Data Collection Using Deep Reinforcement Learning for Serious Games

By

Andrew Kozar

A Thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfilment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg

Abstract

Mild Cognitive Impairment (MCI) often occurs prior to the more serious condition of dementia and early detection of MCI is an important but challenging task because of its indistinct symptoms. Work has been done developing serious games on mobile devices for MCI detection as opposed to a typical application of serious games for growing and maintaining mental acuity. The serious games WarCAT and Locker record player's moves made while playing the game to determine their levels of strategy recognition and learning. To be able to demonstrate this, however, requires a large amount of player data. Therefore, it would be beneficial to develop a method of generating synthetic data that could imitate human player data. The area of machine learning (ML) known as Reinforcement Learning (RL) can be applied to creating a large pool of players since it emulates the way humans learn. In RL, if an action in response to a stimulus is followed by a successful reward, the stimulus-action-reward association will be strengthened, and the reward will be recalled with greater likelihood upon later presentation of the same stimulus and action. Like RL in humans, considerable trial and error (training) is often required. In addition, a growing subfield of machine learning known as Deep Reinforcement Learning uses techniques of Reinforcement Learning along with Artificial Neural Networks for function approximation. The purpose of this thesis is to explore the use of Deep RL to learn to play our serious game and achieve gameplay results comparable to the best human performance. From this, we can define baselines which allow us to create bots with various levels of training to emulate individuals at various stages of learning, or by extension, various levels of cognitive decline.

Acknowledgements

I would like to thank Bob McLeod and Marcia Friesen for allowing me to join their group, for being great advisers and supporting me throughout the work on this thesis. I would also like to thank the professors who supported me including Ken Ferens, Witold Kinsner, and Ahmed Ashraf. I also want to thank my lab mates who were also there to help me and spent time with me. Finally, I would like to thank the University of Manitoba for the support through this master's degree.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

With a rapidly aging population, we face a greater challenge when it comes to predicting impending cognitive impairment early. The global population aged 60 years or over numbered 962 million in 2017, more than twice as large as in 1980 when there were 382 million older persons worldwide. The number of older persons is expected to double again by 2050, when it is projected to reach nearly 2.1 billion [1]. While a few decades ago it was assumed to be sufficient to distinguish dementia from normal cognitive aging, recent years have revealed the desirability of more accurate early diagnostic decisions. The clinical spectrum of dementia has expanded to include mild cognitive impairment (MCI) and finally to preclinical Alzheimer's Disease. In this case, although people are cognitively normal, they have the underlying features of Alzheimer's Disease which makes them susceptible to the disease. One-third (35%) of primary care physicians are not fully comfortable diagnosing MCI, and more than one-half (51%) are not fully comfortable diagnosing MCI due to Alzheimer's disease [2]. Clinicians are now in a difficult but advantageous position to be able to recognize the very early clinical signs of impending disease with the assistance of Machine Learning (ML) techniques. MCI is usually considered a transitional state between age-related cognitive decline and very early dementia, but despite its conceptual validity, the term remains challenging to clinical practice.

For example, in an elderly patient population, the following reflects a typical clinician presentation. A patient is more forgetful than before and is concerned about possible problems that cannot be explained through aging. Their memory problem seems to have worsened in

recent months or years. They appear to have no other known medical conditions that could affect their condition.

MCI is prevalent among the elderly population, where approximately 10% to 20% of people age 65 or older live with MCI [3]. Estimates have indicated that 12.23 million people would have MCI due to all causes in 2020, which will increase to 21.55 million in 2060 [4]. Furthermore, a worrying statistic is that an estimated 8% to 15% of individuals living with MCI develop dementia each year [5] . Additionally, one-third of people living with MCI due to Alzheimer's disease develop dementia within five years [2]. According to World Health Organization's 2022 report, the estimated number of people with dementia is 55 million worldwide, and there are nearly 10 million new cases each year [6]. In 2019, the estimated total global societal cost of dementia was US $1.3 trillion, and these costs are expected to surpass US $2.8 trillion by 2030 as both the number of people living with dementia and care costs increase.

However, despite these statistics numerous studies show that not all MCI cases lead to dementia. There are defined 3 possible outcomes of MCI: worsening condition (progression to dementia or severe cognitive impairment), improvement (revert to normal cognition), and stability (unchanged cognitive status). Population studies have shown smaller proportions of participants with MCI progressing to worse conditions, larger proportions of participants improved or reverted to normal, and most participants remained stable over the duration of these studies [7] . The same pattern can be found in clinical studies. Most subjects remain stable over the duration of the study, fewer progress to dementia while the smallest group improves to a normal cognition level [8] [9]. As with other numerous health conditions, if MCI can be identified early it is possible to stop or slow its progression through various interventions to enhance or preserve

cognitive capacities. Some studies have shown evidence that pharmacological treatments have been ineffective at treatment of MCI [10]. There is promise for newly FDA approved therapeutic aducanumab, however it has shown no cognitive benefit in phase III trials [11]. Conversely, behavioral therapies have shown small benefits and lifestyle factors have been deemed important. Research reported at the 2019 Alzheimer's Association International Conference [12] suggests that adopting multiple healthy lifestyle choices, including healthy diet, not smoking, regular exercise and cognitive stimulation may decrease the risk of cognitive decline and dementia. Given these facts, it is apparent that early detection of MCI would be critical in mitigating further issues related to a patient's worsening cognitive state.

Cognitive screening tests, also known as neuropsychological testing, are the most used techniques for identifying MCI. Typically, the person being examined is asked to respond to questions or complete tasks and a qualified practitioner conducts the exams. The Mini-Mental State Examination (MMSE) is the most widely evaluated screening test although there are others tests such as the Clock Drawing Test (CDT), the Montreal Cognitive Assessment (MoCA), the Informant Questionnaire on Cognitive Decline in the Elderly (IQCODE), Computer Assessment of Mild Cognitive Impairment (CAMCI), The Modified Telephone Interview for Cognitive Status (TICS-M), and others. Some of the screening tests were designed to detect dementia as well as MCI. A comparison of the performance of the well-known tests used for MCI is shown in Table 1 below.

Table 1 A Comparison of Different Well-known MCI Screening Tests [13]

Test Name	**Sensitivity***	**Specificity***
Mini-Mental State Examination (MMSE)	45% to 82%	65% to 90%
Clock Drawing Test (CDT)	43% to 76%	49% to 83%
The Modified Telephone Interview for Cognitive Status (TICS-M)	47% to 82%	77% to 100%
Informant Questionnaire on Cognitive Decline in the Elderly (IQCODE)	71.1% to 82.6%	69.0% to 83.0%
Montreal Cognitive Assessment (MoCA)	80% to 100%	50% to 76%
Mini-Cog	39% to 84%	73% to 87.9%

* These screening tests tend to have different performances in different studies. The range of values is an estimate of were most studies' values lie. Outlier values were excluded from this range.

However, in order to effectively perform the screening test and avoid interfering with the screening procedure, it is typically demanded that whoever is administering them be trained or at least follow specified guidelines. Administrators of the MMSE are instructed, for example, to refrain from emotional reactions that can impair cooperation and performance [14]. In addition, these tests lack entertainment value and are generally difficult to access on smartphones outside of the MoCA test. They are also not adaptable to scaling or complex mathematical operations. There is potential for serious games to alleviate the drawbacks of cognitive screening tests. Therefore, we propose that player data collected from serious games can be a more helpful diagnostic tool which can be used to help with early detection of MCI.

1.2 Serious Games

Games that are intended more for players' education, training, or information than for pure entertainment are called serious games [15]. They can be used to teach people new skills, knowledge, and attitudes in a range of contexts, such as schools, corporate training programs, and healthcare settings, in a way that is more engaging and interactive.

To entice players and maintain their interest, serious games frequently employ gameplay-based mechanics and elements like points, levels, and challenges. They can be made to run on different hardware, including as desktops, mobile devices, and virtual reality headsets.

Serious games can be used to teach concepts in history or science, train personnel in new techniques, or encourage healthy habits and lifestyles. Other examples of serious games are instructional games that teach historical or scientific themes. Because they encourage immersion and let players practice skills in a secure setting, serious games can be successful at encouraging learning and behavior change.

While there are other games on the market that advertise "brain training" to maintain cognitive function or possibly delay MCI, little research has been done on games that try to detect or assess MCI, which is a subtle but crucial distinction. Serious games can be an abundant source of information for mental health [16] [17]. The development and analysis of serious games for cognitive health has generated a lot of research [18] [19]. However, the use of serious games for a variety of real-world health applications is still in its infancy and largely informal despite some efforts to more formally structure the field [20].

A platform is necessary to act as a conduit between the user and the detection software system in order to build any technique that can identify MCI. Such a platform would transform the player's decisions and actions into raw data. Since it allows for the conversion of the player's performance (moves and decisions made) into numerical input data, a serious game can be employed as such a platform. The performance of certain cognitive functions would be integrated in the gameplay data if the performance of the chosen game depends on them, giving the detection software systems a chance to identify them. For this purpose, the serious games WarCAT and Locker are utilized.

1.3 WarCAT

The serious game WarCAT is a card game that has been designed specifically to serve as a platform for developing models for potentially detecting MCI [21]. It is relatively straightforward and is a modified version of the well-known card game WAR. There are different variations of WarCAT, but in this version a human player is dealt five cards that can be played in the order of their choice against an opponent bot or computer that is playing using a set strategy such as always playing low to high cards, or high to low cards. Figure 1 illustrates an instance of WarCAT.

Figure 1 Screenshot of WarCAT

1.3.1 WarCAT Description

The game rules are described below:

- 5 cards are dealt to both the player and an opponent.
- Both play one card at a time at the same time. That is, both players pick a card and then both cards are revealed at the same time.
- The player can play their cards in any order.
- The bot plays in a specific order (a strategy).
- Each card has the value of its number. Jack, Queen, King, and Ace cards are valued at 11, 12, 13, and 14 respectively.

- The higher card wins that round (hand).
- The winner of each round (hand) gets a score value per the following equation:

$$Score = 13 - (Winning\ Card\ Value - Losing\ Card\ Value) \quad (1)$$

- If there is a tie, the score for that play is 0.
- The game is won by the one with the higher aggregate score value after all five rounds as shown in Equation (*2*) below.

$$Aggregate\ Score = \sum_{i=1}^{5} Score_i \quad (2)$$

In the traditional card game of WAR, the highest played card would win. However, in the variation (WarCAT), the score is calculated in a way to evoke greater diversity of strategy. The scoring is more heavily weighted to a narrow victory over a larger difference in card values. In this manner, according to Equation (*1*) above, an Ace card beating a King card gives a score of 12 while an Ace card beating a Two card gives a score of 1. The first case yielded a higher score because the winning play was with a narrower margin.

The opponent bot always plays the same order of its given cards. The order in which the five cards are played is called a strategy. An example of a strategy would be playing the highest card first, then the second highest, then the third highest, then the fourth highest, and lastly the lowest card. At the beginning, the player has no idea what the opponent bot's strategy will be. In the first game or the first few games, an observant player will detect or discover the opponent bot's

strategy by recognizing that a strategy is being played, and then learning what the strategy is. Then, the player can predict the order in which the bot will play its cards for the duration of the level. Therefore, the player can play their cards in a way that maximizes their points in order to win the game. That is, the player would counter the bot's strategy. It is important to note that the player can predict or learn the order of the bot's cards but not the card's exact values. One example of countering a strategy is given as follows. If the opponent bot's strategy is playing their five cards high to low, the player can counter that strategy by playing their lowest card first against the opponent bot's highest card to see what the bot's highest card was. Then, the player's strategy might be playing the remaining four cards high to low so that the player's highest card would go against the opponent bot's second highest card and so forth, as illustrated in Figure 2 below.

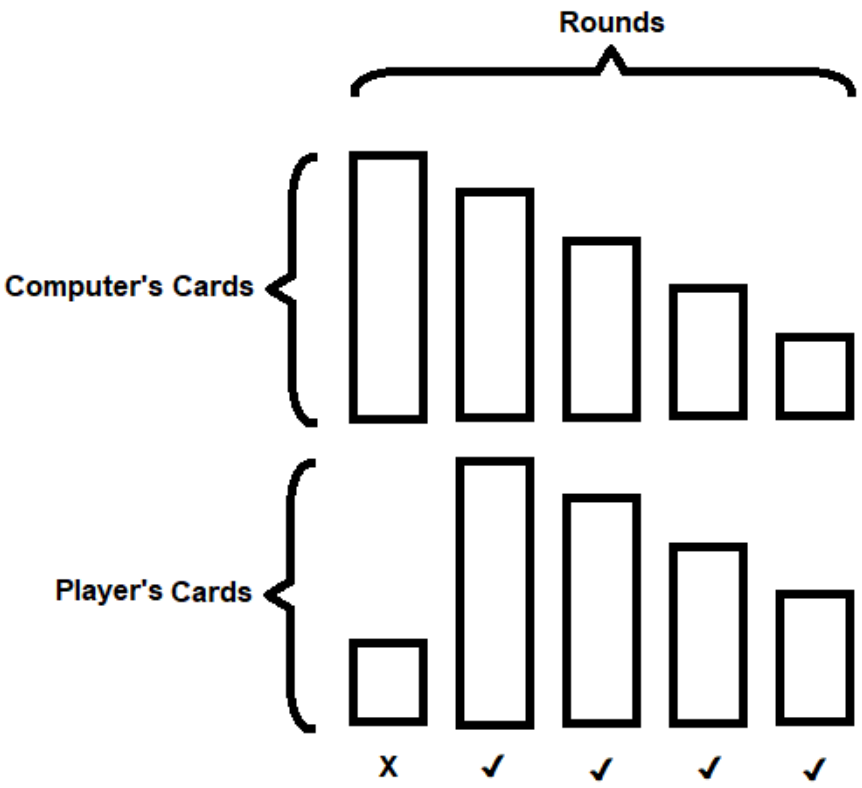

Figure 2 An Example of a Player Countering a Known Strategy

It is not necessarily true that the player's highest card would always be higher than the opponent's second highest card as the cards are dealt randomly. However, after many games played on average the player would win more often if they were able to correctly learn the opponent's strategy. There are other ways to counter-play, some of which might be adaptive and depend on the cards dealt and/or the opponent bot's previous played card in the moment of the game. Those other ways of nuanced counter-play might be expressive of the player's cognitive performance.

An example of a gameplay is shown in Figure 3. In Figure 3(a), each player gets five cards, then the player picks a card out of five to play. In Figure 3(b), the player picks the first card and plays it, the first card played by the bot is then revealed. The one with the highest card wins that round. In the example shown, the bot wins the first round. The bot's card of Queen is valued at 12, and player's card is valued at 3; therefore, according to Equation (*1*), the bot's number of points from the first round is 4. In Figure 3(c), the player wins the second round with 10 points. The player ties in rounds 3 and 4 where neither the player nor the bot gets points. The player wins round 5 which end up with a total cumulative score of 18. Thus, the player wins that game over the bot with a total score of 18 to 4.

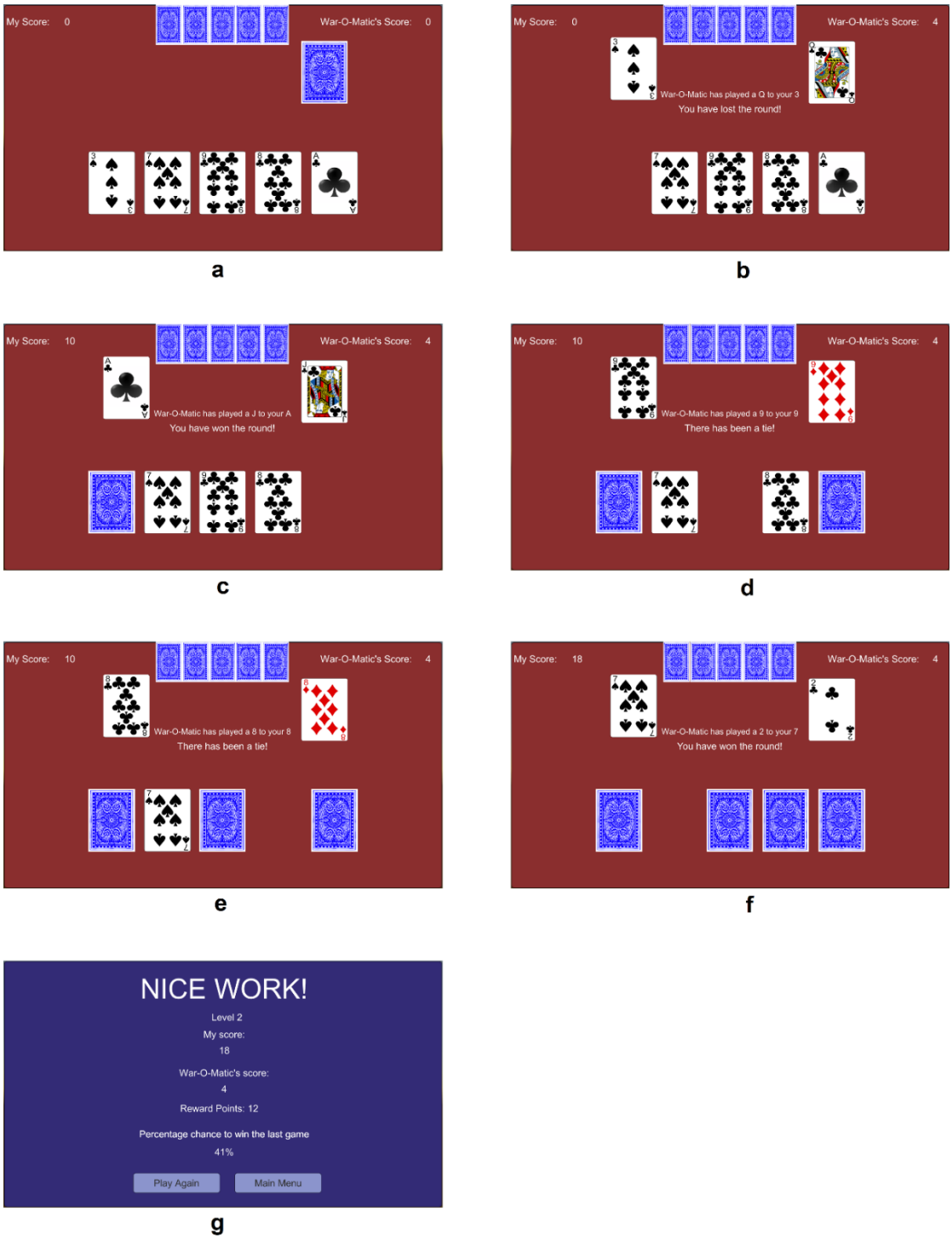

Figure 3 An Example of WarCAT Gameplay

1.3.2 WarCAT Characteristics

Playing any game to be potentially used in the detection of MCI needs to utilize the same mental functions that are affected by MCI. WarCAT encompasses several executive functions of interest when potentially detecting MCI, including strategy recognition, strategy development (learning), memory, recall and retention (remembering & retaining the opponent's strategy once it gets discovered), thinking (planning the strategy that counter the opponent's), and judgement (choosing the best counter play based on the available cards).

