Data Synthesis and Classification through Machine Learning for Detecting Mild Cognitive Impairment

by

Mahmood Aljumaili

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, Manitoba

ABSTRACT

This thesis examines Machine Learning (ML) applicable to detecting Mild Cognitive Impairment (MCI) from gameplay data of a Serious Game (SG). The foundation of the work is a newly developed SG called WAR Cognitive Assessment Tool (WarCAT). WarCAT is based on the familiar card game WAR and is played on mobile devices (phones, tablets). WarCAT captures players' moves during the game to allow inference of cognitive processes of strategy recognition, learning, and memory. This thesis focuses on the potential of detecting MCI by developing a data synthesis model and a classifier model to be applied to WarCAT gameplay. The data synthesis model generates WarCAT gameplay data similar to that of a human, so that it can be used to train and validate the classifier model. The classifier model has the potential to detect MCI from a player's gameplay data. Machine Learning (ML) methods are used by both models. The classifier uses a deep Artificial Neural Network (ANN) that is first trained on a large labelled dataset to help detect MCI. The data synthesis model generates synthetic data to plausibly emulate a large population of players. The sub-paradigm of ML, namely Reinforcement Learning (RL) is used to generate synthetic data, as it most closely emulates the way humans learn. An RL bot undergoes millions of sessions of trial and error (training), processing millions of gameplay training patterns and achieves results comparable or better than the best human performance. A conjecture is that an RL bot with fewer training opportunities emulates a person with less experience or potentially declining cognitive function. This baseline allows for the creation of bots with the goal to emulate individuals at various stages of learning, or conversely, various levels of cognitive decline. The thesis demonstrates the ML framework necessary to generate and classify different levels of play, which correspond to different levels of players' cognitive states or abilities.

ACKNOWLEDGMENTS

I would like to thank Bob and Marcia for the opportunity to join their team, for their support, and for being great mentors. Also, a special thanks to Kai for the insightful discussions. I would also like to thank my labmates: Shachi, Kyle, Ryan, Sanyat, and the others who spent some time in the lab for the stimulating discussions and for the time together. Finally, I would like to thank the University of Manitoba for the support through this master degree.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AD	Alzheimer's Disease
ANN	Artificial Neural Network
CDT	Clock Drawing Test
CNN	Convolutional Neural Network
DNN	Dense Neural Network
FFT	Fast-Fourier Transform
MCI	Mild Cognitive Impairment
ML	Machine Learning
MMSE	Mini-Mental State Examination
MoCA	Montreal Cognitive Assessment
RL	Reinforcement Learning
SG	Serious Game
WarCAT	War Cognitive Assessment Tool

Chapter 1: Introduction

1.1 Background & Motivation

This work focuses on analyzing data from games from which to potentially detect and assess Mild Cognitive Impairment (MCI) utilizing Machine Learning (ML) models. Early detection of MCI is a challenging task because of its subtle symptoms. MCI symptoms reflect a decline in cognitive abilities that may be substantial enough to be noticed by the patient or the individuals surrounding the patient, but not substantial enough to affect the basic daily routine life tasks. MCI affects memory (recall, retention), thinking, and judgment to higher degree than the degree of similar cognitive decline that comes normally with aging. Approximately, 10-20% of adults aged 65 years or older have some form of MCI [1]. Multiple studies worldwide reported MCI prevalence ranging from 3% to as high as 42% [2]. The most concerning serious issue about MCI is that it increases the probability risk of more serious dementia as has been found in [3] where it was reported that 20-40% (10–15% per year) of cases of MCI progress to dementia. According to Alzheimer Disease International's 2018 report [4], the estimated number of people with dementia is 50 million worldwide, and is expected to rise to 152 million by 2050. The total estimated cost worldwide is US$ 1 Trillion in 2018. This figure is projected to rise to US$ 2 Trillion by 2030.

However, not all MCI cases lead to dementia. According to some estimates, 4% to 15% of MCI cases revert back to normal in clinic-based studies [5], [6], and 29% to 55% in population-based studies [7], [8]. If MCI can be detected early, then perhaps the later progression can be diverted or alleviated with therapeutics or other psycho-social interventions to improve or maintain cognitive abilities. It is well recognized that early detection of any health concern usually leads to a more positive prognosis, and early diagnosis of cognitive impairment is no exception [9], [10].

However, the earlier the stage of cognitive decline or MCI, the less observable are the symptoms and the harder it is to detect. On the other hand, the earlier the detection, the greater are the benefits in effective treatment. Serious dementia is significantly easier to detect than MCI, but the therapeutics might be less effective as the disease has progressed to the point beyond being treatable. Although, no evidence has yet been found of MCI improvement by pharmacologic treatments, non-pharmacologic treatments such as exercise programs, cognitive training and rehabilitation were found to have a small but significant difference favoring intervention when compared with controls (MD 1.01, 95% CI 0.25 to 1.77) assessed with MMSE [11]. Of course, any kind of therapy is contingent on MCI being discovered early.

