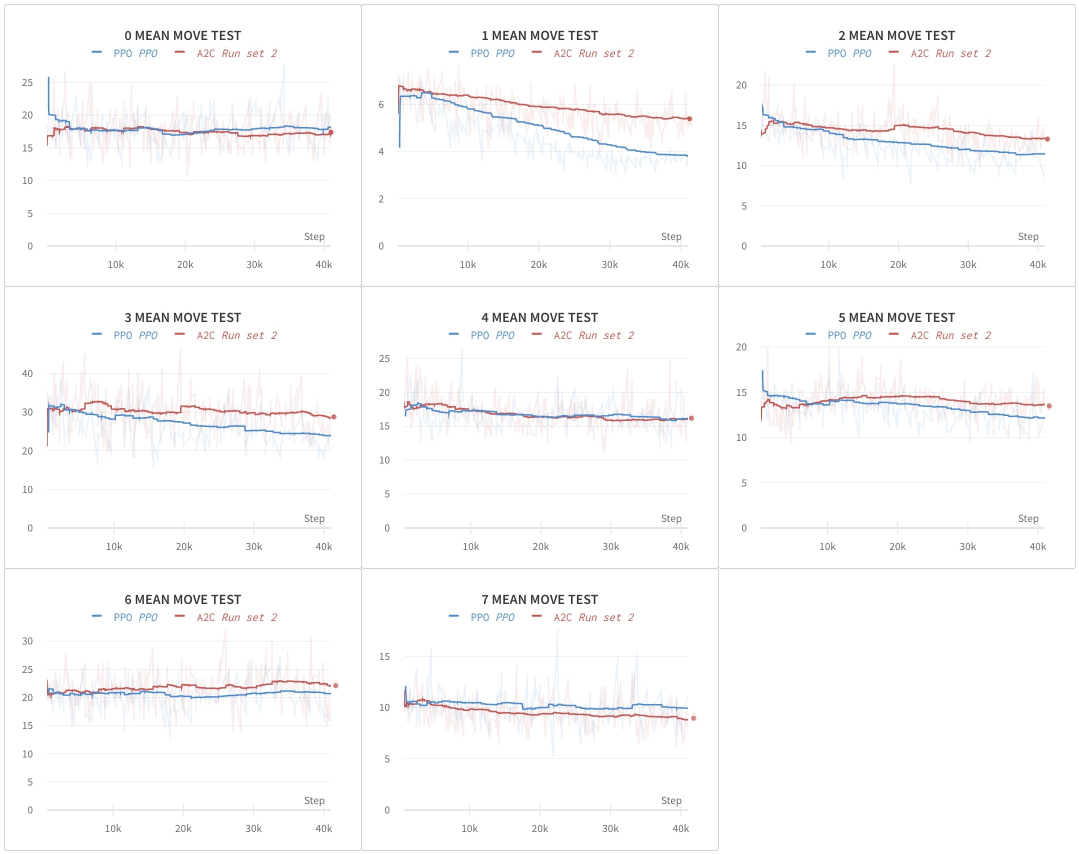

Figure 5.2: A2C vs PPO

5.3 Reward Functions

The importance of immediate and late rewards are tested as seen in Figures 5.3 and Figure 5.4. Creating a fixed reward function was problematic since the number of obstacles is changing with the level. Therefore, the importance of removing obstacles and cumulative reward at the end differs according to the level. This affects training positively because the reward function returns different values according to level. These reward functions are created as immediate rewards dynamically or statically rewarding according to the number of obstacles at a level. Results are shown in Figure 5.3 and Figure 5.4.