It is possible that there are other illnesses that might affect some or all of these cognitive faculties which would get detected or flagged as well in the gameplay data such as Attention Deficit Hyperactivity Disorder (ADHD), Early Alzheimer's Disease (AD), Early-Onset MCI, amnesia as a result of non-aging illness, and others. WarCAT can be one tool in a larger set of approaches to potentially detect MCI, because other factors such as age, severity of cognitive decline in terms of daily life, and medical history are also important factors that a series game may or may not be able to capture well or at all. It is possible that gameplay of this type may also be useful as a means of tracking a person's cognitive decline over the long term such as a component of a longitudinal study.

WarCAT was designed to be accessible to a wide demographic. It can be played on smartphones, tablets, and computers. It does not have a language barrier, although there might be some merits to utilizing language. The game rules are simple and easy to learn in a short time. It attempts to strike a balance between being engaging and complex yet being simple and able to provide detailed player data.

In the initial development of the game, upon signing up to play WarCAT, the player is asked for their age as it is an essential factor to MCI. The player also has the option to answer a question about their sex to preserve the chance for further analysis of MCI risk and prevalence in different sexes, as men appear to be at higher risk than women in having MCI [22]. Also, in the initial versions of the game, the time took by the player to play each card was recorded for future possible further work where the time might refer to the mental effort put during the play.

What differentiates WarCAT from other current screening tests is that WarCAT provides a numerical or quantitative pathway to the rich world of mathematical algorithms to be tested and implemented. This thesis is an attempt at exploring this rich potential. Also, the numerical nature of this game makes it able to convey the cognitive state in a dense numerical representation. For example, tracking decisions or hunches where a player may play a different card with a close value hedging on a close victory. It is also conjectured here that machine learning types of algorithms may be the only practical means of unravelling the ambiguous data associated with gameplay.

WarCAT gameplay gives a role for randomness as any strategy is in reference to the order of cards dealt. For example, the highest card in a hand at any given time may not always be an Ace or King. This randomness factor makes WarCAT a suitable platform as it ensures that ML models built on top are resilient to some amount of randomness (noise). The random nature of the game is likely also a useful artifact as opposed to a truly deterministic game in terms of engaging ones' cognitive processes.

1.4 Locker

The serious game Locker is a lock picking game that has been designed to serve as a platform for developing models for potentially detecting MCI, like WarCAT [21]. The purpose of the game is to unlock the lock in the fewest tries possible when the player does not know the correct combination. However, instead of simply guessing by trying every lock combination, feedback given in the form of a score to let the user know how close they are to unlocking. Therefore, the user can know how close or far away they are from unlocking and develop a strategy for efficiently unlocking. The Figure 4 below shows an example game, where the person started with a reasonably high score and guided by their strategy, proceeded to open the lock in 12 tries.

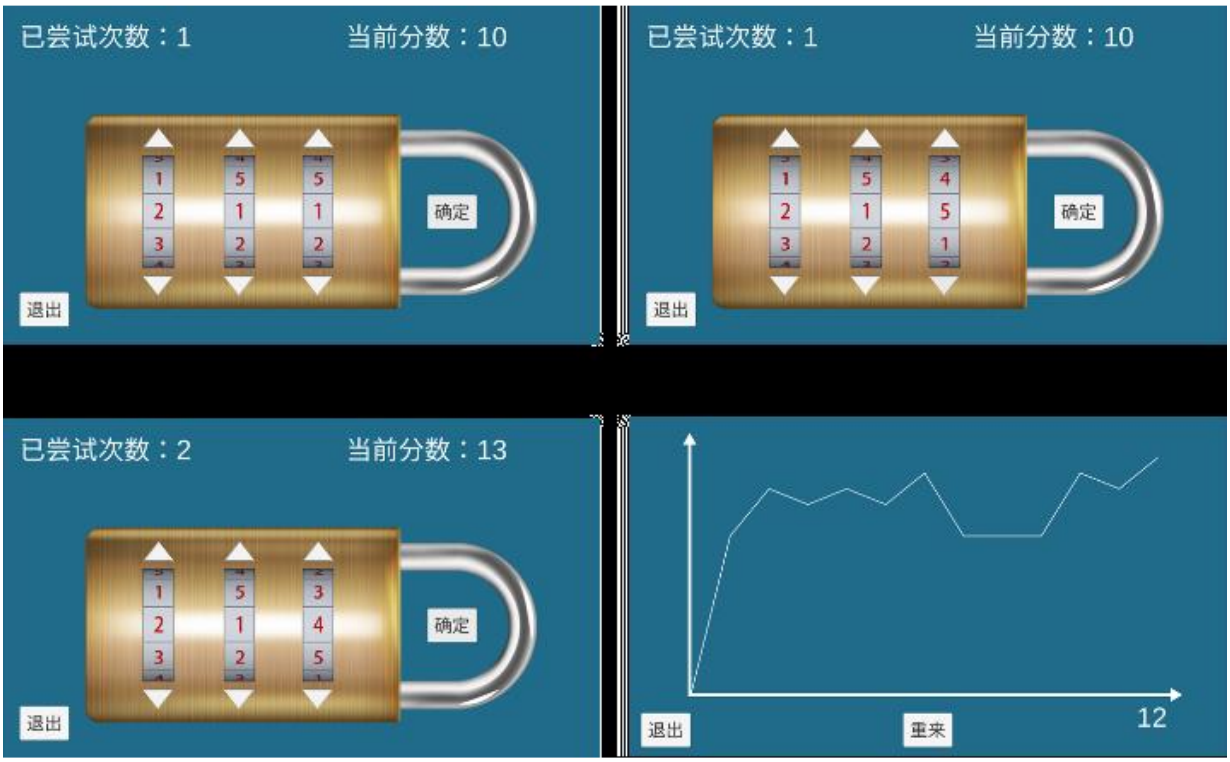

Figure 4 An Example of Locker Gameplay

1.4.1 Locker Description

The game rules are described below:

- There are 3 dials on the locks, with values ranging from 1 to 5.
- The unlocking combination is randomly determined at the start of the game and does not change throughout the game.
- To unlock, a score of 15 is needed.
- The values on the dials can be adjusted however the user wishes before attempting an unlock.
- Each time the user attempts an unlock, their score is updated to let them know how close they are to unlocking and their number of tries is incremented.
- The purpose of the game is to unlock the lock with the lowest number of tries possible.

There are several different strategies a player may employ to get the lowest average number of tries to unlock the lock. Luck will play a part in this game as well, since it is possible to guess the combination correctly on your first try. However, due to feedback in the form of the score it is also possible to have a better strategy than to try every single lock combination. In this configuration there are 125 possible combinations, so it is easily possible to unlock in less than that many tries.

The optimal strategy that is expected from a human player would be the use of a combination of gradient ascent enhanced with some degree of backtracking based on the player's memory of previous rewards and states visited. First, the player would select a random combination to start. Regardless of their initial score, they would then likely increment or decrement locks and

observe how their score changes. Then, based on this information and remembering what combinations they have already tried will eventually discover the unlocking combination.

1.4.2 Locker Characteristics

Beyond memory and recall, problem-solving is one of the cornerstones of higher-level cognitive functioning. The process of searching for an optimal or near-optimal solution is the problem under consideration. In the case of Locker, the search problem of interest is like a guided search. For example, when searching for a word in a dictionary, very few people start at the beginning and proceed until they find the word they are looking for. More typically, people try to home in on a solution guided by rewards, hunches, and heuristics such as divide and conquer. This is because there is feedback in the dictionary in the form of an agreed-upon ordering of characters such that someone can tell how close they are to finding their word in the dictionary. Although Locker is simply a search-based game like looking up a word in the dictionary, it may find a more receptive audience if it is considered a hunt for something or has an aesthetically pleasing interface.

Similar to MCI assessment data in WarCAT, a sequence of plays in the Locker application provides a fingerprint of a person's ability to reason, develop strategy, and retain or recall strategy which is amenable to machine learning. As the selection of attempted lock combinations and a person's score are tracked, the data will contain instances of confusion as well as instances of systematic or algorithmic play.

Like WarCAT, the Locker game also carries the potential to gather player metadata across many players on each move made. This can be used in machine learning approaches to cluster and classify player strategies. Data analysis can focus on measures of cognitive functions within the

domains of attention and memory, assessing the extent to which a player appears to be choosing and then maintaining a strategy in their play.

As an additional note regarding serious games such as these, they can be quite easily made language agnostic. For example, in Figure 4 the Locker game is in Chinese.

Prototypical of the serious games such as WarCAT and Locker discussed above is the potential to capitalize on advances in machine learning to aid in assessing or categorizing play. The initial thought behind the early work in these two particular games was to develop them to be interesting enough for a large number of people to play. This would allow new players to be assessed against others. For this to have merit, many players are required. It was realized relatively early on that this approach had many drawbacks. The most discouraging drawback was that these games are in effect not sufficiently engaging to acquire a large database of play. However, in many areas where the attempt is to demonstrate the utility of machine learning, one is often forced with creating synthetic data with the reduced expectations with respect to the games themselves, while still supporting the conjecture that machine learning could be a useful tool in classifying real data if it were available.

1.5 Machine Learning

Machine learning is an application of artificial intelligence and is defined as a computing machine demonstrating an ability to learn where its performance improves with experience in performing some task [23]. There are three main types of machine learning: supervised learning, unsupervised learning, and reinforcement learning [24].

In supervised learning a machine learns a task from labeled training data. Supervised Learning is one of the most widely used paradigm of machine learning. It is generally used in classification or pattern recognition. Supervised Learning algorithms have shown incredible success since the performance of image classification in 2011 [25]. After this, more attention has been paid to various machine learning algorithms that have worked well in applications such as detecting cancer, detecting spam email, object recognition and/or tracking in images and videos, and speech-to-text [26].

The way Supervised Learning algorithms learn is by training models on large sets of labelled data called training data to improve performance. For example, labelled data of images for a model detecting types of cars means each image is labeled with the type of car it portrays. During training, the model would read the input data and then give a prediction of its label as an output. The predicted output is then compared with the correct output. If they match then no adjustment is needed, otherwise the model is adjusted towards giving the correct output. The training process keeps iterating until the model can produce an output that matches the labeled output or by reaching a certain percentage of correct classifications in the training data. After this process, the model has now learned how to classify or detect the correctly labeled data. By extension, it can generalize to label new data.

In unsupervised learning a machine infers structure from unlabeled data. These algorithms identify hidden patterns or data clusters without human assistance. Suppose the unsupervised learning algorithm is given an input dataset containing images of different types of cars. However, the images are unlabelled so the algorithm will not have any idea about the features of the dataset. The task of the unsupervised learning algorithm is to identify the image features on

their own. An unsupervised learning algorithm will perform this task by clustering the image dataset into the groups according to similarities between images. In this fashion unsupervised learning can be used for more complex tasks as compared to supervised learning. Surprising or atypical associations with the training data may be learned through unsupervised learning. Unsupervised learning can also be preferable as it is easy to get unlabeled data in comparison to labeled data. However, unsupervised learning is intrinsically more difficult than supervised learning as inputs do not have a corresponding output to learn from. As a result, unsupervised learning algorithms might be less accurate compared to supervised learning algorithms. Unsupervised learning applications include clustering of semantics [27], grouping news articles about similar topics [28], anomaly detection [29], and DNA sequencing [30].

Reinforcement Learning (RL) allows the machine to learn a task based on feedback from an environment. It has demonstrated success in training software agents that can interact in an environment. The agent views its environment, and the input would be the state of the environment. Reinforcement Learning then gives an output or an action to take in the environment which moves the agent into a neighbouring state. Unlike in Supervised Learning where the output is checked with an authoritative answer after one step (one iteration or one operation of processing), in Reinforcement Learning the output is checked with an authoritative answer after a sequence of steps when the agent reaches an endpoint. However, it assigns rewards to states so that it can find the sequence of outputs or path of actions that leads to the desired endpoint. Reinforcement Learning is used in applications such as playing games [31], industrial automation [32], self-driving cars [33], stock trading [34], optimization [35], and path finding [36].

1.5.1 Deep Learning

Deep learning is a subset of machine learning based on artificial neural networks which attempts to automatically discover the representations needed for feature detection or classification from raw data [37]. Deep learning distinguishes itself from classical machine learning by the type of data that it works with and the methods in which it learns. Deep learning involves the use of complex models that exceed the capabilities of machine learning tools such as logistic regression and support vector machines.

Deep learning differs from machine learning because machine learning algorithms leverage structured, labeled data to make predictions meaning that specific features are defined from the input data for the model and organized into tables. This does not mean that machine learning algorithms do not use unstructured data. Rather, it means the data generally goes through some pre-processing to organize it into a structured format.

Deep learning eliminates some of data pre-processing that is typically involved with machine learning. These algorithms can ingest and process unstructured data, like text and images, and it automates feature extraction, removing some of the dependency on human experts. For example, in a classification problem deep learning algorithms can determine which features are most important to distinguish classes from each other. In machine learning, this hierarchy of features is established manually by a human expert.

Through the processes of gradient descent and backpropagation, the deep learning algorithm adjusts and fits itself for accuracy, allowing it to make predictions about new input with increased precision.

1.5.2 Artificial Neural Networks

A Supervised Learning method and data processing system called an Artificial Neural Network (ANN) is composed of interconnected neurons that are typically arranged in stacked layers. One layer's neurons are linked to some or all the neurons in the opposite layer's surrounding layers. The data is processed by the ANN in one direction by layers: first the input layer, then the hidden layer(s), and then the output layer. An ANN is referred to as a deep neural network when it has several layers.

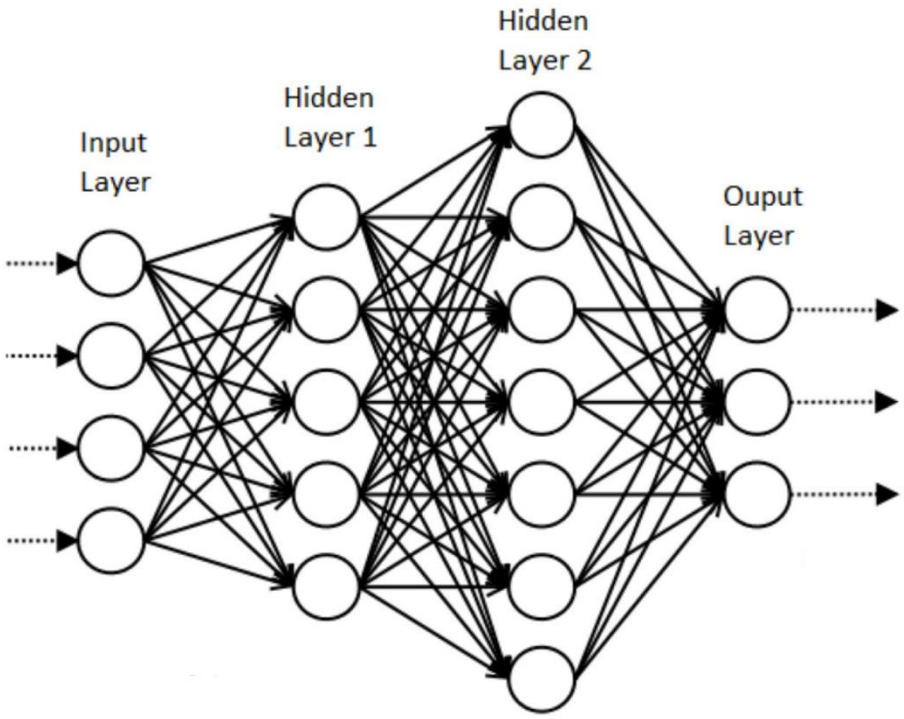

Figure 5 Sketch of multi-layered (deep) artificial neural network

When the neurons in an ANN are connected to all the neurons in the next layer, the ANN is described as a dense neural network or fully connected neural network. Other ANN’s may have

different types of layers that do more than conventional layers such as preprocessing or normalizing layers, convolution layers, or pooling layers.

Each neuron receives multiple inputs from its connections to the previous layer and gives one output. Each connection is assigned a number called the weight that determines the significance of each input. Each input is multiplied by its weight and then the output is the sum of all those products plus a number called the bias. The bias can shift the resulting output value.

$$output = bias + \sum_{i=1}^{n} input_i * weight_i \tag{3}$$

The output of each neuron goes through a function called an activation function which can be any of various types of activation functions as in Table 2 below. The activation function is chosen depending on which machine learning model is used. When the activation function used is non-linear then the ANN is non-linear.