This work is built upon the foundation of a Serious Game (SG) for mobile devices that was previously developed in [12]. An SG is defined as a game that was designed for a purpose other than entertainment. The SG used here is based on the familiar card game War and denoted WarCAT (War Cognitive Assessment Tool). This work includes an overview of the game to provide context for the SG used in the subsequent ML work. It is presented only as a platform for the ML work that follows, in that the objective of this thesis is to discuss the machine learning approaches used to generate synthetic gameplay data using RL approaches, and ML-based classification of gameplay data for various degrees of impairment. The used ML approaches are outlined, including the justifications, opportunities, and trade-offs.

1.2 Related Work

SGs can be a rich source of data for mental health [13], [14], and while there are other games on the market that claim to do 'brain training' to maintain cognitive function or potentially delay MCI, limited work has been done on games that try to detect or assess MCI — a subtle but important distinction. Although there is considerable research activity in developing and analyzing

SGs for cognitive health [15]–[20], the role of SGs for any range of genuine health applications is embryonic at best and considerably ad-hoc, with some effort being made to more formally organize the topic [21].

The common methods of detecting MCI are the cognitive screening tests (also referred to as neuropsychological testing). Typically, there are questions or tasks that are asked of the individual being evaluated to answer or perform. In most cases, a trained clinician performs those tests. 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 1-1 below.

Table 1-1 A Comparison of Different Well-known MCI Screening Tests [22]

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, unlike detection through a SG such as WarCAT, those screening tests usually require whoever is administering them to be trained or at least follow given instructions to conduct the test properly and not interfere with the screening process. In the case of MMSE, the instructions given to administrators include, for instance, avoiding emotional reactions that could compromise cooperation and performance [23]. Another instruction example for the case of CDT; “Consider physical impairment including any issues with muscles in the hand/arm or vision”; “Remove/cover any clocks in the room and be aware if participants have a wrist watch on during this test” [24]. Also, they are not engaging or fun, and most are not easily accessible on smartphones besides MoCA test, and they are not amenable for advanced mathematical algorithms or scaling in general.

1.3 WarCAT

To construct any technique has the potential to detect MCI, a platform is required to serve as an intermediary between the person and the detection software system. Such a platform would convert the cognitive choices and actions of the player into raw data. A serious game can be used as such a platform as it affords the opportunity to convert the player’s performance (moves and decisions made) into numerical input data. If the selected game’s performance depends on certain cognitive functions, then the performance of those cognitive functions would be embedded in the gameplay data, thus giving a chance for the detection software systems to discern them. The serious game WarCAT is used for exactly such a role. In general, a serious game is one in which a primary intention is data collection as opposed to entertainment. It should be noted, that if a serious game were to be developed that could be used directly to help assess cognitive difficulties it would have to be as popular as Candy Crush™. In that scenario, the number of users themselves would provide baseline data as well as data to classify play against. There would likely still be an opportunity for ML as a classifier, whereas within this thesis, ML is also used to generate synthetic data against which gameplay can be classified.

1.3.1 What is WarCAT?

WarCAT is serious card game. It has been discussed in [12] and was designed specifically to serve as a platform for developing models for potentially detecting MCI. It is very simple and it is a modified version of the popular card game WAR. In this variation, the human player is playing with five cards that can be played in the order of their choice against an opponent bot or computer that is playing in a deterministic fashion (i.e. using a set strategy such as low to high cards, high to low cards, etc.). Figure 1-1 illustrates an instance of play during one round of WarCAT.