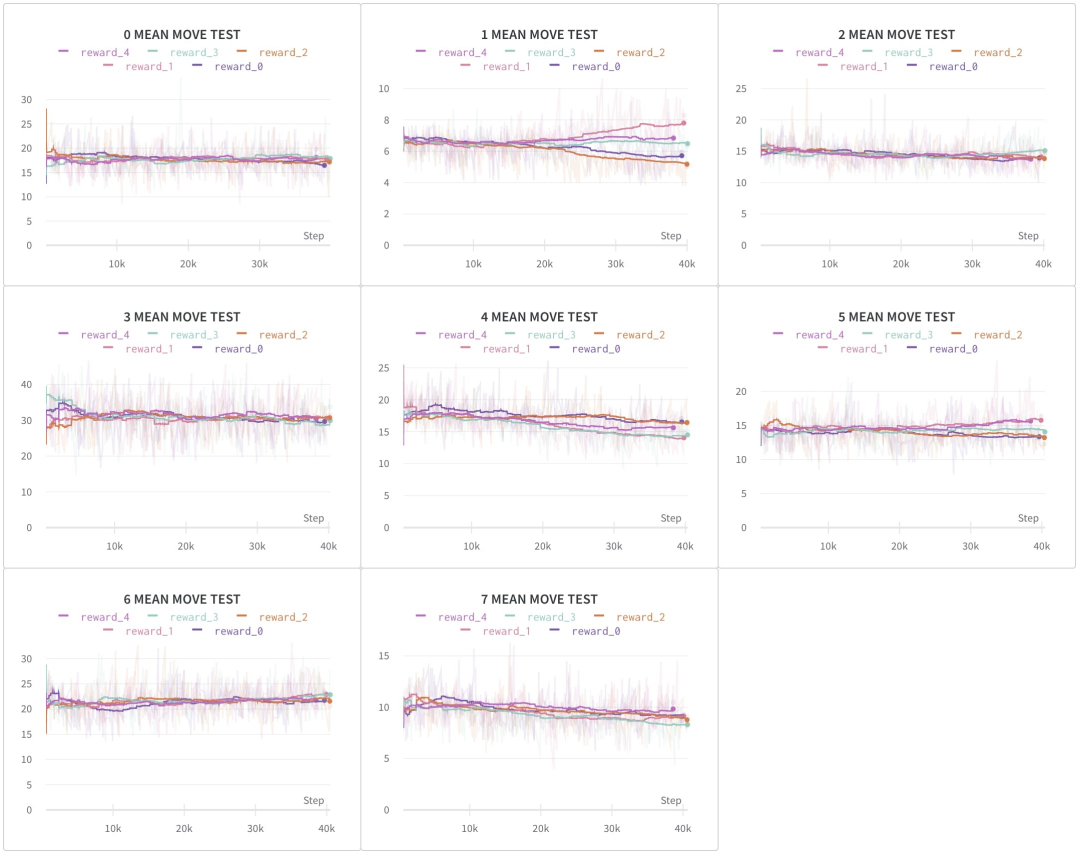

Figure 5.3: Reward function experiment results for A2C.

Reward functions 2 and 3 are the best reward functions for A2C. Reward function 0 is the far best rewarding function for PPO. Other reward functions perform similar results. Reward function 2 is used for A2C and reward function 0 is used for PPO for

the comparison of two reinforcement learning algorithms. These results mean that immediate rewards and rewards at the end of an episode with balanced coefficients give the best performance when the reward function dynamically changes according to the level. The reason is immediate rewards does not help agent complete fewer moves. However, only rewarding at the end is not helpful since it cannot detect good moves better than immediate rewards. Furthermore, adding a small negative reward helps the agent to know the aim of the game is always to finish a level as fast as possible.