Table 2 Widely Used Activation Functions

Binary Step	$f(x) = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geq 0 \end{cases}$
Rectified Linear Unit (ReLU)	$f(x) = \begin{cases} 0 & for\ x \leq 0 \\ x & for\ x > 0 \end{cases}$
Leaky Rectified Linear Unit (LReLU)	$f(x) = \begin{cases} ax & for\ x \leq 0, 0 < a < 1 \\ x & for\ x > 0 \end{cases}$
Sigmoid	$f(x) = \frac{1}{1 + e^{-x}}$
Softmax	$f_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}\ for\ i = 1, \ldots, J$
Hyperbolic Tangent (Tanh)	$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

The Rectified Linear Unit (ReLU) function is one of the more popular activation functions. The ReLU output is 0 if the input is negative, otherwise the output is the same as the input. It is commonly used as it is computationally efficient with comparable results [38]. Additionally, ReLU alleviates the vanishing gradient problem [39]. The Leaky ReLU Activation Function (LReLU) is very similar to the ReLU Activation Function with one change. Instead of sending negative values to zero, a very small parameter is used which allows incorporation of information from negative values instead of reducing them to zero [40]. One of the important aspects of the LReLU is that is solves the “dead ReLU” or “dying ReLU” problem that occurs due to gradients of 0 with negative inputs which will block learning.

The softmax activation function (normalized exponential function) is another popular activation function which receives a vector of inputs at a time and takes the exponential of one of the inputs over the sum of the exponential of all inputs. Softmax is used at the final layer (the output layer) to limit the output between 0 and 1.

When an ANN is learning or training, the values of the weights and the bias are adjusted over many iterations. Their values are either increased or decreased by a certain amount to make the ANN give results consistent with those of the training data. During this training, when an output given by the ANN is inaccurate, the partial derivative of the activation function with respect to each weight and bias in that one neuron determines the direction and the amount of change needed in the previous neuron in the previous layer. Then similarly for the previous layer, the direction and amount of change is found for the next step and continues back until the input layer is reached and adjusted. The amount and direction of change is often determined by Stochastic

Gradient Descent (SGD). This algorithm for finding the gradients throughout the layers is known as backpropagation [41].

1.6 Summary

To summarize, the long-term objectives are to use data generated from human gameplay to extract a cognitive fingerprint representative of higher-level cognitive functions of strategy (learning), retention (memory) and recall. The collected human data will then be used within a ML classifier to help detect issues such as MCI. Early detection of MCI can lead to an early and more effective therapy. The platforms for the ML models are functional, competitive serious games WarCAT and Locker that have been developed and are instrumented to collect human player data. WarCAT is based on the somewhat familiar card game WAR, and Locker is a lock picking game. Even with these simple games, the potential contributions of ML in classification and detection of subtle cognitive change require considerable data. Thus, a RL model is developed to imitate and produce human-like data of mass quantity.

2 Deep Reinforcement Learning: Introduction

Deep reinforcement learning can be seen as a combination of deep learning and reinforcement learning. That is, deep learning applies techniques of reinforcement learning with some type of neural network used as a function approximation [42].

Deep reinforcement learning is still new and a developing field. While simply learning to play games is a frivolous problem, once the technology grows deep reinforcement learning could be one of the main techniques for solving more practical problems such as self-driving cars or robotics. For this project we propose to use deep reinforcement learning algorithms to learn how to play serious games. From these, we can then use the deep reinforcement learning algorithm to play the game and collect data without human players.

There are already numerous examples of using deep reinforcement learning to play simple video games. One of the most famous examples of deep reinforcement learning is DeepMind's model for playing Atari 2600 games [43]. In this project, we first attempt to replicate the results of this paper by implementing their deep reinforcement learning network. After this we created WarCAT and Locker environments and use the same network to learn to play these games.

The method implemented in [43] to play Atari 2600 games was Deep Q Learning. To summarize the results in their paper, they were able to train a single network that could play the games Beam Rider, Breakout, Enduro, Pong, Q*Bert, Seaquest and Space Invaders. Their network achieved better performance than an expert human player on the games Breakout, Enduro and Pong and it achieves close to human performance on Beam Rider. However, the games Q*bert,

Seaquest, Space Invaders, had worse than human performance since these games are more complicated require a longer time period of which an optimal strategy can be found.

A motivating example of using deep reinforcement learning was where DeepMind learned how to play the Atari 2600 game Breakout [44]. The most surprising aspect of this example is that if the network is trained for long enough it will learn “advanced” strategies for the game. That is, in the game Breakout the best strategy is to try and get the ball behind all the targets so that the ball will automatically break many targets without the user having to hit the ball, thus gaining many points in the smallest amount of time.

There are also a variety of examples with regards to using deep learning and machine learning techniques to play games. One such example is MarI/O which is a program using neural networks and genetic algorithms to play the first level of Super Mario World [45]. Next is an application of deep reinforcement learning to play the smartphone game Flappy Bird [46]. Another example is that a deep reinforcement learning model was trained to play Go and beat one of the best players in the world [47]. All these applications were also contributors to the motivation to apply deep reinforcement learning techniques.

2.1 Background

First, we will establish the RL framework which is essentially the core set of terms and concepts that every RL problem can be expressed in. An objective function is defined such that it returns an error value that indicates how off-target we are at meeting our objectives. Hence, we need our RL algorithm to minimize this objective (error) function’s return value with respect to some input data. The input data is generated by the environment. In general, the environment of a RL

(or control) task is any dynamic process that produces data that is relevant to achieving our objective. Since the environment is a dynamic process (a function of time), it may be producing a continuous stream of data of varied size and type. We take the environment data and bundle it into discrete packets that we call the state (of the environment) and then deliver it to our algorithm at each of its discrete time steps. The state reflects our knowledge of the environment at some time.

Another part of the model is the RL algorithm itself. This could be any parametric algorithm that can learn from data to minimize or maximize some objective function by modifying its parameters. Note that it does not need to be a deep learning algorithm. This is because RL is a field of its own, separate from the concerns of any learning algorithm. More formally, we call the learning algorithm the agent.

One of the key differences between RL (or control tasks generally) and ordinary supervised learning is that in a control task the algorithm (agent) needs to make decisions and take actions. These actions will have a causal effect on what happens in the future. Taking an action is a key component in the framework. Every action taken is the result of analyzing the current state of the environment and attempting to make the best decision based on that information. The last concept in the RL framework is that after each action is taken, the algorithm (agent) is given a reward. The reward is a local signal of how well the learning algorithm (agent) is performing at achieving the global objective. The reward can be a positive signal or a negative signal even though both situations are considered “reward.” The reward signal is the indication the learning algorithm must go by as it updates itself in hopes of performing better in the next state of the environment.

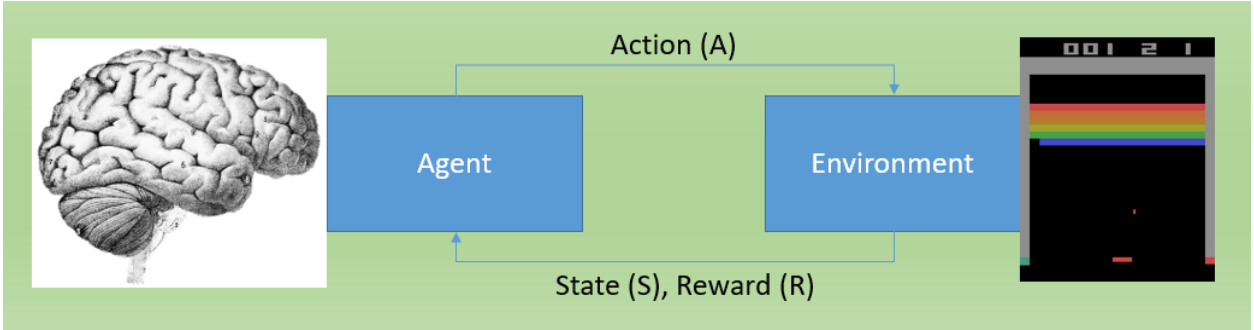

Figure 6 Reinforcement Learning Model

The agent's objective is to maximize its expected cumulative rewards in the long term, which is called the return. The agent repeats of cycle of processing the state information, deciding what action to take, see if it gets a reward, observe the new state, take another action, and so on. The expectation is that the agent will eventually learn to understand its environment and make reliably good decisions at every step.

To be able to define specifically what RL does, there is additional terminology.

An observation is defined as a partial description of a state, which may omit information. When the agent can observe the complete state of the environment, we say that the environment is fully observed. When the agent can only see a partial observation, we say that the environment is partially observed.

Different environments allow different kinds of actions. The set of all valid actions in each environment is often called the action space. Some environments have discrete action spaces, where only a finite number of moves are available to the agent. Other environments have continuous action spaces. In continuous spaces, actions are real-valued vectors.

A policy is a rule used by an agent to decide what actions to take. Policies can be deterministic or stochastic. In RL policies are parameterized. That is, the outputs are computable functions that depend on a set of parameters such the weights and biases of a neural network which we can adjust to change the behavior via some optimization algorithm.

A trajectory is defined as a sequence of states and actions in the world. State transitions are governed by the environment and depend on only the most recent action. State transitions can be either deterministic or stochastic.

2.2 RL Algorithms

We will now talk about the framework of algorithms used in modern reinforcement learning, as well as the many trade-offs made when building algorithms. We illustrate the main design choices made by deep reinforcement learning algorithms based on what to learn and how to learn, highlight these decisions, and place some of the top modern algorithms in the context of

these choices.

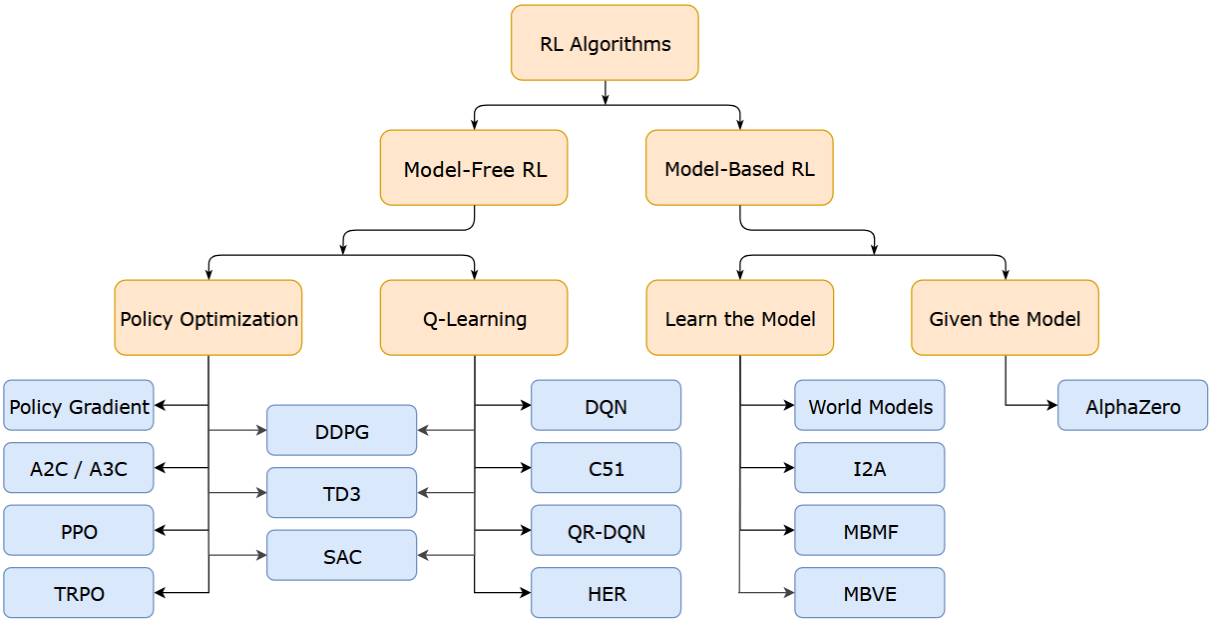

Figure 7 Classifications of some algorithms in modern RL [48]

2.2.1 Model-Free versus Model-Based

One of the most significant branching points in a reinforcement learning algorithm is whether the agent has access to or learns a model of the environment. This will distinguish between model-free types and model-based types of reinforcement learning algorithms. The term “model” refers to a function that predicts rewards and state changes.

The main advantage of having a model is that it gives the agent the ability to plan by looking ahead at events, evaluating the results of different possibilities, and making explicit decisions about its options. The consequences of the agent’s advanced planning can then be converted into a learned policy. A famous example of this approach is AlphaZero [49]. AlphaZero is an algorithm that can achieve superhuman performance in many challenging domains including

Chess, Shogi and Go. When model-based algorithms are effective, sample efficiency can be significantly increased compared to techniques that do not use a model.

The main problem is that the agent typically lacks access to an accurate model of the environment. If an agent wants to use a model in this circumstance, it must learn the model only through experience, which creates several challenges. The agent's ability to exploit model bias to its advantage is the main issue. When this happens, the agent performs well according to its learned model but poorly in the real world. Model learning is also inherently difficult. Even large efforts, like a lot of computing time, may not be successful in learning a model.

Model-free approaches are often easier to set up and fine tune, but they lack the potential sample efficiency advantages that come with using a model. Model-free methods have often undergone more extensive development and testing than model-based ones.

2.2.2 Learning Methods

The next branching point in a reinforcement learning algorithm is what to learn. For model-free algorithms, this includes Policy Optimization and Q-functions. It is more difficult to classify learning methods in Model-Based algorithms, but in these cases the model is either given or learned.

2.2.2.1 Learning in Model-Free Algorithms

The first approach of model-free RL we will discuss is Policy Optimization. Methods in this family represent a policy explicitly. They optimize parameters either directly by gradient ascent on the performance objective, or indirectly by maximizing local approximations of the performance objective. This optimization is almost always performed on-policy, which means

that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximation for the on-policy value function, which gets used in figuring out how to update the policy.

Some examples of policy optimization methods are A2C / A3C which performs gradient ascent to directly maximize performance [50] and PPO whose updates indirectly maximize performance by instead maximizing a “surrogate” objective function using stochastic gradient ascent which gives a conservative estimate for how much the performance objective will change as a result of the update [51].

The second approach is Q-Learning. Methods in this family learn an approximation for the optimal action-value function. Typically, they use an objective function based on the Bellman equation. This optimization is almost always performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained through the connection between the optimal action-value function and the optimal policy function.

Examples of Q-learning methods include DQN which is a classic that significantly advanced the field of deep reinforcement learning [43] and C51 which is a variant that learns a distribution over return whose expectation is the optimal action-value function [52].

The primary strength of policy optimization methods is that they directly optimize the policy. This tends to make them stable and reliable. By contrast, Q-learning methods indirectly optimize agent performance by training an approximation of the optimal action-value function to satisfy a

self-consistency equation. There are many failure modes for this kind of learning, so it tends to be less stable [53]. But Q-learning methods gain the advantage of being considerably more sample efficient when they do work, because they can reuse data more effectively than policy optimization techniques.

By chance, it turns out that policy optimization and Q-learning can be compatible (and under some circumstances equivalent) and there exists a range of algorithms that live in between the two extremes. Algorithms that live on this spectrum can carefully trade-off between the strengths and weaknesses of either side. Examples include DDPG which is an algorithm which concurrently learns a deterministic policy and a Q-function by using each to improve the other [54] and SAC which is a variant that uses stochastic policies, entropy regularization, and a few other tricks to stabilize learning and score higher than DDPG on standard benchmarks [55].

2.2.2.2 Learning in Model-Based Algorithms

Unlike model-free RL, there aren't a small number of easy-to-define clusters of methods for model-based RL. There are many orthogonal ways of using models. A few examples are defined, but the list is far from exhaustive. In each case, the model may either be given or learned.

The most basic approach never explicitly represents the policy and instead uses pure planning techniques like model-predictive control (MPC) to select actions. In MPC, each time the agent observes the environment, it computes a plan which is optimal with respect to the model, where the plan describes all actions to take over some fixed window of time after the present. Future rewards beyond the horizon may be considered by the planning algorithm using a learned value function. The agent then executes the first action of the plan, and immediately discards the rest of it. It computes a new plan each time it prepares to interact with the environment, to avoid using

an action from a plan with a shorter-than-desired planning horizon. MBMF uses MPC combined with medium-sized neural network models on some standard deep reinforcement learning benchmark tasks [56].

Expert Iteration is a straightforward follow-on to pure planning which involves using and learning an explicit representation of the policy. The agent uses a planning algorithm (like Monte Carlo Tree Search) in the model, generating candidate actions for the plan by sampling from its current policy. The planning algorithm produces an action which is better than what the policy alone would have produced, hence it is an "expert" relative to the policy. The policy is afterwards updated to produce an action more like the planning algorithm's output. The ExIt algorithm uses this approach to train deep neural networks to play Hex [57]. AlphaZero is another example of this approach [49].

Data Augmentation for Model-Free Methods uses a model-free RL algorithm to train a policy or Q-function, but either augments real experiences with fictitious ones in updating the agent or uses only fictitious experience for updating the agent. MBVE is an example of an RL algorithm that augments real experiences with fictitious ones [58]. World Models is an example of an RL algorithm that uses purely fictitious experience to train the agent, which is referred to as "training in the dream" [59].

Embedding Planning Loops into Policies is another approach which embeds the planning procedure directly into a policy as a subroutine, so that complete plans become side information for the policy while training the output of the policy with any standard model-free algorithm. The key concept is that in this framework, the policy can learn to choose how and when to use the plans. This makes model bias less of a problem, because if the model is bad for planning in some

states, the policy can simply learn to ignore it. I2A is an example of an RL algorithm featuring agents being endowed with this style of imagination [60].

2.3 Deep Reinforcement Learning for Serious Games

In this section we will go over factors that will be considered for choosing a deep RL algorithm to learn to play our games WarCAT and Locker. One of the first considerations is to determine which algorithms have the best performance on average. Literature supports that the PPO algorithm stands out as performing the best on average [61] [62]. We also need to consider the length of time it takes to train the model with these algorithms. For this reason, we will not be using model-based algorithms because the model is not given, and it will take too long to learn the model.