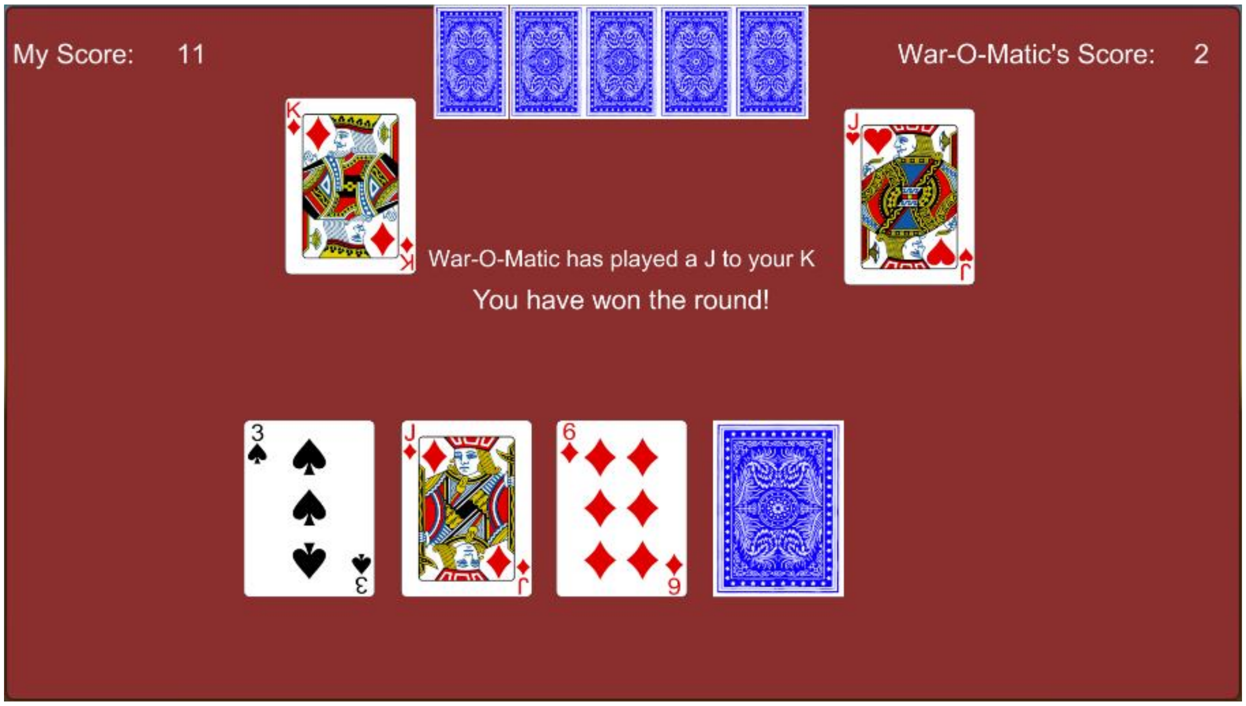


Figure 1-1 Instance of playing one round of WarCAT

Upon starting the game, the player needs to register so that their gameplay data get stored under that player's user name as shown in Figure 1-2(b) below. The type of information collected and any additional information is illustrative of the type of data that would be most desirable if a sufficient user community were to be established. This comes with its own ethical responsibilities but merely points out that the game cannot be totally anonymous and further discussion is not part of the actual thesis considered here. However, the importance or relevance of this type of data is discussed in the sections below.

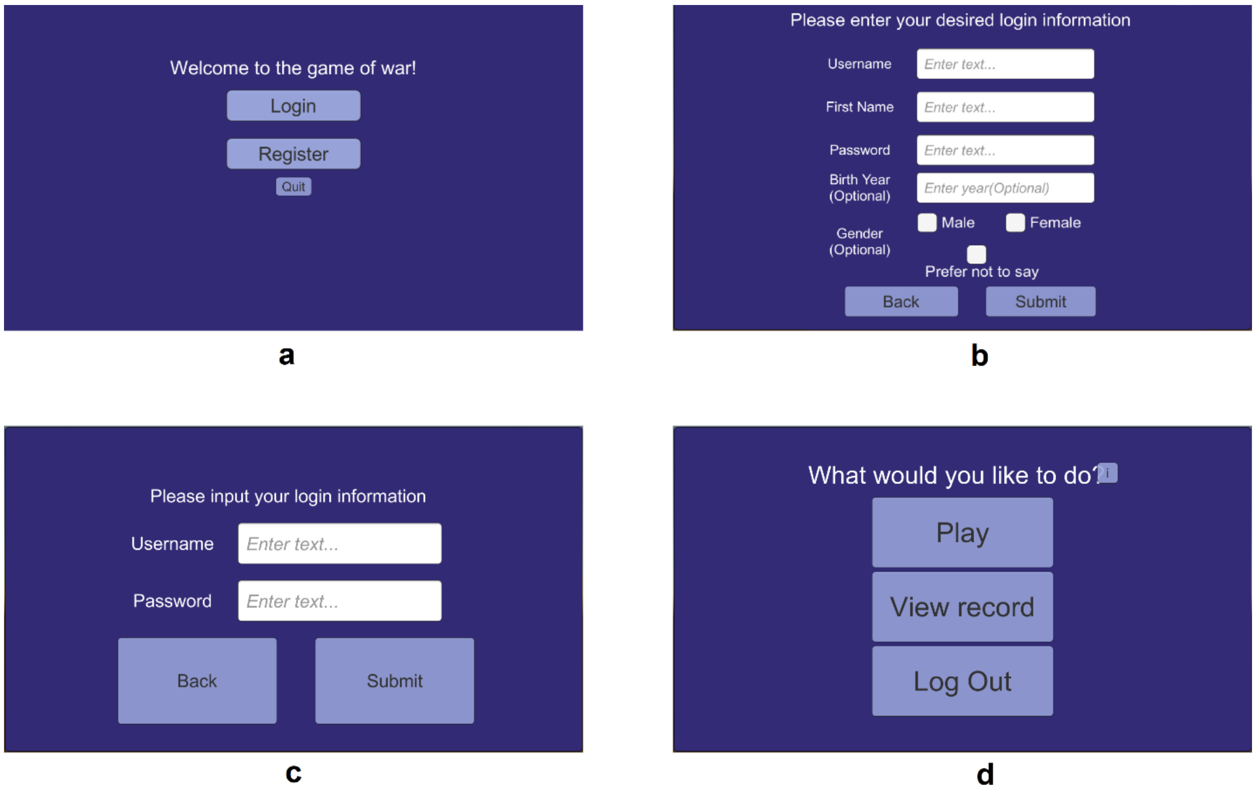


Figure 1-2 (a) initial screen, (b) new user registration, (c) log in screen, (d) menu screen after log in

<u>The game rules are as follows:</u>

- 5 cards are dealt to both the player and an opponent bot (computer).
- Both play one card at a time at the same time. Hence it is concurrent (i.e. both players pick a card, then both cards are revealed at the same time), as opposed to many games which are turn based.
- 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:

$$Score = 13 - (Winning\ Card\ Value - Losing\ Card\ Value) \tag{1-1}$$

  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 (1-2) below.

$$Aggregate\ Score = \sum_{i=1}^{5} Score_i \tag{1-2}$$

Every time a card is played from both players, a *round* has been played. Every five *rounds* constitute a *game*. Every 50 or 150 games constitute a *level*. In the initial work later in Section 4.1 below, 50 games were used. However, 150 games were used later in Section 5.2 below to improve the data. This type of level play is common within games where a level is typically associated with a degree of difficulty. Here rounds, games, and levels are used to compartmentalize the data collected associated with play.