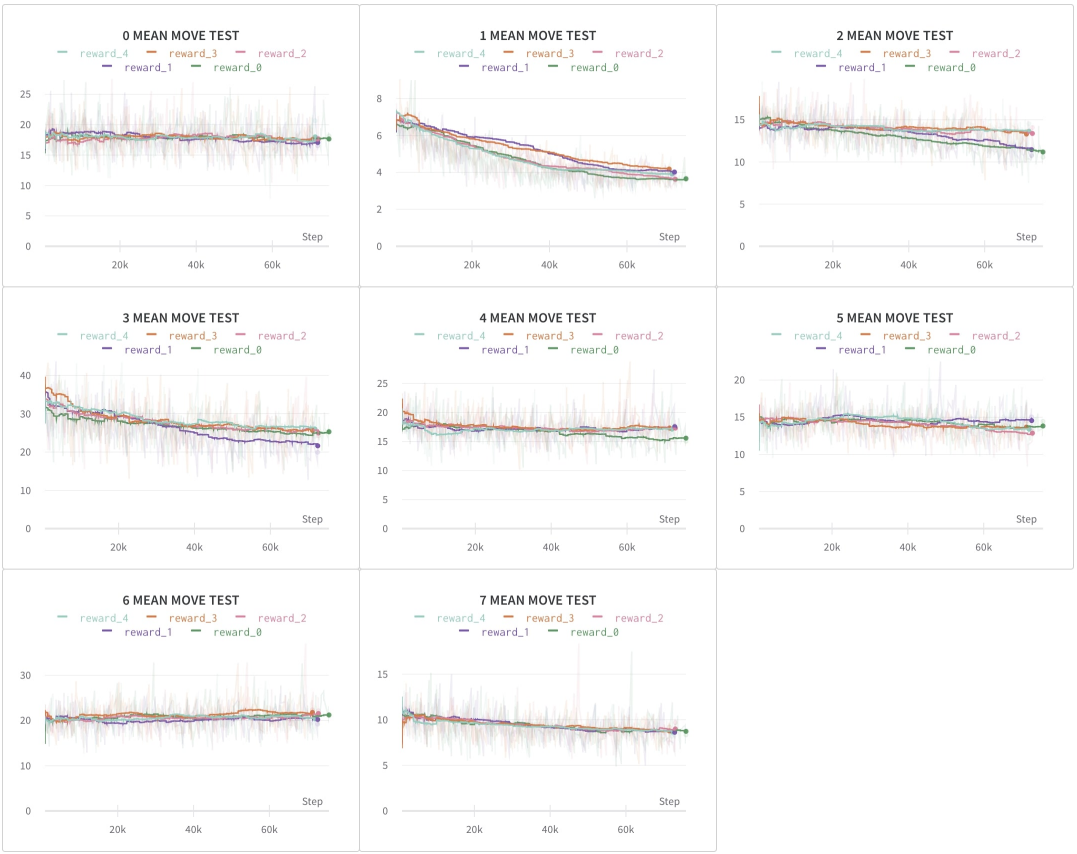

Figure 5.4: Reward function experiment results for PPO.

5.4 Limiting the Length of Episode

While training, the agent first needs to explore the environment and try to make an action. However, it may last longer than average to finish an episode. To be sure that the agent learns in an episode, we need to give it sufficient time to learn and get

rewards from the environment. However, when it is too long the agent may try to extract information from a bad episode since it sticks at doing the same action and it cannot progress to a new state. Different limits are tested for this purpose (20, 50, 100, 150). The limit value is the number of moves allowed in an episode. Results are shown in Figure 5.5 and Figure 5.6.