Another one of the main distinctions for choice of algorithm comes from the action space for your application. That is, we need to determine whether discrete actions (up, down, left, right) or continuous actions (go to a certain speed, move an arm) are used. Some algorithms only permit the use of discrete or continuous action spaces. In our case, for both WarCAT and Locker, they use a discrete action space.

Depending on the game you are trying to learn, algorithms will perform differently [63] [64]. Some algorithms are also not compatible with certain games. For example, MARL cannot be used for Locker because there is one agent. It could be used with WarCAT if we had multiple players, but we are only using one.

Another factor would be whether the algorithm supports the use of single or multi-processing. Algorithms that support multi-processing like A2C and PPO would be better for more complex

environments because you can save time during training through parallelizing. Since our games are simple this does not make a difference to us.

Use of a library such as stable baselines 3 [65] will allow us to try multiple algorithms easily without having to implement them ourselves. However, there are drawbacks to using libraries such as these. For example, without the ability to modify the implementation of the algorithm, performance differences could be due to the absence of some improvements to the algorithm (such as experience replay). Using this library, multiple algorithms were tested such as A2C, DDPG, DQN, PPO and SAC. Although from research PPO was considered one of the best RL algorithms on average [51], it was found that each algorithm performed similarly with WarCAT and Locker. Despite DQN being one of the preliminary algorithms for deep RL, with improvements it has competitive performance [66] and it is sufficient for our applications of WarCAT and Locker due to their simplicity.

3 Methods

3.1 Serious Games

The serious games we are learning to play are called WarCAT and Locker. I will summarize the rules for playing these games that will need to be incorporated as environments for the deep reinforcement learning algorithm.

WarCAT emulates the physical card game WAR, now implemented on a mobile device. In WarCAT there are two players: yourself and an opponent. Each player is dealt five random cards from a standard 52 card deck. Each player then selects one card each and plays them at the same time. (The cards are not returned to the player's hands.) The player with the higher scoring card earns points for the round. At the end of the five rounds, the winner of the game is decided by who has the most points. There are two possible scoring schemes; one where the winning play earns a single point and another where the winning play earns more points the closer in value the cards are.

Locker emulates the task of unlocking a combination lock with three 3 number barrels, now implemented on a mobile device. In Locker, the environment has three modifiable dials with numbers ranging from one to five and an unlock button. The player must set the dials and try to perform a successful unlock. The correct combination is determined randomly, so the player must set the dials and perform unlocks by trial and error to find the correct combination. The player receives a score after an attempted unlock to let them know how close they are to unlocking but must try to unlock in the least number of attempts as possible to maximize their

score. The game is also complicated by local maxima/minima that can fool the player into thinking they are close to the global.

3.2 Deep Q Learning

Next, I will go over basic concepts for reinforcement learning and Deep Q Learning [67]. We will define an agent (a human or computer player) and the game environment. The agent performs an action from a set of total possible actions following the rules in the game environment. In response to actions performed in the environment the agent receives information about the change in state as well as if a reward was obtained from that action. In general, the state is information about the current game being played and the action is the set of possible inputs to the game. For our case the state would be the cards we have in our hand and our score for WarCAT, and values of the dials and our score for Locker. The actions would be selecting a card for WarCAT and changing a lock or trying to unlock for Locker. The reward is dependent on what game the agent is playing or how you set up the environment, but usually the reward is equivalent to the number of points scored. Both of our games have a score, where WarCAT earns points for playing a higher card and in Locker we get a reward upon a successful unlock.

The model that is used is called the Markov Decision Process (MDP) [68]. In general, a MDP provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of the player. For MDP assumptions to hold the game must satisfy the Markov property. The Markov property is defined such that the current state alone contains enough information to choose optimal actions to maximize future reward.

That is, we do not need to remember any past states as an input into our network. If a game satisfies the Markov property, we can use a MDP to simplify our work. For example, drawing cards randomly from a deck with replacement does satisfy the Markov property, because previous card draws do not affect the probability of the next draw. However, drawing a card randomly from a deck without replacement does not satisfy the Markov property, because knowledge of past draws will affect the probability of the next draw. The Atari game Breakout does not satisfy the Markov property because a static game image is unable to tell us if a ball or enemy is moving towards or away from us. For this reason, Deepmind used the last 4 frames as input into their model to play Breakout. For our games, Locker does not satisfy the Markov property because we do not know the correct combination and knowledge of past unsuccessful unlocks in the current game will change our strategy for unlocking. Even though WarCAT could have knowledge of the opponent's past playing strategy to maximize their score, if we are playing against a bot who is using a simple static strategy, then the Markov property would be satisfied. However, in WarCAT we do not know the opponent's cards or what card they will play so the Markov property is not satisfied. The impact of this will be discussed with our results.

When there is a reward, it is not always the result of the action taken immediately before. Some action taken long before might have caused the victory. Because rewards are delayed, good players do not choose their plays only by the immediate reward. Instead, they choose by the expected future reward. From this idea, we will define a function Q(S, A) that will try to predict our future reward when we take the current action. That is, given the current state S and performing some action A it returns an estimate of a total reward we would achieve by starting at this state, taking action A and then following some policy for taking the rest of the actions. Let's

say we are using an optimal policy, meaning that we always select an action which is the best in the context of the current state. The Q function for these optimal policies is called Q*. If we knew Q*, choosing the best action would be easy. Just check the value of Q*(S, A) for all possible actions and then select the action that has the highest value.

However, we also want to prioritize a reward that would occur right now rather than later. For this, we should introduce a factor γ called the discount factor. We make this value very close to 1 (usually ~0.99) so that current rewards will be prioritized but we also want to sacrifice an early reward if we can get a much higher reward later. So, we can write the formula for this as follows:

$$Q^*(S,A) = R_0 + \gamma R_1 + \gamma^2 R_2 + \gamma^3 R_3 + \cdots \quad (4)$$

Where R_x is the reward received from performing action A_x on state S_x.

We can write the above formula in its recursive form:

$$Q^*(S,A) = R_0 + \gamma(R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots) \quad (5)$$
$$Q^*(S,A) = R_0 + \gamma \max_a(Q(S',A'))$$

The final form of this equation is known as the Bellman Equation.

However, in reality we cannot know Q* since there is a massive number of possible states and actions, and we can't store them all. Therefore, the main idea with Deep Q Learning is to estimate Bellman's Equation (Q*) using a neural network:

$$\mathrm{Q}\,(S, A, \theta) \sim = Q^*(S, A) \tag{6}$$

We can then write the loss function we are trying to minimize:

$$\mathcal{L}(\theta_i) = [y_i\text{- Q}\,(S, A, \theta_i)]^2 \tag{7}$$
$$y_i = R_0 + \gamma \max_{a'}(\mathrm{Q}(S', A', \theta_{i-1}))$$

3.3 Environment

The OpenAI Gym framework in Python [69] was used to develop the environments for WarCAT and Locker. OpenAI gym has predefined games available to play, but also provides a framework for creating your own game logic. The required components to implement for a game are the initialization function for setting up the game environment, reset function to start the game from the beginning, step function to provide an action to the environment, the number of possible actions and the format of the observations. After selecting an environment, you can use the step function to provide an action to the environment and upon which will return the next state, your score, a Boolean indicating whether the game has ended, and additional debugging information. Therefore, using OpenAI Gym allows you to create the training data required for the network by providing the current state, the next state, and reward received. At the same time this also allows the network to play the game by selecting our action which corresponds with the maximum Q value from inputting the current state into the network. OpenAI Gym also allows you the option of enabling playback or recording videos of your network playing the game, if supported by your game.

The games have been implemented in the OpenAI Gym framework with the following attributes. WarCAT takes an integer value from 0 to 4 as an input for its action. These actions correspond to selecting one of the five cards in your hand. WarCAT will return an array of 8 float values between -1.0 and 1.0 for its observations. Floating point values have been used for efficiency for training in the reinforcement learning algorithm. The first 5 values correspond to the card values of your hand. A higher float value is a higher card. A value of -1.0 corresponds to a card that has already been used in a previous round. The sixth observation is the card your opponent played that round. The seventh is your total score, and the eighth is your opponent's total score. The reward schedule is returned as a float value between -1.0 and 1.0 as follows: +1.0 when you successfully win the game. +0.5 when you tie the game. -0.5 when you lose the game. -1.0 when you play an invalid card, and you lose one of your cards. 0.0 when you successfully played a card, but the round is not over yet. The game is over after 5 rounds when all 5 cards from both players have been played.

Locker takes an integer value from 0 to 3 as an input for its action. The first three correspond to increasing the number on the dial. The minimum value of a dial is 1 and the maximum is 5. If the dial is increased at the value of 5, then it will roll back to 1. The last action will attempt to unlock. Locker will return an array of 4 float values between -1.0 and 1.0 for its observations. The first three values correspond to the current value on the dial. The last observation is your score, where a higher value means you are closer to unlocking. It is updated only when you perform an unlock action. The reward schedule is as follows: +1.0 on successful unlock. -1.0 on a timeout (when 1000 actions are taken without a successful unlock). -0.25 on an unsuccessful

unlock. -0.25 when changing the dial. The game is over when you unlock successfully or after 1000 actions are taken without a successful unlock.

3.4 Deep Q Learning Improvements

Next, I will describe a few of the other techniques that were implemented to try and improve the results of training. First, I will describe the concept of Experience Replay [70]. Experience Replay is simply where we will save an instance of each experience state (including the State, Action, Reward, Next State, and Done status) into an array of a pre-determined size. At the beginning of the program, the replay memory is filled before performing any training. We will then perform training on a random batch of random samples obtained from the array. The purpose of experience replay is to break the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. This will reduce the bias in samples.

Next, another technique used is Double Deep Q Network [71]. The main idea here is to implement two copies of the networks: the “Online” and “Target” networks, with separate weight values. We will always perform weight updates to Online network every time we train, but every certain number of training iterations we will copy the weights from the Online network to the Target network. We use the Online network for estimating current Q values and therefore actions to perform, and the Target network is used for estimating the Temporal Difference (TD) target or the discounted future Q values. The main idea with Double Deep Q Network is that normal Deep Q Networks tends to overestimate Q values due to its max operation applied to both

selecting and estimating actions. Having a second Target network decouples the selection from the evaluation. This leads to better performance of our network.

During the selection of our action, the Epsilon Greedy technique is used [72]. The main purpose of the Epsilon Greedy technique is to sometimes choose a random action instead of always selecting our optimal Q value action so we can try to explore all possible combinations of actions and possible find a better strategy than what we already know. To implement this, we will define an exploration rate “epsilon,” which is set to 1 in the beginning. This is the rate of steps that are done randomly. That is, in the beginning, this rate is at its highest value, because our network is not very optimized. We will need to do a lot of exploration at the start of training by randomly choosing our actions. The main idea with Epsilon Greedy is that we should have a big epsilon at the beginning of the training. Then, reduce it progressively as we become more confident at estimating Q values.

3.5 Summary

So, putting all these concepts together I will describe the basic flow of the code. First, initialize all variables and then setup the OpenAI Gym environment with the game we want to train. Next, we will select an action using the Epsilon Greedy method and take a step in the environment using that action. We will then save the current state (including Previous State, Action, Reward, Next State, and Done status) to replay memory. Note that for the first batch of iterations we will perform random actions and filling replay memory queue without training the model. After the replay memory queue is filled, we will start training by taking a batch from replay memory and

then perform the training. We will also copy the weights from the Online network to the Target network after a certain number of training iterations. This process continues until the program reaches the number of action steps specified.

After training, we can test our current network to play the game itself by simply setting Epsilon to 0 and skipping the training portion of the code. This way we will always pick the best action determined by our network.

4 Results and Discussion

4.1 Environment Design

First, we will discuss the design philosophy for the environments of Locker and WarCAT. For designing the observation space, the main goal is to give enough information to the agent to solve the task. This means we do not want to break the Markov assumption. It is also important to normalize the observation space to help optimize learning in the agent. You should use running average if you do not know the boundaries of the observations ahead of time. The recommended normalization scheme would be Gaussian around 0 with variance 1.

Design of the observation space of WarCAT and Locker were straight forward considering what an actual player of the games would be observing. In WarCAT, the observation consists of the player's five cards in their hand, the card their opponent played in the previous round, their current score, and the opponent's score. Additionally, a value of zero corresponds to "no card". For example, if a player has an ace, king, five, four and three on the first turn of the game the observation would be an array containing the normalized values of 14, 13, 5, 4, 3, 0, 0, and 0. If the player played their 4 and the opponent played a 10, then the next observation would be an array containing the values 14, 13, 5, 0, 3, 10, 0, and 1 assuming a simple scoring scheme where the opponent received one point for winning the round. In Locker, the observation consists of the values of the dials and the current score. For example, if the values of the dials are two, one, and five and our current score is eight then the observation would be an array containing normalized values of 2, 1, 5, and 8. Note however that the Markov assumption is broken in both games due

to their nature. That is, we cannot see opponent's cards in WarCAT and we cannot see the secret combination in Locker.

For designing the action space, we must choose between discrete actions or continuous actions [73]. An example of discrete actions would be button and switches such as those used in Atari games. An example of a continuous action would be robot movement between two positions. The one we choose depends on what the application of our reinforcement learning algorithm is. In both games discrete actions are chosen. A trade-off for designing the action space to consider is complexity versus final performance. That is, a bigger action space will be more expensive but give the agent more freedom, whereas a smaller action space will lead to quicker learning. Actions should also be normalized if they are continuous actions.

The design of the action space for WarCAT consisted of selecting cards. Therefore, we have used discrete actions. One issue to consider however is that the player may only choose one of their cards once per round and are not allowed to use the same card in subsequent rounds. There were two different approaches to solving this; one approach was to remove the ability for the player to select a card they have already chosen before, and the other approach was to let the player select a card more than once and the round would continue. However, their card value would be a minimum guaranteeing a loss and they would receive a reward penalty. We opted to choose the second approach, since the deep reinforcement learning algorithm would be able to learn that selecting a card a second time would not be a good option for maximizing reward and therefore not perform that action. The action space is defined as an integer value between 0 and 4 where 0 corresponds to playing the first card and 4 corresponds to playing the last card.

We have chosen Locker to have four discrete actions consisting of only the ability to increment each lock and have the lock value reset to the minimum when incremented past the maximum. Initially, an action space of seven actions was considered where the agent would have the ability to increment and decrement the lock values. We found that the agent performed poorly and took longer to train compared to the four actions. In this case the action space is defined such that action 0 will perform an increase to dial one, action 1 will perform an increase to dial two, action 2 will perform an increase to dial three, and action 3 will perform an unlock.

A modified environment of Locker is also defined without an action to perform an unlock. Instead, the unlock is performed automatically upon every step taken. In this case, an action space of size six is used. Action 0 will perform an increase to dial one, 1 will perform an increase to dial two, 2 will perform an increase to dial three, 3 will perform a decrease to dial one, 4 will perform a decrease to dial two, and 5 will perform a decrease to dial three,

Designing the reward function is one of the most important aspects of implementing an environment [74]. First, we will define a primary reward as the task we want the agent to achieve, such as reaching a goal. A secondary reward would include other optimizations related to the problem but are not required to reach the goal. We can choose to have sparse rewards by only including a primary reward or shaped rewards where we would include secondary rewards. With sparse rewards, most rewards the agent will receive will be 0 and will only be given a reward +1 when they reach the goal. Shaped rewards are more informative for the agent, where the agent will be given rewards along the way to the goal. However, the rewards may be deceptive and lead to an incorrect goal. When designing an environment, it is best to start simple with reward shaping but be careful of reward hacking. Reward hacking is when the agent will

learn to maximize reward without solving the task. It is also important to normalize the reward like what we did with the observation space and action space.

In WarCAT, the primary goal is to beat the opponent and so this will result in the maximum reward of +1. Possible secondary rewards could be distributed though assigning positive rewards to winning a single round and negative rewards for losing a single round, however this would not be recommended because a winning strategy may be to play lower cards first and lose the beginning rounds. If we were to reward shape in this fashion the agent might not learn an optimal strategy. Since the main purpose is to beat the opponent after 5 rounds, no secondary rewards were assigned and only a reward of +1 is given for winning or +0.5 for a tie. Negative rewards are given when the agent loses or plays an invalid card. Otherwise, a reward of 0 is given for each round. In this game, the agent may have to learn to sacrifice short term wins through losing earlier rounds to receive long term gains by winning the match.

In Locker, the primary goal is to unlock the lock by inputting the secret combination. This will result in a maximum reward of +1. Besides this, a small penalty of -0.25 is given for each action to try and prevent the agent from getting stuck and a reward of -1 is given on a timeout. The secondary rewards are distributed as an observation with values between -1 and +1, where a lower value means that your combination is further away from the target and a higher value means your combination is close to unlocking. This is necessary because an agent can receive feedback from an unsuccessful unlock to let them know how close they are to unlocking. Without this, an agent would have to randomly guess what the unlock combination is.

There are several ways to decide the termination conditions. Depending on the application, we would most likely terminate once we reach the goal. We can also choose to terminate early with

a timeout so that we do not get stuck. However, a timeout could break Markov assumption. Also, terminating early could lead to reward hacking. For example, if you are penalized at every step the agent could learn to fail quickly to minimize this penalty instead of trying to reach the goal.

WarCAT simply terminates after each player has played 5 cards and Locker would terminate once the lock has been unlocked. However, Locker needed a timeout option with a penalty in case the agent got stuck selecting the same action repeatedly without unlocking the lock. A timeout of 1000 actions was given in Locker, since this would be more than enough to allow for a player to unlock the lock given there are only 125 total combinations given 3 dials and 5 values on each dial.