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. In that, the scoring is more heavily weighted to a narrow victory over a larger difference in card values. In this manner, according to (1-1) 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 was with a narrower margin, therefore; it is more favorable.

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*. So, the bot plays with the same *strategy* throughout the 50 or 150 *games* or one *level*. An example of a strategy would be playing the highest card first, then the second highest…etc. 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 – both in *recognizing* that a strategy is being played, and then *learning* what the strategy is. Then, the player can predict the order although not the exact cards 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, i.e. the player would counter the bot's *strategy*. It's important to note that the player can predict or learn the order of the bot's cards but not the cards' values. One example of countering a strategy, 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 i.e. deliberately losing (sacrificing) that card 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 1-3 below.

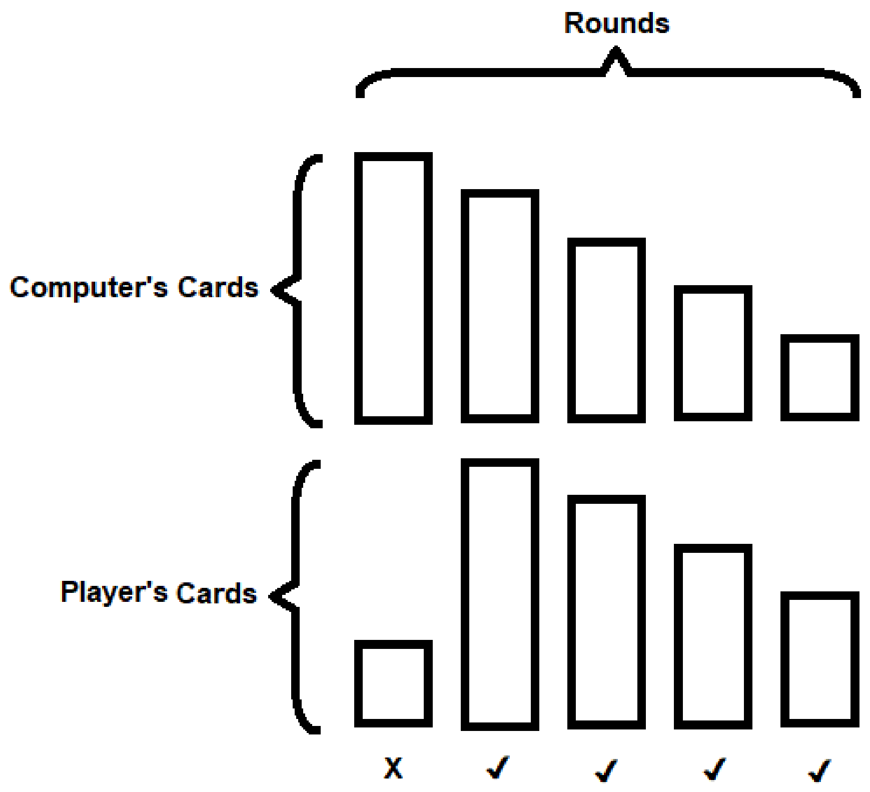
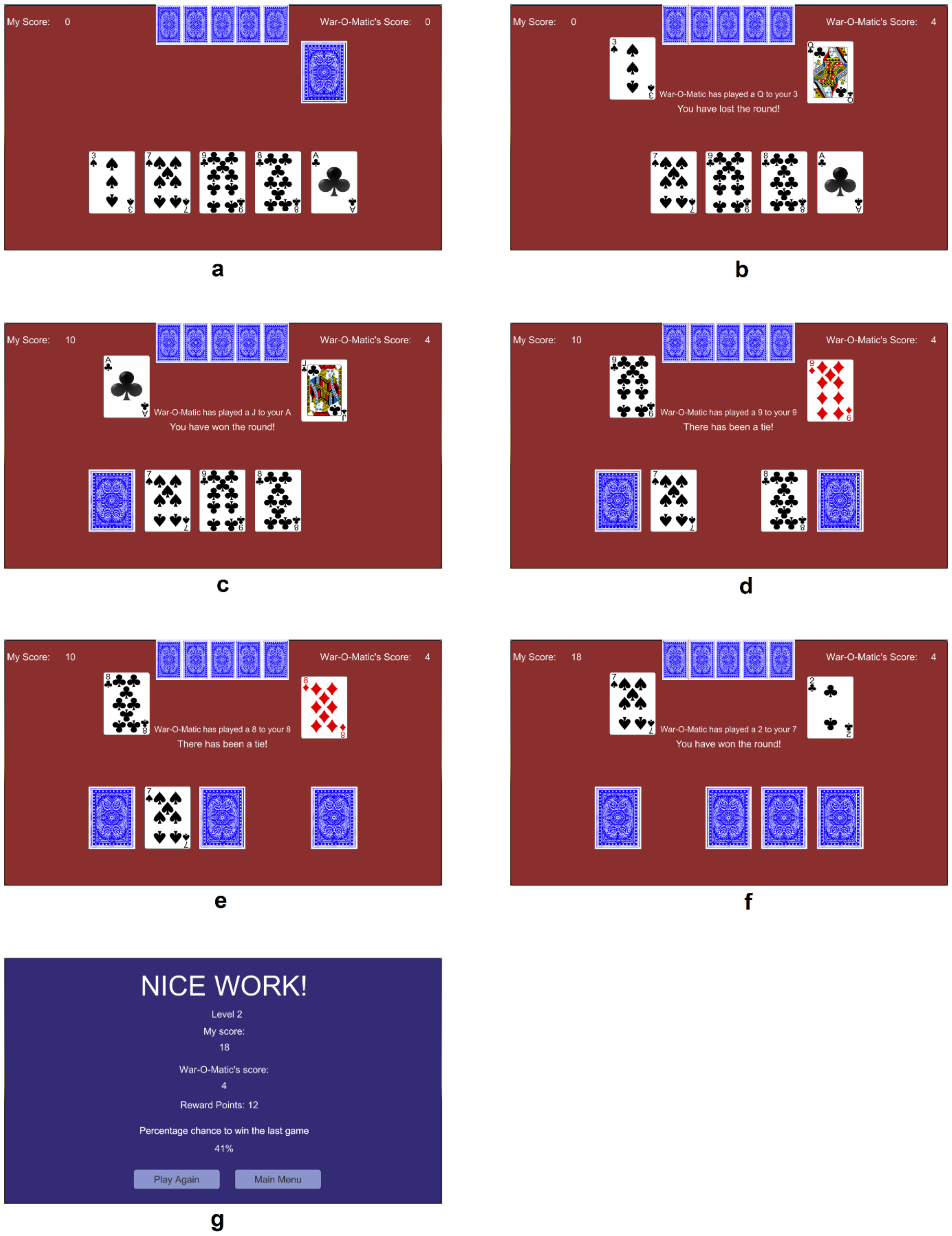