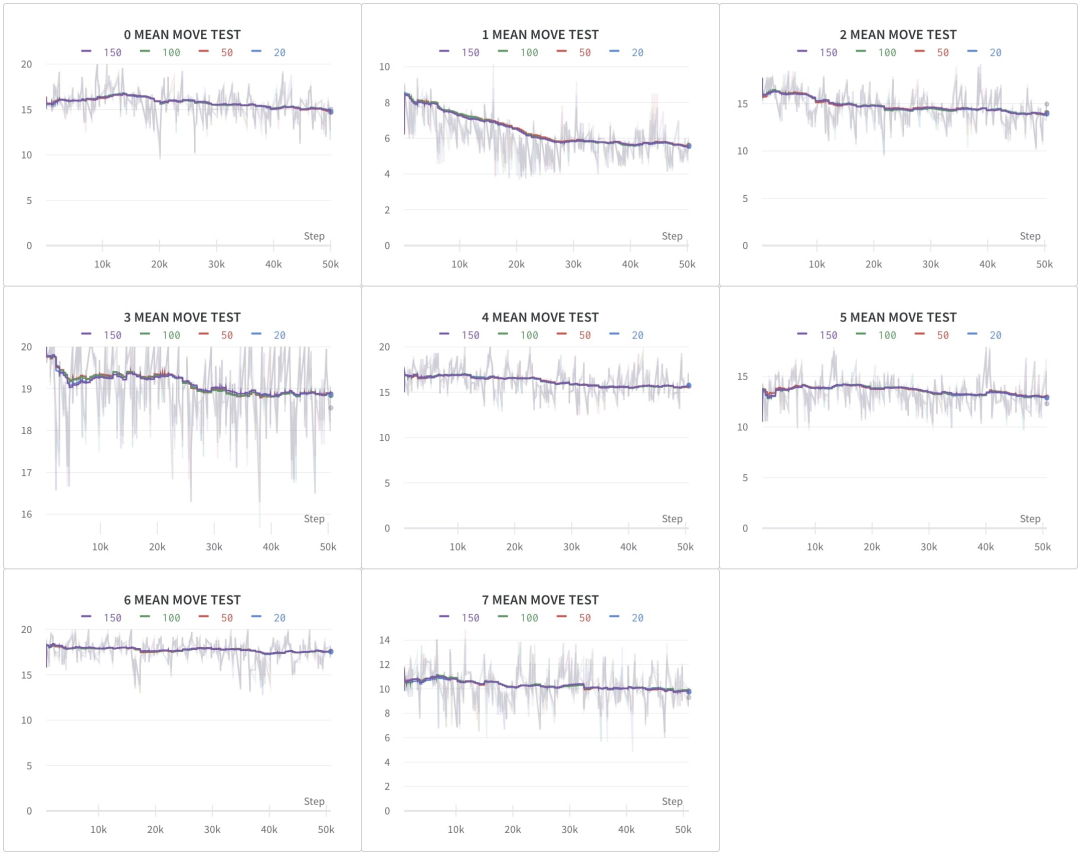

Figure 5.5: Limit experiment results for A2C.

Limiting the length of an episode does not affect the results of A2C. However, PPO works differently at different limits. Limiting an episode with the number of moves equals 20 looks more effective than other values at levels 2, 4, and 5. The reason is not giving the agent permission to exploit bad behavior after the average move is exceeded. Therefore, the agent uses information from only the good episodes. Since the average move of levels is near to 15, 20 performs best from others. A2C performed the same for every limit value. This means increasing the limit does not affect the A2C agent in a negative way.

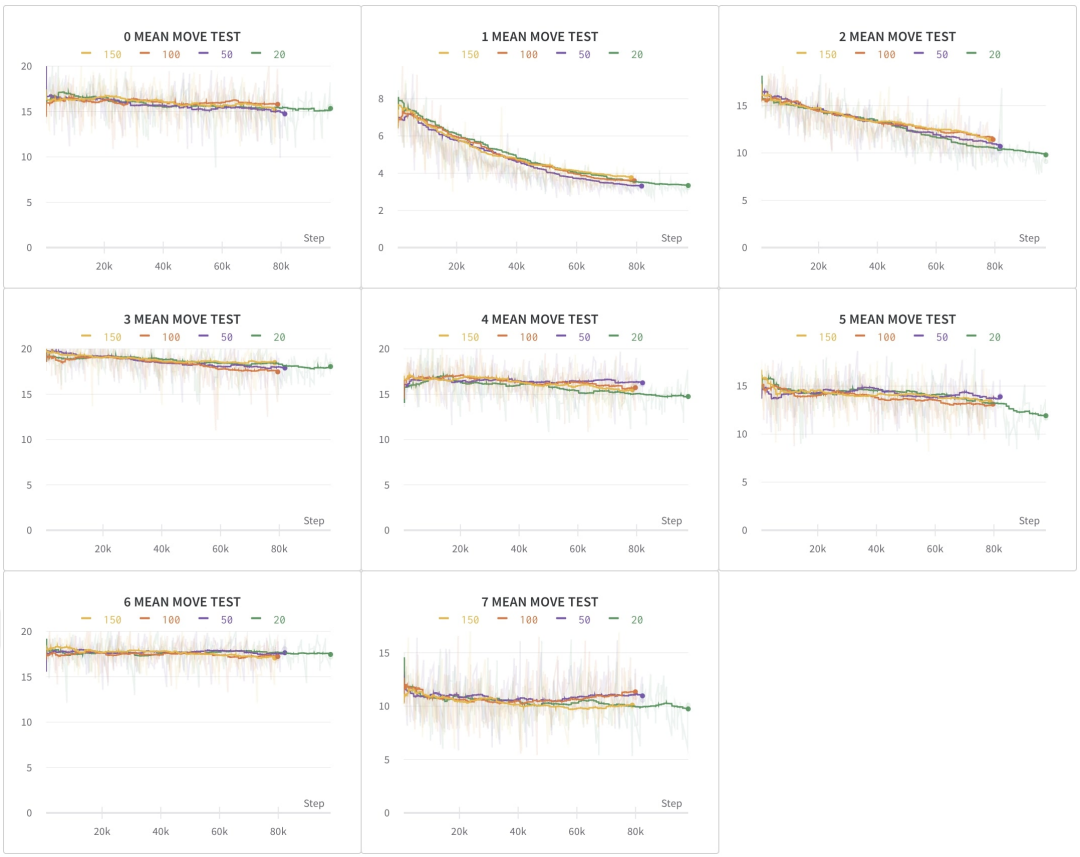

Figure 5.6: Limit experiment results for PPO.