4.2 Metrics

The main measure for the performance of a deep reinforcement learning algorithm is the reward [75]. That is, we allow a deep reinforcement learning agent to play the game and sum all the obtained rewards until the termination condition is reached. The best performance will be determined by the agent that has the highest sum of rewards. In some environments, termination itself would be the goal and in that case, we would assign minimal rewards unless the goal is reached, and a better performing agent would be able to reach the termination state quicker. Therefore, we will use this to track the agent's performance throughout training and we should see the rewards increase as the agent is trained.

Another metric we could look at is the plot of the loss versus epochs. In general, the loss is a summation of the errors made for each example in training or validation sets. The loss implies

how well or poorly a certain model behaves after each iteration of optimization. Ideally, one would expect the reduction of loss after each iteration. However, due to the nature of deep reinforcement learning the loss does not usually stabilize and so it is not as helpful to analyze.

4.3 WarCAT Performance

First, we will analyze the performance of WarCAT. From Figure 8, we can see that the reward starts out low at around -1.5 due to performing mostly random actions and selecting invalid cards. Later we see a big jump as the agent starts learning and eventually stabilizes with a reward of around +0.5. Due to the player and opponent receiving a random set of cards as their hand it is expected that the agent will not be able to win every single game.

Also, we can look at the following example output and see that WarCAT learned how to counter simple bot by playing lower cards first then higher cards later. Looking at the values of playerCard and opponentCard, we can see that the player chose lower values first and then higher values later: 2, 3, 8, 5 and then 4. The opponent chose cards 12, 10, 6, 2, and then 2.

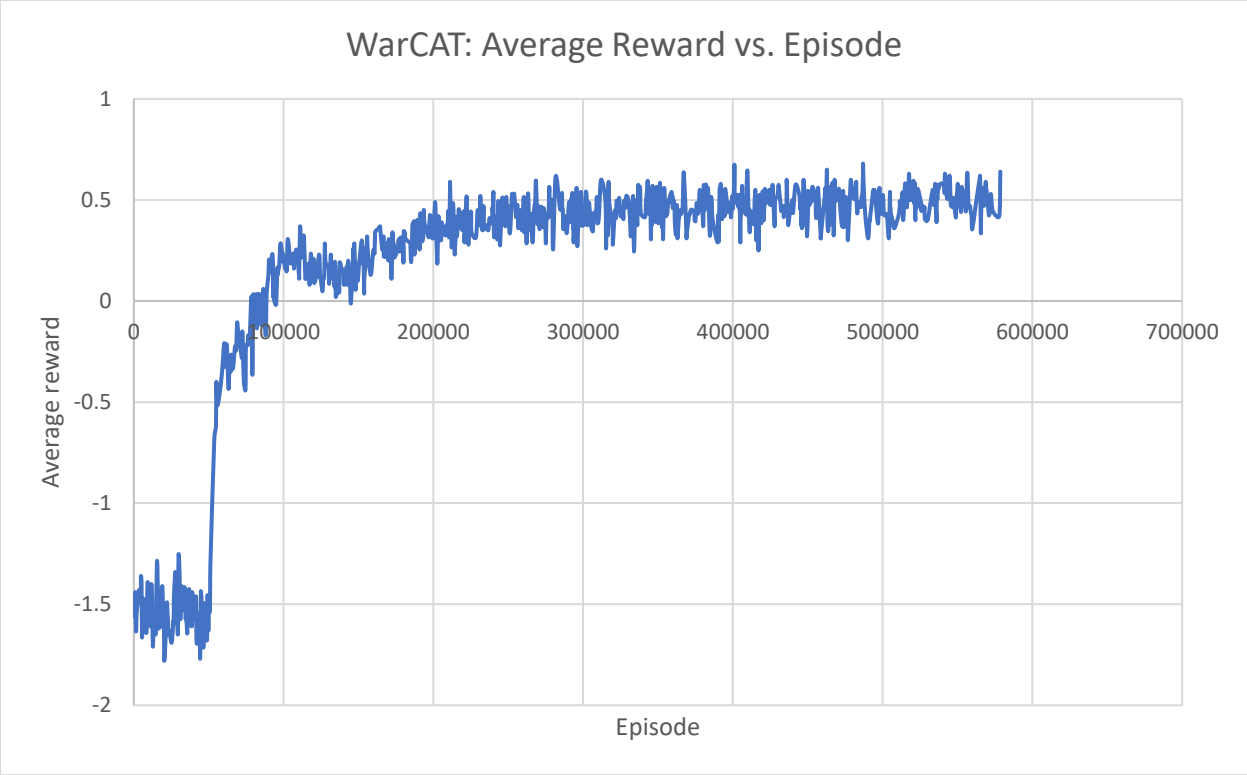

Figure 8 Graph of WarCAT performance

[-1. 0.14285715 -0.2857143 -0.5714286 -0.42857143 0.71428573

-1. -0.6] 0.0 False {'playerHand': [0, 8, 5, 3, 4], 'opponentHand': [10, 2, 2, 0, 6], 'playerCard': 2, 'opponentCard': 12, 'playerRoundScore': 0, 'opponentRoundScore': 1, 'playerTotalScore': 0.0, 'opponentTotalScore': 1.0} 0

[-1. 0.14285715 -0.2857143 -1. -0.42857143 0.42857143

-1. -0.2] 0.0 False {'playerHand': [0, 8, 5, 0, 4], 'opponentHand': [0, 2, 2, 0, 6], 'playerCard': 3, 'opponentCard': 10, 'playerRoundScore': 0, 'opponentRoundScore': 1, 'playerTotalScore': 0.0, 'opponentTotalScore': 2.0} 3

[-1. -1. -0.2857143 -1. -0.42857143 -0.14285715

-0.6 -0.2] 0.0 False {'playerHand': [0, 0, 5, 0, 4], 'opponentHand': [0, 2, 2, 0, 0], 'playerCard': 8, 'opponentCard': 6, 'playerRoundScore': 1, 'opponentRoundScore': 0, 'playerTotalScore': 1.0, 'opponentTotalScore': 2.0} 1

[-1. -1. -1. -1. -0.42857143 -0.71428573

-0.2 -0.2] 0.0 False {'playerHand': [0, 0, 0, 0, 4], 'opponentHand': [0, 0, 2, 0, 0], 'playerCard': 5, 'opponentCard': 2, 'playerRoundScore': 1, 'opponentRoundScore': 0, 'playerTotalScore': 2.0, 'opponentTotalScore': 2.0} 2

[-1. -1. -1. -1. -1. -0.71428573

0.2 -0.2] 1.0 True {'playerHand': [0, 0, 0, 0, 0], 'opponentHand': [0, 0, 0, 0, 0], 'playerCard': 4, 'opponentCard': 2, 'playerRoundScore': 1, 'opponentRoundScore': 0, 'playerTotalScore': 3.0, 'opponentTotalScore': 2.0} 4

Episode ended, length = 5

Figure 9 Example output from a round of WarCAT

4.4 Locker Performance

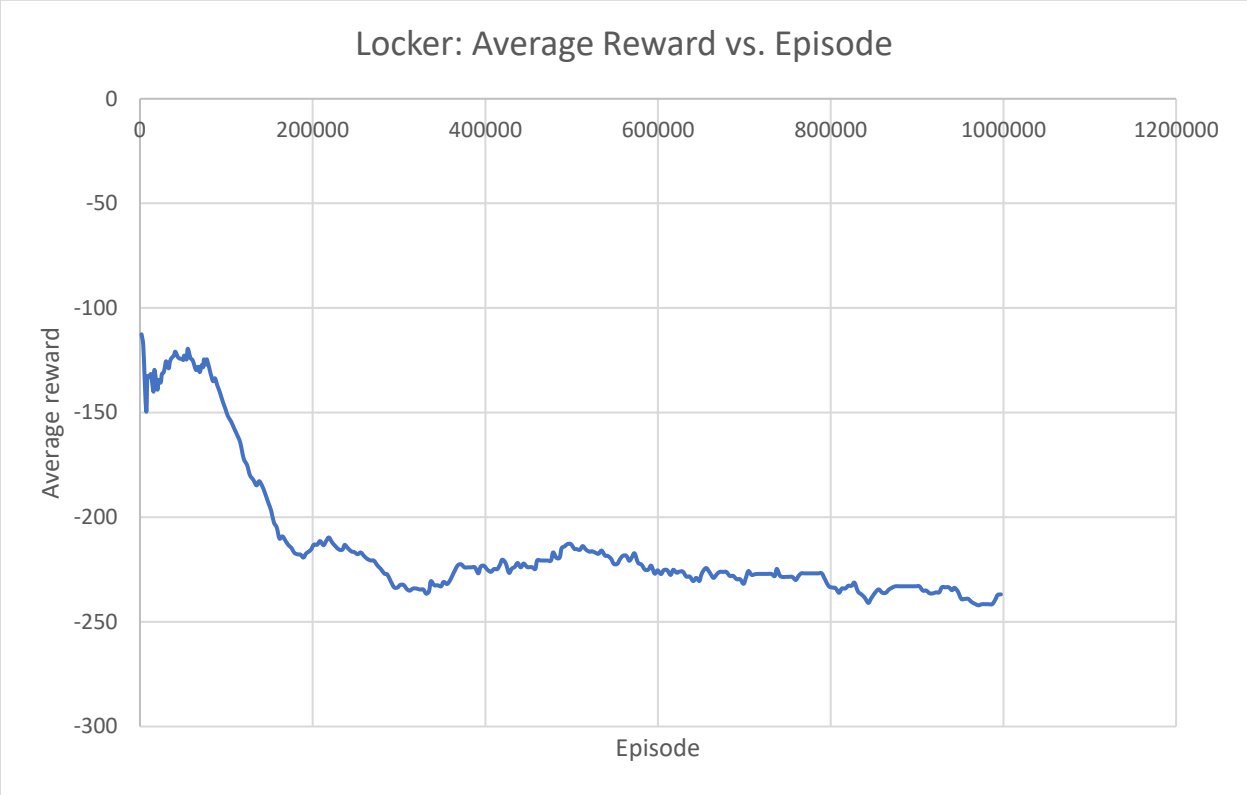

Figure 10 Graph of Locker performance

Next, we will look at the performance of Locker. Unfortunately, our agent does not learn how to play Locker as well as WarCAT. In fact, in the above Figure 10 we can see that the average reward decreases as the agent attempts to learn. This means that random actions are performing better than when our agent attempts to learn to play the game. By watching the learned agent play the game, we can see why this is the case. There were several different behaviors noticed depending on how much the agent is trained. In one case where the agent is not trained for long enough the agent gets stuck performing the same action of changing one of the locks over and

over until it reaches the timeout. Another case is where the agent tries to guess what the correct locker combination is (For example the agent will change the combination to 4-3-2) and if this combination is incorrect, the agent will continually try to unlock until it reaches the timeout. In both cases, the agent displays a lack of flexibility in correcting its behavior when it is unable to know what the correct combination is. Therefore, performing random actions would lead to a better outcome because there is a chance that it will randomly find the correct combination as opposed to always timing out.

There are several reasons why we believe the performance of Locker is poor. One reason which was alluded to earlier is that Locker does not satisfy the Markov property due to the correct combination being hidden. To attempt to remedy this, frame stacking was implemented in a similar manner that was done in DeepMind's agent that learned to play Atari Breakout. That is, the past four observations are used as an input into the model as opposed to only one observation. The purpose for this is to give the model some form of memory of past recent actions. In theory, knowing past unsuccessful unlocks should allow the model to adapt and modify its strategy. Unfortunately, the same poor performance was observed even when frame stacking was implemented. It is possible frame stacking did not work due to the constraints of the action space's ability to affect the environment in each step. That is, since a single action will only perform one unit increase or decrease on a lock it is possible that four observations in the past would not be enough for the agent to know another possible combination to try. Therefore, some ways to try and correct this behavior would be to modify the action space in such a way that bigger increments in the lock value would be possible. For example, the agent could have the ability to change a lock value from one to four in a single step. Additionally, the number of

frames in the frame stacking could be increased which would give the agent more memory of past plays. However, this also introduces the argument of how much memory we should give to make the agent's performance similar to a human player. That is, a human player experiencing MCI would have a lower amount of memory than a player without MCI. However, a lightly trained agent may not use the memory to their full advantage anyways.

Some more reasons for the poor performance could be due to the design of the environment. For example, the unlock being programmed as a separate action. That is, we expect to see better performance if there were only three actions in total for changing the locks. The unlock action would then be performed automatically after each change to the lock. This modification would remove a part of the complexity of the game since it would be easier to come across the correct combination randomly as opposed to strategically choosing a combination based on how close you were with the previous unlock action.

Another environment design choice that could affect performance would be the choice of actions that change the locks. One option would be to have three actions that only increment the lock value, and then rollover back to the minimum lock value once incremented past the maximum value. Another option is to give the ability to increment or decrement the locks but incrementing past the maximum or decrementing past the minimum would result in no change of the lock. Each of these options seem to be essentially equivalent, however it is arguable that the model could have problems in either case. For example, in option one the rollover of the locks could cause issues and in option two having more actions may make it more difficult for the model to learn.

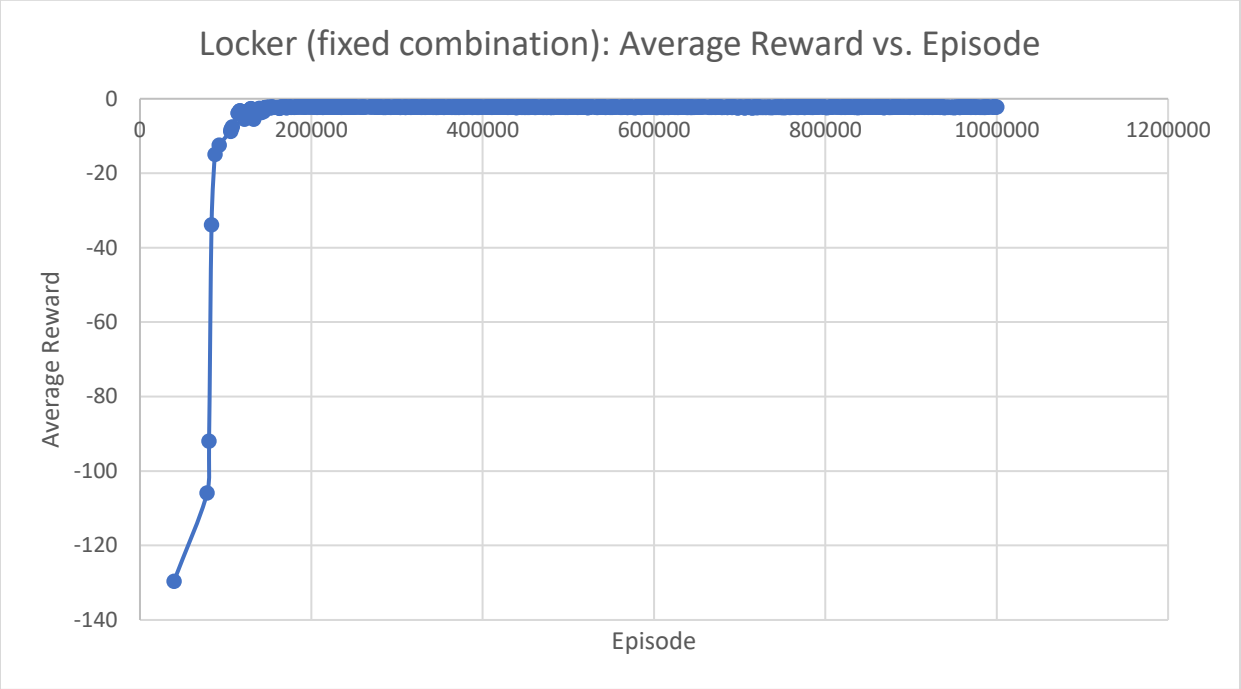

Figure 11 Graph of Locker with a fixed combination performance

Due to the poor performance of the original Locker game, we made an easier version of the same game where the correct combination is always the same (5-5-5). As can be seen in Figure 11, the agent can quickly learn the right combination and afterwards is able to consistently perform the minimum set of actions needed to get to the correct combination and unlock. However, this is a trivial game to learn since the set of actions for a winning condition are always the same.