Figure 1-3 One Example of the Player's Countering Play Against A Known Strategy

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, on average on many games played, the average card value is higher, hence the reason for a stream of 50 or 150 *games* per level. Basically, the player (on average) would lose the first *round*, but would win (on average) the other four *rounds*, and ultimately win (on average) the *game*. 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 1-4 below. In Figure 1-4(a), each player gets five cards, then the player picks a card out of five to play. In Figure 1-4(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 (1-1) above, the bot’s amount of points from the first round is 4. In Figure 1-4(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.



a
My Score: 0
War-O-Matic's Score: 0

b
My Score: 0
War-O-Matic's Score: 4
War-O-Matic has played a Q to your 3
You have lost the round!

c
My Score: 10
War-O-Matic's Score: 4
War-O-Matic has played a J to your A
You have won the round!

d
My Score: 10
War-O-Matic's Score: 4
War-O-Matic has played a 9 to your 9
There has been a tie!

e
My Score: 10
War-O-Matic's Score: 4
War-O-Matic has played a 8 to your 8
There has been a tie!

f
My Score: 18
War-O-Matic's Score: 4
War-O-Matic has played a 2 to your 7
You have won the round!

g
NICE WORK!
Level 2
My score:
18
War-O-Matic's score:
4
Reward Points: 12
Percentage chance to win the last game
41%

Figure 1-4 an Example of One Gameplay of WarCAT

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 (e.g. stroke, tumor, etc.), 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, medical history etc. are also important factors that a SG 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, i.e. 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/complex, and being simple and yet able to provide a detailed data trail of play.

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 [1]. Also, the game records the time took by the player to play each card 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/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 ML 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 may not always be an Ace or King, etc. This randomness factor makes WarCAT a suitable platform as it insures 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.

The highest winning percentage is 92.70% and a score average of 25. However, this only possible if the player always has perfect knowledge of all of the opponent's cards and strategy in advance. So, the achievable numbers for the winning percentage and the score average are below these formerly mentioned upper limits.

1.3.3 Data Representation

For every *game* played, the game records (under username handle or any suitable pseudo identifier) the ten cards dealt to both the player and the computer bot, the order in which they were played, the probability of winning, the maximum score, the expected score, the winner, and the score result. From these data, the relevant data representations are extracted. Other types of data are recorded for potential future use, such as the time spent in decision making, but they are not used in this work. One level of data would consist of a stream of 50 or 150 games. This stream is referred to as the "cognitive fingerprint" [25]. Figure 1-5 shows an example of "fingerprint" data. The quantization nature of the data in the shown example is due to normalization with respect to the highest possible score. This normalization is explained further in Section 5.3.2 below. Further gameplay data samples are illustrated in Appendix A. From this data, considerable information can be inferred that align with strategy recognition, learning, and retention, i.e. whether the player recognized and countered the bot's strategy, was consistently beating the bot, or was getting confused (suffering from momentary "loss of set" – i.e. forgetting the strategy or was distracted). Essentially, one can initially infer whether the player won when they should have won, lost when they should have won, won when they should have lost, and lost when they should have lost, all based on the random set of cards dealt to them and to the bot in that hand.