5.5 Color-shuffling

The color-shuffling strategy helps the agent does not prioritize an item by its color. This helps also training to get a more generalized model for different levels. Comparison of color-shuffling is tested with soft-masking on. Results are shown in Figure 5.7 and Figure 5.8.

Whether using color-shuffling or not performed the same results except for levels 1, 5, and 7. Color-shuffling shows levels as they are different levels. Therefore, the agent does not memorize the initial state of a level. The agent uses a more generic approach to perform better at levels that it does not play before. The difference is not large in Figure 5.7 and Figure 5.8 because initial states has a small impact. After two or three moves, new falling items would be different. Hence, color-shuffling does not matter except to prevent the agent from memorizing the initial states.

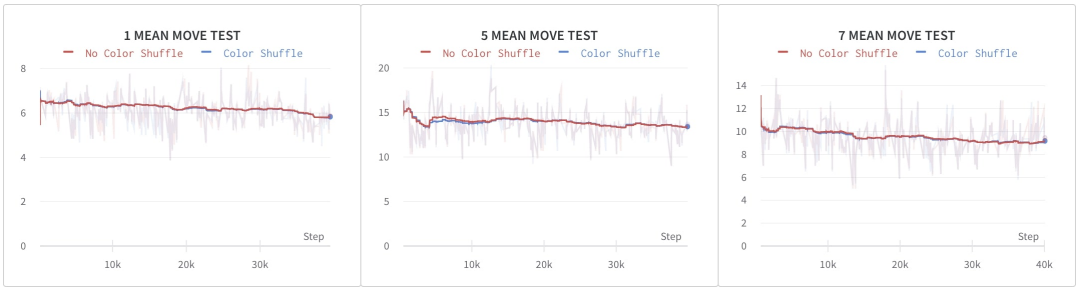

Figure 5.7: Color shuffle experiment results for A2C.

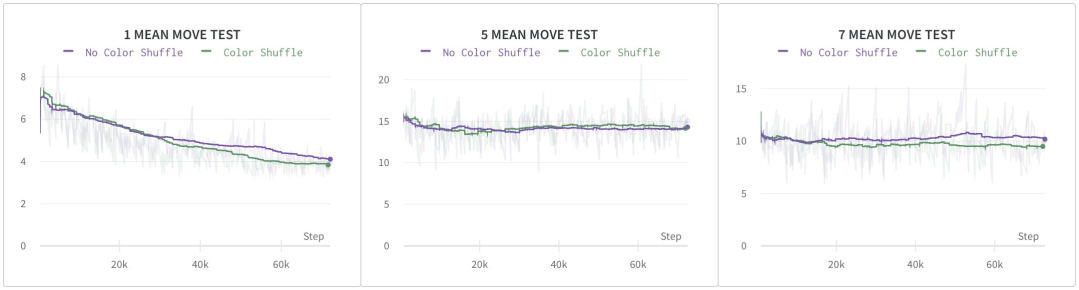

Figure 5.8: Color shuffle experiment results for PPO.

5.6 Masking

Most effective and the core section of this study is adding different masking techniques. There are 3 types of mask techniques used in this study. These are hard-masking, soft-masking and soft-masking-v2 as represented results in Figure 5.9 and Figure 5.10.

- hard-masking: It forces the agent to choose an action from valid ones
- soft-masking: It does not force the agent to choose an action. It gives a piece of information about valid actions and this information is included in the input.
- soft-masking-v2: This has a little difference from soft-masking and that is giving more detailed information about valid actions as 5 layers of each type of item rather than one layer of valid actions.

Hard masking has more episodes in Figure 5.9 and Figure 5.10 due to not allowing invalid actions. It removes steps with invalid actions where step means one iteration of