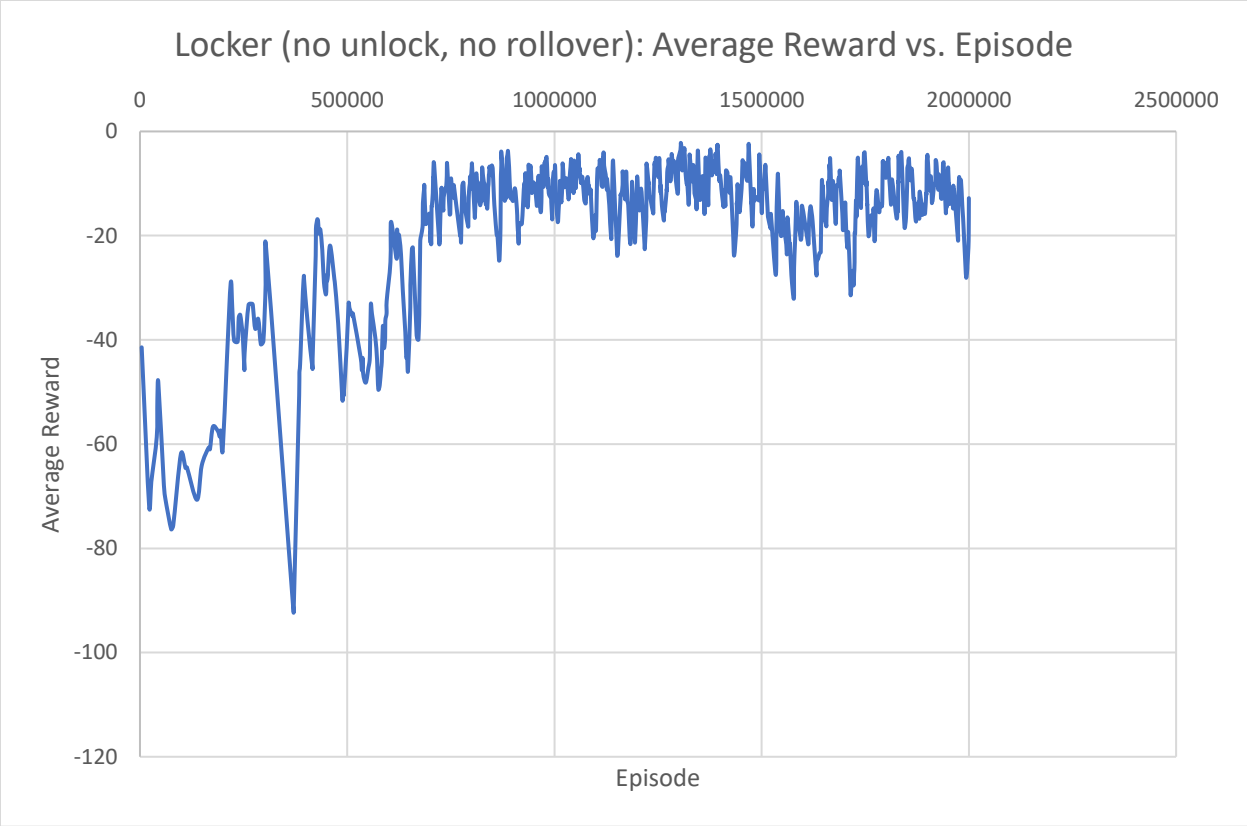

Figure 12 Graph of Locker with “no unlock” and “no rollover” performance

Another version of Locker that was tested was a version where the environment was modified to have no unlocking and no rollover. That is, there were six actions the agent could perform: three for incrementing the locks and three for decrementing the locks. If a lock was incremented past the maximum or decremented past the minimum, then no change would occur. An unlock would automatically be attempted after every action performed. Figure 12 shows the performance after training, and we can see that after around 700000 steps the model seems to be learning how to solve the lock with a score of around -10. If only one of these environment modifications were present, then the performance was poor and like that in Figure 10.

Additionally, the stable-baselines3 library was used to try different RL algorithms outlined in Section 2.2 RL Algorithms. One of the algorithms used was the PPO algorithm due to the recommendation of increased performance. However, we noticed similar performance with the PPO algorithm and our DQN algorithm across all games including the poor performance of the Locker game. One reason for similar performance would be that our games are simple compared to the games RL algorithms are typically used to learn. This includes Atari games such as Pac-Man which is more complicated than our games and requires the use of convolutional networks since its observation space is the image of the game screen. Additionally, a reason for the poor performance of Locker could be that all the outlined RL algorithms assume the Markov property which is not met. That is, history in the Locker game may be much more important than the WarCAT game and has a large effect on its performance which is not yet resolved with other RL algorithms.

5 Conclusion

In conclusion, we were able to describe, implement and train an agent using Deep Q Networks to learn how to play serious games WarCAT and Locker. The agent was able to learn to play WarCAT successfully and modified versions of Locker could also be played at an acceptable performance. In WarCAT, the agent was able to use a strategy of playing lower cards first to counter the simple AI who would play cards from highest to lowest. The agent was able to play a trivial version of Locker where the correct combination was always the same. In this version, the agent will eventually learn what the correct combination is and then always use the minimum amount of moves to unlock. The agent was unable to play the original version of Locker when the correct combination was randomly decided. In this version of Locker, the agent will end up

getting stuck performing the same action until it reaches the timeout. With modifications to the environment, Locker can learn to play with a decent performance. Hopefully future developments in this field will allow for more robust implementations with improved performance of the Locker game.

In regard to generating synthetic game play data, the RL agent playing WarCAT can immediately be used to generate game play data as a function of learning. These synthetic game play data would be of sufficient volume and could then be used as a means to classify real game play from people. This clearly would not address the effectiveness of helping to assess MCI but would help support the premise that serious games play data may lend itself to ML classification. In this scenario a lightly trained agent would be analogous to a person with greater cognitive difficulty whereas a highly trained agent would be analogous to a person experiencing no cognitive difficulty. The most interesting aspects leaned that map to our experience in problem solving are strategies that are learned and relatively complex behaviours such as sacrificing multiple low cards or playing high cards in an attempt to find out the highest card played by the bot.

The results for Locker were less useful in generating synthetic data for similar purposes.

6 Future Work and Considerations

The performance of the models relied heavily on the design of the environment within the OpenAI Gym framework. An initial consideration was to use the Open AI Universe platform [76]. This platform would allow training of an agent using only the pixel data from the actual game and therefore designing a custom environment would not be required. Unfortunately support for this platform has been abandoned as of March 5, 2017. If support for this platform or another similar platform would open in the future a more robust implementation of a deep reinforcement learning model could be possible that does not rely on the design of the environment within the OpenAI Gym framework.

References

[1] United Nations Department of Economic and Social Affairs Population Division, "World Population Ageing 2017 - Highlights," p. ST/ESA/SER.A/397, 2017.

[2] Alzheimer's Association, "More Than Normal Aging: Understanding Mild Cognitive Impairment," *Alzheimer's Disease Facts and Figures,* 2022.

[3] K. M. Langa and D. A. Levine, "The diagnosis and management of mild cognitive impairment: a clinical review," *JAMA,* 2014, DOI: 10.1001/jama.2014.13806.

[4] K. B. Rajan, J. Weuve, L. L. Barnes, E. A. McAninch, R. S. Wilson and D. A. Evans, "Population estimate of people with clinical Alzheimer's disease and mild cognitive impairment in the United States (2020-2060)," *Alzheimers Dement,* pp. 1966-1975, 2021, doi: 10.1002/alz.12362.

[5] R. C. Petersen, "Mild Cognitive Impairment," *Continuum (Minneap Minn),* p. 404–418, 2016.

[6] World Health Organization, "Dementia Facts," https://www.who.int/news-room/fact-sheets/detail/dementia, 2022.

[7] M. Ganguli, B. E. Snitz, J. A. Saxton, C.-C. H. Chang, C.-W. Lee, J. V. Bilt, T. F. Hughes, D. A. Loewenstein, F. W. Unverzagt and R. C. Petersen, "Outcomes of mild cognitive impairment by definition: a population study," *Arch Neurol*, 2011, doi: 10.1001/archneurol.2011.101.

[8] R. Gallassi, F. Oppi, R. Poda, S. Scortichini, M. S. Maserati, G. Marano and L. Sambati, "Are subjective cognitive complaints a risk factor for dementia?," *Neurol Sci.*, 2010, DOI: 10.1007/s10072-010-0224-6.

[9] A. Nordlund, S. Rolstad, O. Klang, A. Edman, S. Hansen and A. Wallin, "Two-year outcome of MCI subtypes and aetiologies in the Göteborg MCI study," *J Neurol Neurosurg Psychiatry*, 2010, doi: 10.1136/jnnp.2008.171066.

[10] D. Fitzpatrick-Lewis, R. Warren, M. U. Ali, D. Sherifali and P. Raina, "Treatment for mild cognitive impairment: a systematic review and meta-analysis," *CMAJ Open*, 2015, doi: 10.9778/cmajo.20150057.

[11] D. J. Selkoe, "Alzheimer disease and aducanumab: adjusting our approach," *Nature Reviews Neurology*, pp. 365-366, 2019, https://doi.org/10.1038/s41582-019-0205-1.

[12] K. Dhana, D. A. Evans, K. B. Rajan, D. A. Bennett and M. C. Morris, "Healthy lifestyle and the risk of Alzheimer dementia: Findings from 2 longitudinal studies," *Neurology*, 2020, doi: 10.1212/WNL.0000000000009816.

[13] J. R. Cockrell and M. F. Folstein, "Mini-Mental State Examination (MMSE)," *Psychopharmacol Bull,* vol. 24, no. 4, pp. 689-692, 1988.

[14] J. S. Lin, E. O'Connor, R. C. Rossom, L. A. Perdue and E. Eckstrom, "Screening for cognitive impairment in older adults: A systematic review for the U.S. Preventive Services Task Force," *Ann Intern Med,* 2013, doi: 10.7326/0003-4819-159-9-201311050-00730..

[15] T. Susi, M. Johannesson and P. Backlund, "Serious Games - An Overview," 2015.

[16] S. Mccallum, "Gamification and serious games for personalized health," *Studies in Health Technology and Informatics,* 2012, DOI:10.3233/978-1-61499-069-7-85.

[17] C. Muscio, P. Tiraboschi, U. P. Guerra, C. A. Defanti and G. B. Frisoni, "Clinical trial design of serious gaming in mild cognitive impairment," *Front Aging Neurosci.,* 2015, doi: 10.3389/fnagi.2015.00026.

[18] T. Tong, M. Chignell, M. C. Tierney and J. Lee, "A Serious Game for Clinical Assessment of Cognitive Status: Validation Study," *JMIR Serious Games,* 2016, doi: 10.2196/games.5006.

[19] V. Vallejo, P. Wyss, L. Rampa, A. V. Mitache, R. M. Müri, U. P. Mosimann and T. Nef, "Evaluation of a novel Serious Game based assessment tool for patients with Alzheimer's disease," *PLoS One,* 2017, doi: 10.1371/journal.pone.0175999.

[20] S. Verschueren, C. Buffel and G. V. Stichele, "Developing Theory-Driven, Evidence-Based Serious Games for Health: Framework Based on Research Community Insights," *JMIR Serious Games*, 2019, doi: 10.2196/11565.

[21] K. Leduc-McNiven, B. White, H. Zheng, R. D. McLeod and M. R. Friesen, "Serious games to assess mild cognitive impairment: ‘The game is the assessment," *Research and Review Insights*, 2018, DOI: 10.15761/RRI.1000128.

[22] R. Roberts, Y. Geda, D. Knopman, R. Cha, V. Pankratz, B. Boeve, E. Tangalos, R. Ivnik, W. Rocca and R. Petersen, "The incidence of MCI differs by subtype and is higher in men," *The Mayo Clinic Study of Aging*, 2012, https://doi.org/10.1212/WNL.0b013e3182452862.

[23] T. Mitchell, Machine Learning: A Multistrategy Approach, McGraw-Hill Education, 1997.

[24] H. U. Dike, Y. Zhou, K. K. Deveerasetty and a. Q. Wu, "Unsupervised Learning Based On Artificial Neural Networks: A Review," 2018, doi: 10.1109/CBS.2018.8612259.

[25] A. Krizhevsky, I. Sutskever and a. G. E. Hinton, "ImageNet Classification With Deep Convolutional Neural Networks," *Commum ACM*, vol. 60, no. 6, pp. 84-90, 2017.

[26] A. Sharma, V. Sharma, M. Jaiswal, H.-C. Wang, D. N. K. Jayakody, C. M. W. Basnayaka and A. Muthanna, "Recent Trends in AI-Based Intelligent Sensing," *Electronics*, 2022, https://doi.org/10.3390/electronics11101661.

[27] P. Liu, Y. Ning, K. K. Wu, K. Li and H. Meng, "Open Intent Discovery through Unsupervised Semantic Clustering and Dependency Parsing," *IEEE CogInfoCom*, 2021, https://doi.org/10.48550/arXiv.2104.12114.

[28] M. T. Altuncu, S. N. Yaliraki and M. Barahona, "Content-driven, unsupervised clustering of news articles through multiscale graph partitioning," 2018, https://doi.org/10.48550/arXiv.1808.01175.

[29] A. Berg, J. Ahlberg and M. Felsberg, "Unsupervised Learning of Anomaly Detection from Contaminated Image Data using Simultaneous Encoder Training," 2019, https://doi.org/10.48550/arXiv.1905.11034.

[30] A. Yang, W. Zhang, J. Wang, K. Yang, Y. Han and L. Zhang, "Review on the Application of Machine Learning Algorithms in the Sequence Data Mining of DNA," *Frontiers in Bioengineering and Biotechnology,* 2020, https://doi.org/10.3389/fbioe.2020.01032.

[31] N. Justesen, P. Bontrager, J. Togelius and S. Risi, "Deep Learning for Video Game Playing," 2017, https://doi.org/10.48550/arXiv.1708.07902.

[32] Q. Xin, G. Wu, W. Fang, J. Cao and Y. Ping, "Opportunities for Reinforcement Learning in Industrial Automation," *7th International Conference on Big Data and Information Analytics (BigDIA)*, 2021, doi: 10.1109/BigDIA53151.2021.9619637.

[33] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani and P. Pérez, "Deep Reinforcement Learning for Autonomous Driving: A Survey," *IEEE Transactions on Intelligent Transportation Systems,* 2020, https://doi.org/10.48550/arXiv.2002.00444.

[34] T. Kabbani and E. Duman, "Deep Reinforcement Learning Approach for Trading Automation in The Stock Market," *International Conference on Artificial Neural Networks,* 2022, https://doi.org/10.48550/arXiv.2208.07165.

[35] P. Dell'Aversana, "Reinforcement learning in optimization problems. Applications to geophysical data inversion," *AIMS Geosciences,* 2022, doi: 10.3934/geosci.2022027.

[36] G. Sartoretti, J. Kerr, Y. Shi, G. Wagner, T. K. S. Kumar, S. Koenig and H. Choset, "PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning," 2018, https://doi.org/10.48550/arXiv.1809.03531.

[37] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, The MIT Press, 2016.

[38] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. S. Dhillon and L. Daniel, "Towards Fast Computation of Certified Robustness for ReLU Networks," 2018, https://doi.org/10.48550/arXiv.1804.09699.

[39] H. Ide and T. Kurita, "Improvement of learning for CNN with ReLU activation by sparse regularization," *2017 International Joint Conference on Neural Networks (IJCNN),* 2017, DOI: 10.1109/IJCNN.2017.7966185.

[40] A. L. Maas, A. Y. Hannun and A. Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," 2013, https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf.

[41] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533-536, 1986, https://doi.org/10.1038/323533a0.

[42] H. Dong, Z. Ding and S. Zhang, Deep Reinforcement Learning: Fundamentals, Research and Applications, Springer, 2020.

[43] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," 2013.

[44] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, p. 529–533, 2015.

[45] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99-127, 2002.

[46] K. Chen, "Deep Reinforcement Learning for Flappy Bird," 2018.

[47] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J.

Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484-489, 2016.

[48] J. Achiam, Spinning Up in Deep Reinforcement Learning, 2018.

[49] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," 2017, https://doi.org/10.48550/arXiv.1712.01815.

[50] 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," *ICML*, 2016, https://doi.org/10.48550/arXiv.1602.01783.

[51] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal Policy Optimization Algorithms," 2017, https://doi.org/10.48550/arXiv.1707.06347.

[52] M. G. Bellemare, W. Dabney and R. Munos, "A Distributional Perspective on Reinforcement Learning," *ICML*, 2017, https://doi.org/10.48550/arXiv.1707.06887.

[53] C. Szepesvari, "Algorithms for Reinforcement Learning," *Morgan & Claypool Publishers*, 2009.

[54] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, "Continuous control with deep reinforcement learning," 2015, https://doi.org/10.48550/arXiv.1509.02971.

[55] T. Haarnoja, A. Zhou, P. Abbeel and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *ICML*, 2018, https://doi.org/10.48550/arXiv.1801.01290.

[56] A. Nagabandi, G. Kahn, R. S. Fearing and S. Levine, "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning," 2017, https://doi.org/10.48550/arXiv.1708.02596.

[57] T. Anthony, Z. Tian and D. Barber, "Thinking Fast and Slow with Deep Learning and Tree Search," 2017, https://doi.org/10.48550/arXiv.1705.08439.

[58] V. Feinberg, A. Wan, I. Stoica, M. I. Jordan, J. E. Gonzalez and S. Levine, "Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning," 2018, https://doi.org/10.48550/arXiv.1803.00101.

[59] D. Ha and J. Schmidhuber, "Recurrent World Models Facilitate Policy Evolution," in *Advances in Neural Information Processing Systems 31*, Curran Associates, Inc., 2018, pp. 2451-2463.

[60] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Hassabis, D. Silver and D.

Wierstra, "Imagination-Augmented Agents for Deep Reinforcement Learning," 2017, https://doi.org/10.48550/arXiv.1707.06203.

[61] T. N. Larsen, H. Ø. Teigen, T. Laache, D. Varagnolo and A. Rasheed, "Comparing Deep Reinforcement Learning Algorithms' Ability to Safely Navigate Challenging Waters," *Front. Robot. AI*, 2021, https://doi.org/10.3389/frobt.2021.738113.

[62] H. Shengren, E. M. Salazar, P. P. Vergara and P. Palensky, "Performance Comparison of Deep RL Algorithms for Energy Systems Optimal Scheduling," 2022, https://doi.org/10.48550/arXiv.2208.00728.

[63] U. Tewari, "Which Reinforcement learning-RL algorithm to use where, when and in what scenario?," 2020, https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1.

[64] J. Hui, "RL — Reinforcement Learning Algorithms Comparison," 2021, https://jonathan-hui.medium.com/rl-reinforcement-learning-algorithms-comparison-76df90f180cf.

[65] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus and N. Dormann, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," *Journal of Machine Learning Research*, vol. 22, pp. 1-8, 2021, http://jmlr.org/papers/v22/20-1364.html.

[66] M. Hessel, J. Modayil, H. v. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," 2017, https://doi.org/10.48550/arXiv.1710.02298.

[67] Kaelbling, L. Pack, M. L. Littman and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research,* vol. 4, pp. 234-285, 1996.

[68] Puterman, Martin L, "Markov decision processes.," *Handbooks in operations research and management science,* no. 2, pp. 331-434, 1990.

[69] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, "OpenAI Gym," *arXiv:1606.01540v1,* 2016.

[70] T. Schaul, J. Quan, I. Antonoglou, D. Silver, "Prioritized Experience Replay," *arXiv:1511.05952,* 2016.