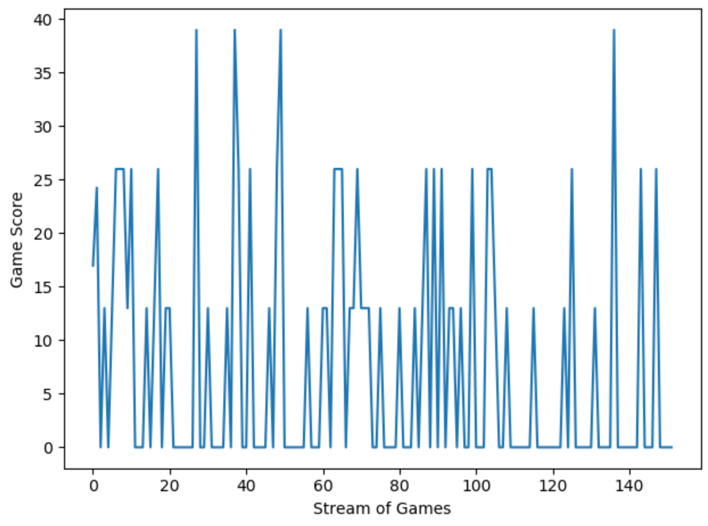


Figure 1-5 An early example data visualization of normalized "cognitive fingerprint" over 150 games

1.4 Machine Learning Background

There are three basic ML paradigms: Unsupervised Learning, Supervised Learning, and Reinforcement Learning [26]. Each paradigm consists of many ML algorithms. No algorithm from Unsupervised Learning is used in this work. This work uses algorithms from Supervised Learning and algorithms from Reinforcement Learning.

Supervised Learning is the most widely used paradigm of ML and the most successful. It is generally used in classification or pattern recognition. Supervised Learning algorithms have shown incredible success since the (rather shocking) better than human performance of image classification in 2011 [27]. Since then, more attention has been paid to various ML algorithms that have worked well in other applications such as: detecting cancer, detecting spam email, object recognition and/or tracking in images and videos, speech-to-text, etc.

The way Supervised Learning algorithms work is typically by training models on large sets of data to improve performance. Those data are labelled. 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 data then give an output. Then the given output is compared with the labeled (correct) output. If they match then no adjusting/correction needed, otherwise the model is adjusted towards giving a matching output. The adjusting process will be detailed in Section 1.4.1 below. The training process keeps reiterating until the model can produce an output that matches the labeled output or close to it. That is, the model had now learned how to classify or detect the correctly labeled data. By extension, it can then go on to label new data correctly.

In Unsupervised Learning, the data do not come with labels. There is a smaller variety of useful applications than Supervised Learning. Unsupervised learning applications include: clustering of semantics, grouping news articles about similar topics, anomaly detection, and DNA sequencing. This ML paradigm is not used in this work and only mentioned for completeness.

The third basic paradigm is Reinforcement Learning (RL). 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. RL 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 RL 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 (i.e. path of actions) that leads to the desired endpoint. RL is used in applications such as: playing games, industrial automation, self-driving cars, stock trading, optimization, path finding, etc. RL will be explained in more details in Section 5.2.3 below.

1.4.1 Artificial Neural Networks

Artificial Neural Network (ANN), a Supervised Learning algorithm, is a data processing system that consists of connected neurons typically grouped in stacked layers. A neuron in one layer is connected to some or all the neurons in the neighbouring layers on either side. The data goes through the ANN in one direction layer by layer: input layer, hidden layers, and output layer, Figure 1-6. When the ANN is made of multiple layers, it is then called deep neural network.

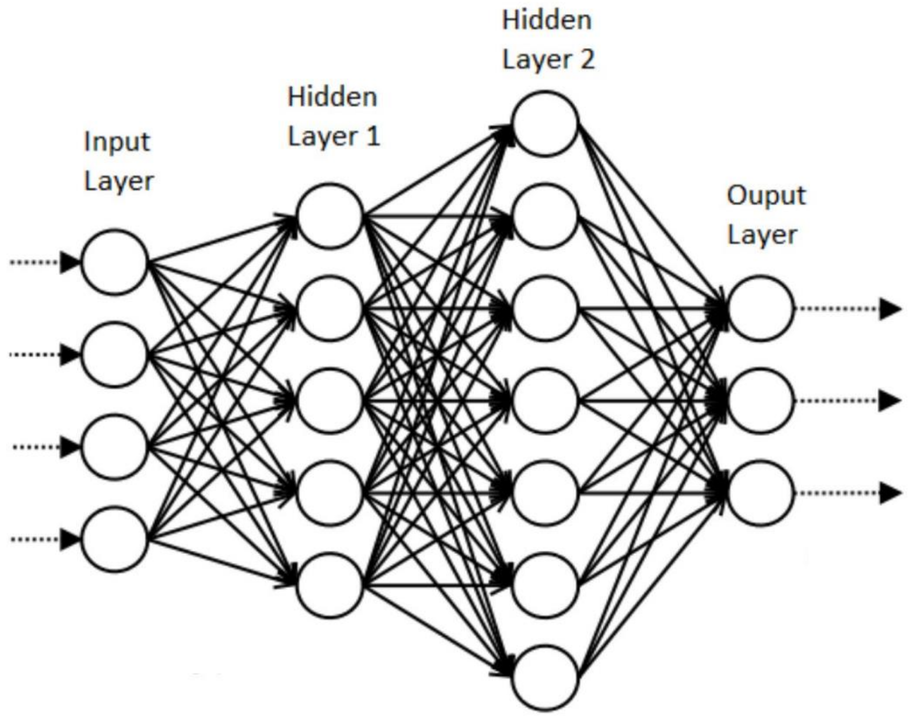


Figure 1-6 Sketch of multi-layered (deep) artificial neural network

When the neurons in ANN are connected to simply all the neurons in the next layer, the ANN is described as Dense Neural Network (DNN) or fully-connected neural network. Other ANN may have different types of layers that do more than conventional layers such as preprocessing or normalizing layer, convolution layer, or pooling layer. DNN and other types are used in Section 4.2 below.