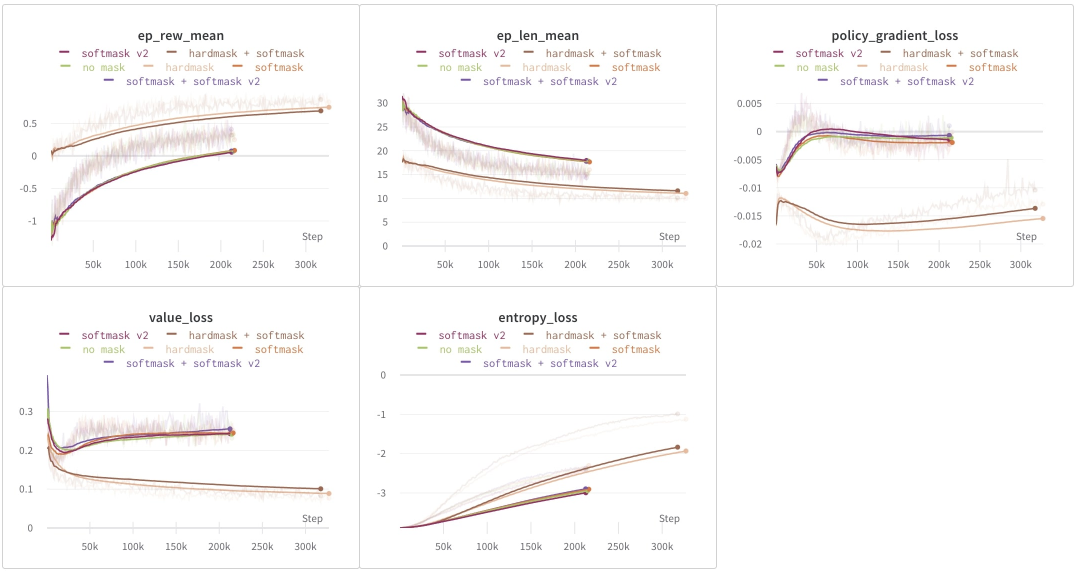

Figure 5.9: Episode reward mean, episode length mean and losses for masking experiments

taking an action and changing the current state. Since invalid actions do not change the state, the state remains the same but an iteration/step passed. Therefore, hard masking agents continue after others finished their training. Graphs in Figure 5.9 and Figure 5.10 show that the number of invalid actions of an episode contains is nearly between a quarter and half of the total actions taken in an episode.

Best results are obtained from hard masking. However, while hard masking affected some levels better, some of them goes bad beyond some point compared to the initial values. This problem may occur because of the designs of levels and small datasets. Hard masking performed better than others for levels 1, 2, 3, and 6. The PPO agent without hard masking may be affected by invalid actions and local approximations can be less accurate. Hard masking does not allow the agent to choose invalid actions and it means the agent chooses better actions on average because there are usually less than half valid actions in a state for this type of game. Furthermore, softmask v2 performs worse than softmask. The reason can be increasing layers in observation would give more details but more details mean a more complex problem. It means making a simple problem a harder one. Soft masking is the second best and gives more general results for every level and it may help to find a more optimum model for the game.