[71] V. Hasselt, Hado, A. Guez and D. Silver, "Deep reinforcement learning with double q-learning," *Proceedings of the AAAI conference on artificial intelligence,* vol. 30, no. 1, 2016.

[72] M. Tokic and G. Palm, "Value-difference based exploration: adaptive control between epsilon-greedy and softmax," *Annual conference on artificial intelligence,* Vols. Springer, Berlin, Heidelberg, pp. 335-346, 2011.

[73] Hausknecht, Matthew and P. Stone, "Deep reinforcement learning in parameterized action space," *arXiv preprint arXiv:1511.04143,* 2015.

[74] Mataric and M. J., "Reward functions for accelerated learning," *Machine learning proceedings 1994,* pp. 181-189, 1994.

[75] Henderson, Peter, et al, "Deep reinforcement learning that matters," *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[76] OpenAI, "Universe," 6 December 2016. [Online]. Available: https://openai.com/blog/universe/.

Appendices

Appendix A Code/Pseudo-Code

The code presented here was used in multiple instances with varying parameters. The parameters in this presented code may be only one of many different runs with differing parameters. The code that is given here is not fully comprehensive of what was used. However, it should provide a summary of the used code.

Dqn.py

```
import os
import random
import gym
import pylab
import numpy as np
from collections import deque
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Input, Dense, Lambda, Add
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras import backend as K
import WarCAT_env
import Locker_env

def DQNModel(input_shape, action_space, dueling):
    X_input = Input(input_shape)
    X = X_input

    # 'Dense' is the basic form of a neural network layer
    # Input Layer of state size(4) and Hidden Layer with 512 nodes
    X = Dense(512, input_shape=input_shape, activation="relu", kernel_initializer='he_uniform')(X)

    # Hidden layer with 256 nodes
    X = Dense(256, activation="relu", kernel_initializer='he_uniform')(X)

    # Hidden layer with 64 nodes
    X = Dense(64, activation="relu", kernel_initializer='he_uniform')(X)

    if dueling:
        state_value = Dense(1, kernel_initializer='he_uniform')(X)
        state_value = Lambda(lambda s: K.expand_dims(s[:, 0], -1),
output_shape=(action_space,))(state_value)
```

```
        action_advantage = Dense(action_space, kernel_initializer='he_uniform')(X)
        action_advantage = Lambda(lambda a: a[:, :] - K.mean(a[:, :], keepdims=True),
output_shape=(action_space,))(
            action_advantage)

        X = Add()([state_value, action_advantage])
    else:
        # Output Layer with # of actions: 2 nodes (left, right)
        X = Dense(action_space, activation="linear", kernel_initializer='he_uniform')(X)

    model = Model(inputs=X_input, outputs=X)
    model.compile(loss="mean_squared_error", optimizer=RMSprop(lr=0.00025, rho=0.95,
epsilon=0.01),
                  metrics=["accuracy"])

    model.summary()
    return model


class DQNAgent:
    def __init__(self, env_name):
        self.env_name = env_name
        self.env = WarCAT_env.WarCATGameEnv()
        self.state_size = self.env.observation_space.shape[0]
        self.action_size = self.env.action_space.n

        self.EPISODES = 10000
        self.memory = deque(maxlen=4000)
        self.gamma = 0.95  # discount rate

        # Exploration hyperparameters for epsilon and epsilon greedy strategy
        self.epsilon = 1.0  # exploration probability at start
        self.epsilon_min = 0.01  # minimum exploration probability
        self.epsilon_decay = 0.00005  # exponential decay rate for exploration prob

        self.batch_size = 64

        # defining model parameters
        self.ddqn = True  # use double deep q network
        self.Soft_Update = False  # use soft parameter update
        self.dueling = True  # use dueling network
        self.epsilon_greedy = True  # use epsilon greedy strategy

        self.TAU = 0.1  # target network soft update hyperparameter

        self.Save_Path = 'Models'
        if not os.path.exists(self.Save_Path): os.makedirs(self.Save_Path)
        self.scores, self.episodes, self.average = [], [], []
```

```
        self.Model_name = os.path.join(self.Save_Path, self.env_name + "_e_greedy.h5")

        # create main model and target model
        self.model = DQNModel(input_shape=(self.state_size,), action_space=self.action_size,
dueling=self.dueling)
        self.target_model = DQNModel(input_shape=(self.state_size,), action_space=self.action_size,
                                     dueling=self.dueling)

    # after some time interval update the target model to be same with model
    def update_target_model(self):
        if not self.Soft_Update and self.ddqn:
            self.target_model.set_weights(self.model.get_weights())
            return
        if self.Soft_Update and self.ddqn:
            q_model_theta = self.model.get_weights()
            target_model_theta = self.target_model.get_weights()
            counter = 0
            for q_weight, target_weight in zip(q_model_theta, target_model_theta):
                target_weight = target_weight * (1 - self.TAU) + q_weight * self.TAU
                target_model_theta[counter] = target_weight
                counter += 1
            self.target_model.set_weights(target_model_theta)

    def remember(self, state, action, reward, next_state, done):
        experience = state, action, reward, next_state, done
        self.memory.append((experience))

    def act(self, state, decay_step):
        # Epsilon greedy strategy
        if self.epsilon_greedy:
            # Use an improved version of epsilon greedy strategy for Q-learning
            explore_probability = self.epsilon_min + (self.epsilon - self.epsilon_min) * np.exp(
                -self.epsilon_decay * decay_step)
        # Epsilon strategy
        else:
            if self.epsilon > self.epsilon_min:
                self.epsilon *= (1 - self.epsilon_decay)
            explore_probability = self.epsilon

        if explore_probability > np.random.rand():
            # Make a random action (exploration)
            return random.randrange(self.action_size), explore_probability
        else:
            # Get action from Q-network (exploitation)
            # Estimate the Qs values state
            # Take the biggest Q value (= the best action)
            return np.argmax(self.model.predict(state)), explore_probability

    def replay(self):
```

```
    if len(self.memory) < self.batch_size:
        return
    # Randomly sample minibatch from the memory
    minibatch = random.sample(self.memory, self.batch_size)

    state = np.zeros((self.batch_size, self.state_size))
    next_state = np.zeros((self.batch_size, self.state_size))
    action, reward, done = [], [], []

    # do this before prediction
    # for speedup, this could be done on the tensor level
    # but easier to understand using a loop
    for i in range(self.batch_size):
        state[i] = minibatch[i][0]
        action.append(minibatch[i][1])
        reward.append(minibatch[i][2])
        next_state[i] = minibatch[i][3]
        done.append(minibatch[i][4])

    # do batch prediction to save speed
    # predict Q-values for starting state using the main network
    target = self.model.predict(state)
    # predict best action in ending state using the main network
    target_next = self.model.predict(next_state)
    # predict Q-values for ending state using the target network
    target_val = self.target_model.predict(next_state)

    for i in range(len(minibatch)):
        # correction on the Q value for the action used
        if done[i]:
            target[i][action[i]] = reward[i]
        else:
            if self.ddqn:  # Double - DQN
                # current Q Network selects the action
                # a'_max = argmax_a' Q(s', a')
                a = np.argmax(target_next[i])
                # target Q Network evaluates the action
                # Q_max = Q_target(s', a'_max)
                target[i][action[i]] = reward[i] + self.gamma * (target_val[i][a])
            else:  # Standard - DQN
                # DQN chooses the max Q value among next actions
                # selection and evaluation of action is on the target Q Network
                # Q_max = max_a' Q_target(s', a')
                target[i][action[i]] = reward[i] + self.gamma * (np.amax(target_next[i]))

    # Train the Neural Network with batches
    self.model.fit(state, target, batch_size=self.batch_size, verbose=0)

def load(self, name):
```

```
        self.model = load_model(name)

    def save(self, name):
        self.model.save(name)

    pylab.figure(figsize=(18, 9))

    def PlotModel(self, score, episode, name):
        self.scores.append(score)
        self.episodes.append(episode)
        self.average.append(sum(self.scores[-50:]) / len(self.scores[-50:]))
        pylab.plot(self.episodes, self.average, 'r')
        pylab.plot(self.episodes, self.scores, 'b')
        pylab.ylabel('Score', fontsize=18)
        pylab.xlabel('Steps', fontsize=18)
        dqn = 'DQN_'
        softupdate = ''
        dueling = ''
        greedy = ''
        if self.ddqn: dqn = 'DDQN_'
        if self.Soft_Update: softupdate = '_soft'
        if self.dueling: dueling = '_Dueling'
        if self.epsilon_greedy: greedy = '_Greedy'
        try:
            pylab.savefig(dqn + self.env_name + softupdate + dueling + greedy + name + ".png")
        except OSError:
            pass

        return str(self.average[-1])[:5]

    def train(self):
        decay_step = 0
        for e in range(self.EPISODES):
            state = self.env.reset()
            state = np.reshape(state, [1, self.state_size])
            done = False
            i = 0
            reward_sum = 0
            while not done:
                # self.env.render()
                decay_step += 1
                action, explore_probability = self.act(state, decay_step)
                next_state, reward, done, _ = self.env.step(action)
                next_state = np.reshape(next_state, [1, self.state_size])
                self.remember(state, action, reward, next_state, done)
                state = next_state
                i += 1
                reward_sum += reward
                if done:
```

```
                # every step update target model
                self.update_target_model()

                # every episode, plot the result
                average = self.PlotModel(reward_sum, e, "-train")

                print("episode: {}/{}, score: {}, e: {:.2}, average: {}".format(e, self.EPISODES,
reward_sum,
                                                        explore_probability, average))
                print("Saving trained model to", self.Model_name)
                self.save(self.Model_name)

            self.replay()

    def test(self):
        self.load(self.Model_name)
        for e in range(self.EPISODES):
            state = self.env.reset()
            state = np.reshape(state, [1, self.state_size])
            done = False
            i = 0
            reward_sum = 0
            while not done:
                # self.env.render()
                action = np.argmax(self.model.predict(state))
                next_state, reward, done, _ = self.env.step(action)
                state = np.reshape(next_state, [1, self.state_size])
                i += 1
                reward_sum += reward
                if done:
                    average = self.PlotModel(reward_sum, e, "-test")
                    print("episode: {}/{}, score: {}, average: {}".format(e, self.EPISODES, reward_sum,
average))
                    break


if __name__ == "__main__":
    env_name = 'WarCAT'
    agent = DQNAgent(env_name)
    agent.train()
    agent.test()
```

Stable_baselines3-dqn-warcat.py

```
from stable_baselines3 import DQN
import WarCAT_env
```

```
ENV = WarCAT_env.WarCATGameEnv()
MODEL_TIMESTEPS = 1000000
NUM_TEST_ACTIONS = 10000
MODEL_FILENAME = "dqn-warcat"


model = DQN("MlpPolicy", ENV, device="cuda", verbose=1,
tensorboard_log="./dqn_warcat_tensorboard/")
# model = PPO.load(MODEL_FILENAME, env=env)
model.learn(total_timesteps=MODEL_TIMESTEPS)
model.save(MODEL_FILENAME)

obs = ENV.reset()
ep_len = 0
for i in range(NUM_TEST_ACTIONS):
   action, _states = model.predict(obs, deterministic=True)
   obs, reward, done, info = ENV.step(action)
   print(obs, reward, done, info, action)
   ep_len += 1
   if done:
      obs = ENV.reset()
      print("Episode ended, length = " + str(ep_len))
      ep_len = 0

ENV.close()
```

Stable_baselines3-dqn-locker.py

```
from stable_baselines3 import DQN
import Locker_env

ENV = Locker_env.LockerGameEnv()
MODEL_TIMESTEPS = 1000000
NUM_TEST_ACTIONS = 10000
MODEL_FILENAME = "dqn-locker"


model = DQN("MlpPolicy", ENV, device="cuda", verbose=1,
tensorboard_log="./dqn_locker_tensorboard/")
# model = PPO.load(MODEL_FILENAME, env=env)
model.learn(total_timesteps=MODEL_TIMESTEPS)
model.save(MODEL_FILENAME)

obs = ENV.reset()
ep_len = 0
for i in range(NUM_TEST_ACTIONS):
   action, _states = model.predict(obs, deterministic=True)
   obs, reward, done, info = ENV.step(action)
```

```
    print(obs, reward, done, info, action)
    ep_len += 1
    if done:
        obs = ENV.reset()
        print("Episode ended, length = " + str(ep_len))
        ep_len = 0

ENV.close()
```

WarCAT_env.py

```
import numpy as np
import gym
from gym import spaces
from collections.abc import Iterable


def normalize(x, minx, maxx, lower, upper):
    # return 2*((x-minx)/(maxx-minx)) - 1
    return (upper-lower)*(x-minx)/(maxx-minx) + lower


def normalize_card(x):
    return normalize(x, 0, 14, -1, 1)


def reverse_normalize_card(x):
    return normalize(x, -1, 1, 0, 14)


def normalize_reward(x):
    return normalize(x, -2.0, 2.0, -1, 1)


def normalize_score_basic(x):
    return normalize(x, 0, 5, -1, 1)


def normalize_score_advanced(x):
    return normalize(x, 0, 60, -1, 1)


def flatten(l):
    for el in l:
        if isinstance(el, Iterable) and not isinstance(el, (str, bytes)):
            yield from flatten(el)
        else:
            yield el
```

```
class WarCATGameEnv(gym.Env):

    def __init__(self):
        # Start with 5 actions; pick from one of your cards.
        # As the game progress, this will have to change since the number of cards will decrease as you
choose one.
        self._action_spec = 5
        # Your observation is your own cards (between 2 and 13), the card your opponent played, your
score and your
        # opponent's score.
        # As the game progresses, after you play a card it's value will become a '0'.
        self._observation_spec = 8
        # Observation spec contains your 5 cards in your hand, your opponent's played card,
        # your current score and the opponent's current score
        self.action_space = spaces.Discrete(self._action_spec)
        self.observation_space = spaces.Box(low=-1.0, high=1.0, shape=(self._observation_spec,),
dtype="float32")

        self._totalScore = 0.0
        self._totalOpponentScore = 0.0
        self._deck = DeckOfCards()
        self._deck.shuffleDeck(1000)
        # Player always get dealt cards first
        self._playerHand = self._deck.dealFive()
        self._opponentHand = self._deck.dealFive()
        self._memory = ""
        self._memory += ("Test_init player hand: " + str(self._playerHand) + " bot hand: " + str(
            self._opponentHand) + " score: " + str(self._totalScore) + " opponent score: " +
str(self._totalOpponentScore) +"\n")
        self._episode_ended = False

    def action_spec(self):
        return self._action_spec

    def observation_spec(self):
        return self._observation_spec

    def get_obs(self, opval):
        obs = []
        for x in self._playerHand:
            obs.append(normalize_card(x))
        obs.append(normalize_card(opval))
        obs.append(normalize_score_basic(self._totalScore))
        obs.append(normalize_score_basic(self._totalOpponentScore))
        return obs

    def reset(self):
```

```
        self._totalScore = 0.0
        self._totalOpponentScore = 0.0
        self._deck = DeckOfCards()
        self._deck.shuffleDeck(1000)
        # Player always get dealt cards first
        self._playerHand = self._deck.dealFive()
        self._opponentHand = self._deck.dealFive()
        self._episode_ended = False
        self._memory += ("Test_reset player hand: " + str(self._playerHand) + " bot hand: " + str(
            self._opponentHand) + " score: " + str(self._totalScore) + " opponent score: " +
str(self._totalOpponentScore) +"\n")
        self._info = {"playerHand":self._playerHand, "opponentHand":self._opponentHand}
        # observation
        # obs = list(flatten([self._playerHand, 0, self._totalScore, self._totalOpponentScore]))
        return np.array(self.get_obs(0), dtype="float32")

    # Two ways of handling the score:
    # https://datascience.stackexchange.com/questions/28858/card-game-for-gym-reward-shaping
    # 1. Once you play a card, update the observation of that card to be a 0, and then modify the action
space
    # such that you can no longer play that card.
    # 2. Allow the model to play using any card. If an invalid card is played, do not continue to the next
turn and
    # apply a penalty to the score.
    # I used option 2.
    def step(self, action):
        if self._episode_ended:
            # The last action ended the episode. Ignore the current action and start
            # a new episode.
            return self.reset()

        if action == 0:
            # Do first action
            playerValue = self._playerHand[0]
            self._playerHand[0] = 0
        elif action == 1:
            # Do second action
            playerValue = self._playerHand[1]
            self._playerHand[1] = 0
        elif action == 2:
            # Do second action
            playerValue = self._playerHand[2]
            self._playerHand[2] = 0
        elif action == 3:
            # Do second action
            playerValue = self._playerHand[3]
            self._playerHand[3] = 0
        elif action == 4:
            # Do second action
```

```
            playerValue = self._playerHand[4]
            self._playerHand[4] = 0
        else:
            raise ValueError(`action` should be between 0 to 4.')

        # Remove a random card from our hand if we selected an invalid card
        if playerValue == 0:
            self._memory += ("Invalid card from player hand selected. Action = " + str(action)+ "\n")
            # Remove a card from the player's hand as a penalty
            i = 0
            for card in self._playerHand:
                if card != 0:
                    self._playerHand[i] = 0
                    break
                i = i + 1
            # observation, reward, done, info
            self._info = {"playerHand": self._playerHand, "opponentHand": self._opponentHand,
"playerValue":playerValue}

        # Get the bot's card
        opponentValue = self._botPlaySimple()

        playerScoreRound = self._calculateScore(playerValue, opponentValue)
        opponentScoreRound = self._calculateScore(opponentValue, playerValue)
        self._totalScore += playerScoreRound
        self._totalOpponentScore += opponentScoreRound

        self._memory += ("Player card: " + str(playerValue) + ", Opponent card: " + str(opponentValue) +
                    ", player round score: " + str(playerScoreRound) +
                    ", opponent round score: " + str(opponentScoreRound) +
                    ", player total score: " + str(self._totalScore) +
                    ", opponent total score: " + str(self._totalOpponentScore) + "\n")

        self._info = {"playerHand": self._playerHand, "opponentHand": self._opponentHand,
"playerCard":playerValue, "opponentCard":opponentValue, "playerRoundScore":playerScoreRound,
"opponentRoundScore":opponentScoreRound, "playerTotalScore":self._totalScore,
"opponentTotalScore":self._totalOpponentScore}

        if self._playerHand.count(0) >= 5:
            # If we have played all of our cards, then the game is over.
            if self._totalScore > self._totalOpponentScore:
                # Win
                reward = 2.0
                self._memory += "Game has ended. Player won\n"
            elif self._totalScore == self._totalOpponentScore:
                # Tie
                reward = 1.0
                self._memory += "Game has ended. Tie\n"
            else:
```

```
            # Lose
            reward = -1.0
            self._memory += "Game has ended. Opponent won\n"
        self._episode_ended = True
        # observation, reward, done, info
        return np.array(self.get_obs(opponentValue), dtype="float32"), normalize_reward(reward),
True, self._info

    elif playerValue == 0:
        # Assign penalty if we selected an invalid card
        return np.array(self.get_obs(opponentValue), dtype="float32"), normalize_reward(-2.0), False,
self._info

    else:
        # not the end yet, go to the next play
        self._memory += "Go to the next round.\n"
        # observation, reward, done, info
        return np.array(self.get_obs(opponentValue), dtype="float32"), normalize_reward(0.0), False,
self._info

  # Simple bot will always play cards from highest to lowest
  def _botPlaySimple(self):
    highestCardIndex = self._opponentHand.index(max(self._opponentHand))
    highestCardValue = self._opponentHand[highestCardIndex]
    self._opponentHand[highestCardIndex] = 0
    return highestCardValue

  def _calculateScore(self, playerVal, opponentVal):
    # Use only one of these types of scoring.
    returnScore = 0
    # 1. Basic scoring: you score 1 point if you beat the opponent's card
    if playerVal > opponentVal:
      returnScore += 1
    # # 2. Advanced Scoring: more points for a card close in value to the opponent's card
    # if playerVal > opponentVal:
    #     returnScore = 13 - (playerVal - opponentVal)
    return returnScore


class DeckOfCards:

  def __init__(self):
    self._deck = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 2, 3, 4,
            5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
    self._workingDeck = self._deck
    self._deckIndex = 0

  def dealFive(self):
    dealHand = []
```

```
        for i in range(5):
            dealHand.append(self._workingDeck[self._deckIndex])
            self._deckIndex += 1
        return dealHand

    def shuffleDeck(self, iterations):
        for i in range(iterations):
            index1 = np.random.randint(0, 52)
            index2 =  np.random.randint(0, 52)
            temp = self._workingDeck[index1]
            self._workingDeck[index1] = self._workingDeck[index2]
            self._workingDeck[index2] = temp
        self._deckIndex = 0

    def resetDeck(self):
        self._workingDeck = self._deck
        self._deckIndex = 0
```