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, the output is the sum of all those products plus a number called the *bias*. The bias can shift the resulting output value (along the y axis).

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

The output of each neuron goes through a function called *activation function* which can any of various types of *activation functions* as in Table 1-2 below. The *activation function* is picked depending on the used ML model. When the *activation function* used is non-linear then the ANN is non-linear.

Table 1-2 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)	$f(x) = \begin{cases} 0 & for\ x \leq 0 \\ x & for\ x < 0 \end{cases}$
Sigmoid	$f(x) = \dfrac{1}{1 + e^{-x}}$
Softmax	$f_i(x) = \dfrac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}\ for\ i = 1, \dots, J$
Hyperbolic Tangent (Tanh)	$tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

The Rectified Linear Unit (ReLU) function is used heavily in this work. 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 [28] with comparable results. Plus, it reduces the vanishing gradient problem [29]. The softmax activation function (normalized exponential function), which is used in this work as well, 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 (output layer) to limit the output between 0 and 1.

When an ANN is "learning," 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 gives correct results. When a final output of the ANN is given that is off target, 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 further 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 [30].

1.5 Summary

To summarize thus far, 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. These 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 platform for the ML models is a functional competitive serious game denoted WarCAT that has been developed and is instrumented to collect human player data.

WarCAT is based on the somewhat familiar card game WAR. Even with a game as simple as WarCAT, the potential contributions of ML in classification and detection of subtle cognitive change require considerable data. Thus, an RL model is developed to imitate and produce human-like data of mass quantity.

Chapter 2: Problem Definition & Analysis

In order to develop a software that can potentially detect MCI from gameplay data of WarCAT, a classification model must be built. There are two problems that needs to be tackled in order to get a working classification model with a useful prediction accuracy. Firstly, a model must be designed that can predict a level of confusion or impairment based on raw or processed gameplay data. This model, however, would need to train on a sizable amount of labelled data in order to have useful prediction accuracy percentage. This is especially difficult when the game at hand was designed recently and it does not have a large or any sizable player-base. This would lead to the second problem, we need a mechanism that can generate more data in order to be used for the classification model in attempt to explore the original conjecture that serious games can be useful in helping to assess one's performance across a range of cognitive functions. The generated data need to be similar to the human player data. This is a common mode for many ML techniques that rely on considerable data, in that, if the data does not yet exist, one needs to be able to synthesize a suitable amount of realistic data to help validate the original hypothesis that ML may be useful for subsequent data analysis in the first place. This also points to one of the difficulties of using serious games for the purposes of aiding in detecting anything - cognitive or otherwise. The game has to be extremely engaging to entice people to play, likely to the point of being addictive, which poses another potential ethical dilemma.

2.1 Designing a Classification Model

The model has to be able to read the WarCAT gameplay data as presented and classify it into one of two or more classes with differing MCI levels. If more than two classes are used, they can mean different levels of MCI severity. Since a positive diagnosis is rare in real data, we also need to compare sensitivity and specificity values unless balanced training data are used, meaning that 50% of the data are of one class and the other half is of the other class. Also, since winning a WarCAT game depends partially on the value of cards dealt, there is a randomness aspect to the data. Therefore, the classification model must be robust to such noise implicit in the WarCAT game.

2.2 Difficulties of Data Acquisition

Since the approach of designing such a classification model taken in this research is a Machine Learning model, it needs to be trained first on a sizable labelled dataset. Due to the lack of such a dataset, we then must design a method of generating data that has similar characteristics to real data. These data must satisfy several validity tests of characteristics such as error burstyness, score average, win percentage, high cross-correlation with real data, similar strategies (sacrifice 1st card, etc.), correlation with perfect play, etc. Issues such as burstyness was initially thought here to reflect moments of mental lapse or "loss of set". These types of subjective impairments in older persons are common, often expressed as "senior moments" manifesting in word-finding difficulties and momentary forgetfulness. The difficulty associated with data acquisition was compounded as there are very few studies related to correlating gameplay data of any sort of assessment aligned to cognitive difficulties.

Chapter 3: Methodology

3.1 Possible Data Processing Opportunities

Before the gameplay data enters the model, it may need to go through a preprocessing phase in order to present the model with only the data features that are important to the model and in a form that is readable by the model. This phase might include: removing features that have little to no contribution to the prediction process, altering features into a desired format such as normalization, and abridging multiple features into one feature, for example, taking the difference of two features instead of taking those two features directly, or taking the average value instead of many values. Determining the most appropriate features is a difficult task and largely more art than science especially in areas where ML techniques have not been widely applied.