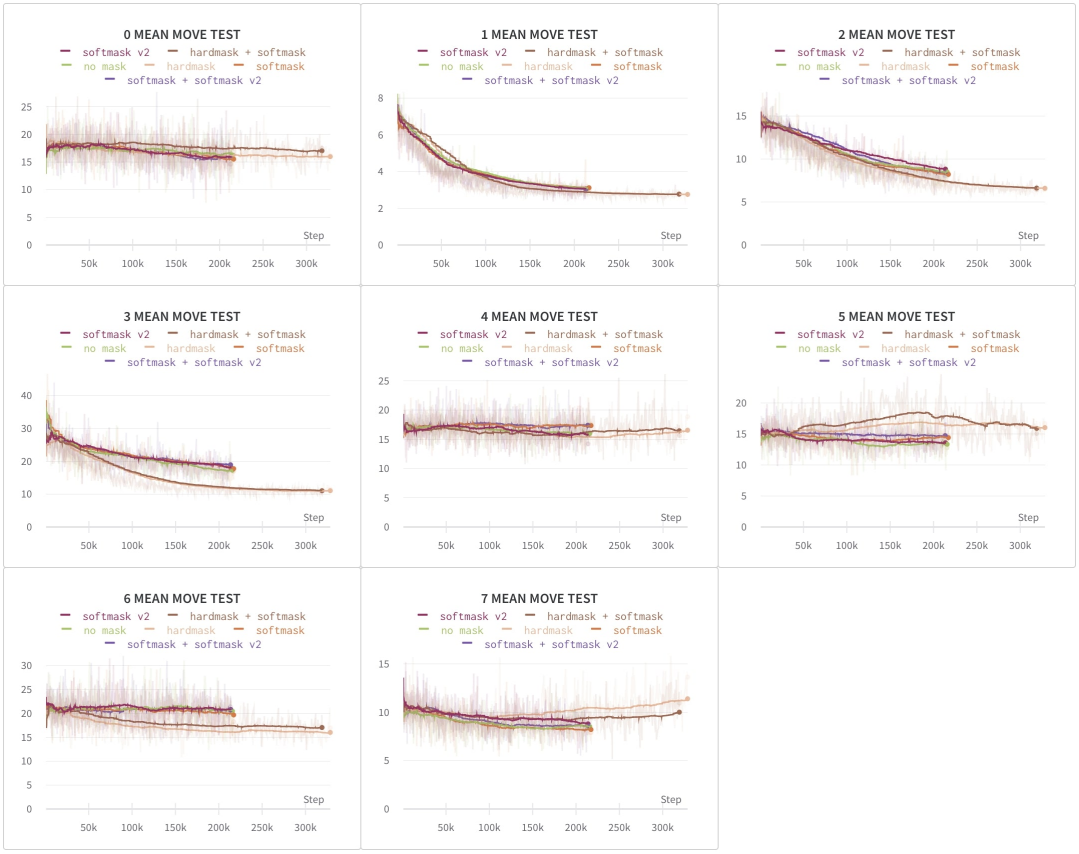

Figure 5.10: Level values for masking experiments

5.7 Human Players

Zombie Blast is put up on the market live a game store. Anonymous information from players is collected to compare the learning agent with the rule-based agent to find out the success of the learning agent by a reliable resource. Six different types of players are evaluated by training and test levels. Players except humans played 10,000 times every level to complete with the given the same number of moves given to human players. Definition of players is listed. Results are shown in Table 5.3.

- **Human First Try:** Players who play the level for the first
- **Human Total:** Players who play the level until they pass or give up
- **Random:** Playing by selecting random actions from the grid
- **Rule-based:** Choosing the match that removed the most tiles

- **Rule-based v2:** Choosing the match that removed the most obstacles
- **RL Agent:** PPO agent using hard-mask with best result hyperparameters

Table 5.3: Comparison of Win-Ratio of Levels

Level	Human First Try	Human Total	Random	Rule-based	Rule-based v2	RL Agent
0	93%	97%	55%	**100%**	99%	87%
1	99%	99%	95%	**100%**	**100%**	**100%**
2	97%	97%	67%	**100%**	**100%**	**100%**
3	57%	90%	7%	46%	59%	**93%**
4	97%	98%	48%	**100%**	**100%**	98%
5	88%	94%	70%	**100%**	**100%**	92%
6	73%	95%	59%	97%	**100%**	95%
7	99%	99%	88%	**100%**	**100%**	99%

Rule-based and rule-based v2 came first at levels 1, 2, 4, 5, and 7. However, other players also reached close to them. The random player only succeeded at levels 1 and 7. The most important level of the table is level 3. Human players succeeded at all levels except level 3 when they played for the first time. Moreover, the random player barely completed level 3. Rule-based and rule-based v2 also did not perform well. The only level they have difficulty with is level 3. Contrary to these players, the RL agent performed better than other players by far. In addition to that, the RL agent passed other levels with a better success rate as well. The reason why the RL agent beat other players at level 3 can be the RL agent learned a better strategy for level 3. First actions may be important than other levels and these first actions help the player complete the level in a short time. If the first actions are bad, it may make the game more difficult to complete. Because some obstacles may not be removed if they have no neighbor to a match and the game lasts longer than usual. The reason why rule-based players came first at many levels can be these levels do not need a strategy to finish in a short time. In other words, choosing the longest match or choosing the match that removed the most obstacles is enough to finish with the optimum number of moves.

CHAPTER 6

CONCLUSION

According to match-3 games are learned by reinforcement learning algorithms to beat human players. There are different types of match-3 games. One is swiping one item with another to match three same items in horizontal or vertical. Another one is selecting an item and dragging neighbor items that have the same color as the first selected one. The last one is tapping one item and neighbor items except for corner ones with the same color. Studies generally focused on the first type of game since it is the classic one and it is very simple to create a well-defined environment for them. In this study, a different kind of match-3 game has been tested with reinforcement learning algorithms. A2C and PPO are the most used algorithms for match-3 games in the literature. We used these algorithms in this study by adding a level generation mechanism to create a more autonomous system. Furthermore, match-3 game studies usually try vanilla reinforcement learning algorithms and compare them for the problem. In this study, some strategies are added and tested to increase the success of the agent and decrease the time spent on learning. These strategies are action masking and color shuffling. Action masking has three different types. The first one is hard masking that forces the agent to select only valid actions. Because half of the actions are generally invalid and the probability of selecting a valid action is low, learning can be hard and take a long time to train. Hard masking is a great option to remove invalid actions and make an agent focus only on valid actions. Another one is soft masking which is included in the observation space as input and does not force the agent to select valid actions but gives a piece of information to the agent like a map. The last masking strategy is using soft masking by adding all valid actions connected layer by layer and dividing them by colors. This gives more information about valid actions and relations of actions. Color shuffling is another strategy that avoids giving

more priority to some colors based on the initial states of levels.