Locker_env.py

```
import numpy as np
import gym
from gym import spaces


def normalize(x, minx, maxx, lower, upper):
    #return 2*((x-minx)/(maxx-minx)) - 1
    return (upper-lower)*(x-minx)/(maxx-minx) + lower


def normalize_dial(x):
    return normalize(x, 1, 5, -1, 1)


def reverse_normalize_dial(x):
    return normalize(x, -1, 1, 1, 5)


def normalize_score(x):
    return normalize(x, 3, 15, -1, 1)


def reverse_normalize_score(x):
    return normalize(x, -1, 1, 3, 15)


def normalize_reward(x):
    return normalize(x, -100, 100, -1, 1)
```

```
class LockerGameEnv(gym.Env):
    def __init__(self):
        self._action_spec = 4  # Total number of actions you can take
        self._observation_spec = 4  # Total number of observations that will be returned
        self.action_space = spaces.Discrete(self._action_spec)
        self.observation_space = spaces.Box(low=-1.0, high=1.0, shape=(self._observation_spec,),
dtype="float32")
        self.payoffMatrix = [[1, 5, 4, 3, 2], [2, 1, 5, 4, 3], [3, 2, 1, 5, 4], [4, 3, 2, 1, 5], [5, 4, 3, 2, 1]]
        self._dial_one = 1
        self._dial_two = 1
        self._dial_three = 1
        self._random_one = np.random.randint(1, 6)
        self._random_two = np.random.randint(1, 6)
        self._random_three = np.random.randint(1, 6)
        self._score = 0.0
        self._highest_score = 0.0
        self._penalty = 0.0
        self._episode_ended = False
        self._memory = ""
        self._memory += ("Test_init dial_one: " + str(self._dial_one) + " dial_two: " + str(self._dial_two)
+ " dial_three: " + str(self._dial_three) + "\n")
        self._memory += ("Test_init random_one: " + str(self._random_one) + " random_two: " +
str(self._random_two) + " random_three: " + str(self._random_three) + "\n")
        self._info = {"dial_one": self._dial_one, "dial_two": self._dial_two, "dial_three": self._dial_three,
"soln_one": self.solver(self._random_one), "soln_two": self.solver(self._random_two), "soln_three":
self.solver(self._random_three), "num_actions":self._penalty}

    def solver(self, x):
        x = x - 1
        solution = -1
        for i in range(0,5):
            if self.payoffMatrix[i][x] == 5:
                solution = i
        return solution + 1

    def action_spec(self):
        return self._action_spec

    def observation_spec(self):
        return self._observation_spec

    def state(self):
        return [self._dial_one, self._dial_two, self._dial_three, self._score]

    def reset(self):
        self.payoffMatrix = [[1, 5, 4, 3, 2], [2, 1, 5, 4, 3], [3, 2, 1, 5, 4], [4, 3, 2, 1, 5], [5, 4, 3, 2, 1]]
        self._dial_one = 1
```

```
        self._dial_two = 1
        self._dial_three = 1
        self._random_one = np.random.randint(1, 6)
        self._random_two = np.random.randint(1, 6)
        self._random_three = np.random.randint(1, 6)
        self._score = 3.0
        self._highest_score = 3.0
        self._penalty = 0.0
        self._episode_ended = False
        self._memory += ("Test_reset dial_one: " + str(self._dial_one) + " dial_two: " +
str(self._dial_two) + " dial_three: " + str(self._dial_three) + "\n")
        self._memory += ("Test_reset random_one: " + str(self._random_one) + " random_two: " +
str(self._random_two) + " random_three: " + str(self._random_three) + "\n")
        # return ts.restart(np.array([self._dial_one, self._dial_two, self._dial_three, self._score,
self._penalty],
        #                            dtype=np.int32))
        # observation
        self._info = {"dial_one": self._dial_one, "dial_two": self._dial_two, "dial_three": self._dial_three,
"soln_one": self.solver(self._random_one), "soln_two": self.solver(self._random_two), "soln_three":
self.solver(self._random_three), "num_actions":self._penalty}
        return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32")

    def step(self, action):
        if self._episode_ended:
            # The last action ended the episode. Ignore the current action and start
            # a new episode.
            return self.reset()

        new_score = 0
        reward = 0
        self._penalty = self._penalty + 1
        if action == 0:
            self._memory += ("Performed action: dial one increase\n")
            self._dial_one = self._dial_one + 1 if self._dial_one < 5 else 1
        elif action == 1:
            self._memory += ("Performed action: dial two increase\n")
            self._dial_two = self._dial_two + 1 if self._dial_two < 5 else 1
        elif action == 2:
            self._memory += ("Performed action: dial three increase\n")
            self._dial_three = self._dial_three + 1 if self._dial_three < 5 else 1
        elif action == 3:
            # 1. Get score from payoff matrix
            new_score = self.payoffMatrix[self._dial_one - 1][self._random_one - 1] + \
                        self.payoffMatrix[self._dial_two - 1][self._random_two - 1] + \
                        self.payoffMatrix[self._dial_three - 1][self._random_three - 1]
            # 2. Simple scoring for testing
            # new_score = self._dial_one + self._dial_two + self._dial_three
```

```
            reward = new_score - self._highest_score
            if new_score > self._highest_score:
                self._highest_score = new_score
            self._score = new_score
            self._memory += ("Performed action: attempt unlock. Score = " + str(self._score) + " Current
number of attempts: " + str(self._penalty) + "\n")

        else:
            raise ValueError('`action` should be between 0 and 3.')

        self._memory += ("Test_step dial_one: " + str(self._dial_one) + " dial_two: " + str(self._dial_two)
+
                         " dial_three: " + str(self._dial_three) + "\n")
        self._info = {"dial_one": self._dial_one, "dial_two": self._dial_two, "dial_three": self._dial_three,
"soln_one": self.solver(self._random_one), "soln_two": self.solver(self._random_two), "soln_three":
self.solver(self._random_three), "num_actions":self._penalty}

        if self._score >= 15:
            self._memory += ("Unlock successful. Number of unlock attempts:" + str(self._penalty) +
"\n\n")
            # observation, reward, done, info
            self._episode_ended = True
            # return [self._dial_one, self._dial_two, self._dial_three, self._score], (5*5*5 - self._penalty),
True, self._memory
            return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32"),
normalize_reward(100.0), True, self._info

        elif (1000 - self._penalty) <= 0:
            self._memory += ("Unlock unsuccessful. Maximum attempts reached:" + str(self._penalty) +
"\n\n")
            self._episode_ended = True
            # return [self._dial_one, self._dial_two, self._dial_three, self._score], 0, True, self._memory
            return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32"), normalize_reward(-
100.0), True, self._info

        elif action == 3:
            # return ts.transition(np.array([self._dial_one, self._dial_two, self._dial_three, self._score,
self._penalty],
            #                               dtype=np.int32), reward=reward, discount=1.0)
            # observation, reward, done, info
            # Reward structure 1:
            # return [self._dial_one, self._dial_two, self._dial_three, self._score], reward, False,
self._memory
            # Reward structure 2: return a -1 on unsuccessful unlock
            return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32"), normalize_reward(-
25.0), False, self._info
```

```
        else:
            # return ts.transition(np.array([self._dial_one, self._dial_two, self._dial_three, self._score,
self._penalty],
            #                              dtype=np.int32), reward=0.0, discount=1.0)
            # observation, reward, done, info
            return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32"), normalize_reward(-
25.0), False, self._info
```

Locker_env_nounlock_norollover.py

```
import numpy as np
import gym
from gym import spaces


def normalize(x, minx, maxx, lower, upper):
    #return 2*((x-minx)/(maxx-minx)) - 1
    return (upper-lower)*(x-minx)/(maxx-minx) + lower


def normalize_dial(x):
    return normalize(x, 1, 5, -1, 1)


def reverse_normalize_dial(x):
    return normalize(x, -1, 1, 1, 5)


def normalize_score(x):
    return normalize(x, 3, 15, -1, 1)


def reverse_normalize_score(x):
    return normalize(x, -1, 1, 3, 15)


def normalize_reward(x):
    return normalize(x, -100, 100, -1, 1)


class LockerGameEnv(gym.Env):
    def __init__(self):
        self._action_spec = 6  # Total number of actions you can take
        self._observation_spec = 4  # Total number of observations that will be returned
        self.action_space = spaces.Discrete(self._action_spec)
```

```
        self.observation_space = spaces.Box(low=-1.0, high=1.0, shape=(self._observation_spec,),
dtype="float32")
        self.payoffMatrix = [[1, 5, 4, 3, 2], [2, 1, 5, 4, 3], [3, 2, 1, 5, 4], [4, 3, 2, 1, 5], [5, 4, 3, 2, 1]]
        self._dial_one = 1
        self._dial_two = 1
        self._dial_three = 1
        self._random_one = np.random.randint(1, 6)
        self._random_two = np.random.randint(1, 6)
        self._random_three = np.random.randint(1, 6)
        self._score = 0.0
        self._highest_score = 0.0
        self._penalty = 0.0
        self._episode_ended = False
        self._memory = ""
        self._memory += ("Test_init dial_one: " + str(self._dial_one) + " dial_two: " + str(self._dial_two)
+ " dial_three: " + str(self._dial_three) + "\n")
        self._memory += ("Test_init random_one: " + str(self._random_one) + " random_two: " +
str(self._random_two) + " random_three: " + str(self._random_three) + "\n")
        self._info = {"dial_one": self._dial_one, "dial_two": self._dial_two, "dial_three": self._dial_three,
"soln_one": self.solver(self._random_one), "soln_two": self.solver(self._random_two), "soln_three":
self.solver(self._random_three), "num_actions":self._penalty}

    def solver(self, x):
        x = x - 1
        solution = -1
        for i in range(0,5):
            if self.payoffMatrix[i][x] == 5:
                solution = i
        return solution + 1

    def action_spec(self):
        return self._action_spec

    def observation_spec(self):
        return self._observation_spec

    def state(self):
        return [self._dial_one, self._dial_two, self._dial_three, self._score]

    def reset(self):
        self.payoffMatrix = [[1, 5, 4, 3, 2], [2, 1, 5, 4, 3], [3, 2, 1, 5, 4], [4, 3, 2, 1, 5], [5, 4, 3, 2, 1]]
        self._dial_one = 1
        self._dial_two = 1
        self._dial_three = 1
        self._random_one = np.random.randint(1, 6)
        self._random_two = np.random.randint(1, 6)
        self._random_three = np.random.randint(1, 6)
        self._score = 3.0
        self._highest_score = 3.0
```

```
        self._penalty = 0.0
        self._episode_ended = False
        self._memory += ("Test_reset dial_one: " + str(self._dial_one) + " dial_two: " +
str(self._dial_two) + " dial_three: " + str(self._dial_three) + "\n")
        self._memory += ("Test_reset random_one: " + str(self._random_one) + " random_two: " +
str(self._random_two) + " random_three: " + str(self._random_three) + "\n")
        # return ts.restart(np.array([self._dial_one, self._dial_two, self._dial_three, self._score,
self._penalty],
        #                            dtype=np.int32))
        # observation
        self._info = {"dial_one": self._dial_one, "dial_two": self._dial_two, "dial_three": self._dial_three,
"soln_one": self.solver(self._random_one), "soln_two": self.solver(self._random_two), "soln_three":
self.solver(self._random_three), "num_actions":self._penalty}
        return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32")

    def step(self, action):
        if self._episode_ended:
            # The last action ended the episode. Ignore the current action and start
            # a new episode.
            return self.reset()

        new_score = 0
        reward = 0
        self._penalty = self._penalty + 1
        if action == 0:
            self._memory += ("Performed action: dial one increase\n")
            self._dial_one = self._dial_one + 1
            if self._dial_one > 5:
                self._dial_one = 5
        elif action == 1:
            self._memory += ("Performed action: dial two increase\n")
            self._dial_two = self._dial_two + 1
            if self._dial_two > 5:
                self._dial_two = 5
        elif action == 2:
            self._memory += ("Performed action: dial three increase\n")
            self._dial_three = self._dial_three + 1
            if self._dial_three > 5:
                self._dial_three = 5
        elif action == 3:
            self._memory += ("Performed action: dial one decrease\n")
            self._dial_one = self._dial_one - 1
            if self._dial_one < 1:
                self._dial_one = 1
        elif action == 4:
            self._memory += ("Performed action: dial two decrease\n")
            self._dial_two = self._dial_two - 1
            if self._dial_two < 1:
```

```
            self._dial_two = 1
        elif action == 5:
            self._memory += ("Performed action: dial three decrease\n")
            self._dial_three = self._dial_three - 1
            if self._dial_three < 1:
                self._dial_three = 1
        else:
            raise ValueError('`action` should be between 0 and 5.')

        # 1. Get score from payoff matrix
        new_score = self.payoffMatrix[self._dial_one - 1][self._random_one - 1] + \
                    self.payoffMatrix[self._dial_two - 1][self._random_two - 1] + \
                    self.payoffMatrix[self._dial_three - 1][self._random_three - 1]
        # 2. Simple scoring for testing
        # new_score = self._dial_one + self._dial_two + self._dial_three

        reward = new_score - self._highest_score
        if new_score > self._highest_score:
            self._highest_score = new_score
        self._score = new_score
        self._memory += ("Performed action: attempt unlock. Score = " + str(self._score) + " Current
number of attempts: " + str(self._penalty) + "\n")

        self._memory += ("Test_step dial_one: " + str(self._dial_one) + " dial_two: " + str(self._dial_two)
+
                         " dial_three: " + str(self._dial_three) + "\n")
        self._info = {"dial_one": self._dial_one, "dial_two": self._dial_two, "dial_three": self._dial_three,
"soln_one": self.solver(self._random_one), "soln_two": self.solver(self._random_two), "soln_three":
self.solver(self._random_three), "num_actions":self._penalty}

        if self._score >= 15:
            self._memory += ("Unlock successful. Number of unlock attempts:" + str(self._penalty) +
"\n\n")
            # observation, reward, done, info
            self._episode_ended = True
            # return [self._dial_one, self._dial_two, self._dial_three, self._score], (5*5*5 - self._penalty),
True, self._memory
            return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32"),
normalize_reward(100.0), True, self._info

        elif (1000 - self._penalty) <= 0:
            self._memory += ("Unlock unsuccessful. Maximum attempts reached:" + str(self._penalty) +
"\n\n")
            self._episode_ended = True
            # return [self._dial_one, self._dial_two, self._dial_three, self._score], 0, True, self._memory
            return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32"), normalize_reward(-
100.0), True, self._info
```

```
      else:
          # return ts.transition(np.array([self._dial_one, self._dial_two, self._dial_three, self._score,
self._penalty],
          #                              dtype=np.int32), reward=0.0, discount=1.0)
          # observation, reward, done, info
          return np.array([normalize_dial(self._dial_one), normalize_dial(self._dial_two),
normalize_dial(self._dial_three), normalize_score(self._score)], dtype="float32"), normalize_reward(-
25.0), False, self._info
```