Experiments show that the PPO agent beat the A2C agent in every strategy. The result of the A2C agent is ineffective on strategies limiting the length of an episode and color shuffling. The result of the PPO agent is effective for these strategies but not much. Limiting the length of an episode with the number of moves equals 20 looks most effective value. The reason is making the agent avoid bad behavior because the average move is exceeded and no need to continue to learn from the episode. Therefore, the agent uses information from only the good episodes. Color shuffling helps the agent to get a more generalized solution for different levels. There is little performance improvement because only the initial state is affected by color shuffling. Furthermore, reward function experiments show that a reward function with balanced coefficients gives the best performance when the reward function dynamically changes according to the level. The best performance improvement is obtained from masking experiments. Hard masking makes a big difference compared to no masking and soft masking. It makes the agent avoid taking an invalid action. Hence, the agent learns from only valid actions and evaluates its policy by results from only valid actions.

To better understand how well the agent works, the best RL agent is compared to other players who are human players, the random player, rule-based agents. The overall success of the RL agent for all levels is better than the other players especially for one level that probably needs a better strategy than straightforward logic. However, rule-based agents usually came first at many levels. That's because many levels can be completed by using a simple logic such as choosing the longest match or selecting a match that clears the most obstacles at a current state. Since the aim of this study is to provide a more reliable system to create a game solver, the RL agent provides this system than rule-based players. Because when rule-based players failed at a level that humans succeeded, the RL agent can also succeed at that level as humans did.

This study aims to develop a learning agent for match-3 games by automating level generation. The study also aims to use generated levels at production in Zombie Blast and place these levels after setting their difficulty. This thesis can be used as a guide for people doing research on level design and difficulty adjustment for match-3

games. For future work, more different obstacles can be added to teach the agent to learn and optimize the game to train the agent more efficiently. Moreover, more generic match-3 games systems can be implemented results in a tool that generates levels and gives information about the difficulty of the level. This tool can be very helpful in the game industry.

REFERENCES

[1] F. Richter, "Infographic: Gaming: The most lucrative entertainment industry by far," Sep 2020.

[2] I. Kamaldinov and I. Makarov, "Deep reinforcement learning in match-3 game," *2019 IEEE Conference on Games (CoG)*, 2019.

[3] F. Mourato, M. P. D. Santos, and F. Birra, "Automatic level generation for platform videogames using genetic algorithms," *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology - ACE 11*, 2011.

[4] 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, p. 51593–51600, 2020.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, and et al., "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, p. 1140–1144, 2018.

[6] J. T. Kristensen and P. Burelli, "Strategies for using proximal policy optimization in mobile puzzle games," *International Conference on the Foundations of Digital Games*, 2020.

[7] K. D. Jong, "Adaptive system design: A genetic approach," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 10, no. 9, p. 566–574, 1980.

[8] J. R. Sampson, "Adaptation in natural and artificial systems (john h. holland)," *SIAM Review*, vol. 18, no. 3, p. 529–530, 1976.

[9] X.-S. Yang, "Genetic algorithms," *Nature-Inspired Optimization Algorithms*, p. 77–87, 2014.

[10] "What is machine learning? - i school online," Jan 2021.

[11] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems 12*, vol. 12, pp. 1057–1063, MIT Press, 2000.

[12] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust region policy optimization.," in *ICML* (F. R. Bach and D. M. Blei, eds.), vol. 37 of *JMLR Workshop and Conference Proceedings*, pp. 1889–1897, JMLR.org, 2015.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms.," *CoRR*, vol. abs/1707.06347, 2017.

[14] S. M. Kakade and J. Langford, "Approximately optimal approximate reinforcement learning.," in *ICML* (C. Sammut and A. G. Hoffmann, eds.), pp. 267–274, Morgan Kaufmann, 2002.

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016. cite arxiv:1602.01783.

[16] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3." `https://github.com/DLR-RM/stable-baselines3`, 2019.

[17] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. cite arxiv:1606.01540.

[18] "Weights and biases."