3.2 Designing a Classification Model

The classification model was designed based on ML, specifically a deep (multi-layered) Artificial Neural Network (ANN). Several methods of deep ANNs that were explored in this research include Dense Neural Networks (DNN), Convolutional Neural Networks (CNN) and others. The ANN trains on an abundance of labelled data and is then validated on independent data until its prediction accuracy reaches its maximum. First, it will be trained on two classes which can then classify WarCAT player's gameplay data into either 'has MCI' or 'does not have MCI' characteristics (a binary classifier). Furthermore, it will be trained on five distinct classes which can then take in the processed data and provide a classification of the degree of MCI severity or degree of cognitive decline.

It is likely that the designed models require to be run with multiple values or types of hyperparameters. Many hyperparameters and factors affect the ML models such as learning rate, number of layers, number of neurons in each layer, drop rate, batch size, whether or not to use batch normalization, type of activation function used, etc. Selecting an untuned value can heavily affect the model performance as some hyperparameters have a high tunability [31].

Also, a non-ML classification model designed by simple statistical method is used in addition to the ML classification models. This will help keep track of comparison performances by the simple statistical model and the ML models. So that, we will know when an ML model is not necessary, as the simple model can do just as good. This is also common within ML research, where simple models are often overlooked, whereas they may in actuality be equally effective.

3.3 Designing Data Synthesis Mechanism

Synthetic data of a stream (*level*) of 50 or 150 games can be created with at least two different approaches. Figure 1-5 earlier shows an example of such a synthetic data stream. The two approaches used here means that the model simulates each game on either one of two different scales. First, we can design a high-level (naïve) model that simply assigns either a win or a loss to each game without simulating the inner details of a game. Second, we can design a low-level (granular) model that simulates each game fully. It simulates a player and an opponent, gives five cards to each, and gives an order of play or a strategy to each. Here, the high-level model is designed first and then the low-level model is designed pending whether the results from the high level were satisfactory or not.

It is common to use synthetic data to validate a model, however, for our purpose here, the goal of data synthesis is not only to validate the classification model but also to train it for functional use. It is worth noting that the role of synthetic data can diminish if more real labelled data are collected later on. It is imperative to note however, that the synthetic data generated here is being used to help validate the overall conjecture that ML and SG in combination can help in detecting MCI. Ultimately, the aim is to make WarCAT or similar games widely available and sufficiently engaging with the MCI ML detection models running on those data, ideally driven from a dataset of human users. This dataset can then be used to set baseline cognitive functions and help detect pre-symptomatic MCI using ML approaches on real data.

3.4 Tools

3.4.1 Software

The model was written in the Python Programing Language as it is widely used with ML models for its simplicity and the abundance of highly optimized numerical and ML libraries such as Numpy and Tensorflow. There are also libraries that can read data from either a SQL server or a Comma Separated Values (CSV) file format which are the two ways of receiving the WarCAT data.

3.4.2 Algorithms

For the classifier (prediction) model, the neural network was trained using the backpropagation training algorithm [30].

For the high-level data synthesis, Bernoulli trials (i.e. similar to coin flipping) and the Burst-Error model (Markov Decision Processes) were used to generate a 1 or 0 for a win or a loss. For the low-level data synthesis model, RL was used as it closely resembles the way humans learn [32]. Furthermore, learning in humans includes reward-based learning [33]. Similarly, the Q-Learning algorithm used here is based on reward-based learning.

Chapter 4 Developing Classification with Preliminary Data Synthesis

A preliminary foray into a solution/approach was undertaken to better understand the problem and potential solutions. In this chapter, I briefly present an initial attempt to synthesize data which are then used for the classification model. The data synthesis mechanisms in this chapter are simplistic and preliminary. However, they are explored more extensively in Chapter 5. The main focus in this chapter is developing the ML classifier model.

4.1 Data Synthesis

4.1.1 Introduction

A difficulty faced by most ML efforts in extracting meaningful inferences from gameplay data is data volume. Also, especially when applying ML in a medical field [34], lack of data is a common theme due to privacy concerns, and difficulty of unifying different forms of data. Therefore, tackling this problem of data shortage is inevitable, especially with data for this newly developed game.

A large amount of data are needed for the ML classification models. Such data would ideally be real humans' WarCAT gameplay data labelled according to the cognitive function or ability of the player. The data need to be in the thousands as is typical with ML models training. Also, more data are highly favorable to produce a trained classifier with better accuracy. This is particularly the case here, although MCI becomes more prevalent with ageing, the symptoms are "buried in the noise" of normal cognitive decline. The subtler the symptoms, the more data are required to detect the symptoms. Another reason for the need of large amounts of data is that the data collected by humans will likely, in future work, need to be controlled for age, gender, and the many confounding variables that impact one's day to day (even hour to hour) abilities to concentrate. Due to the lack of such data from human play in sufficient quantities, the need for synthetic data arises to help verify or establish the classification model. The generation of synthetic data is common in scenarios where real data are scarce or difficult to obtain but are required to validate ML classifiers [35]. Towards this end, in this work, there is heavy reliance on generating synthetic data to help verify or establish the validity and utility of ML classification techniques.

The data synthesis efforts in this chapter explore high-level (naïve) simulation. In such simulation, each game is assigned a value of only the final result of the game. In the subsequent chapter, more detailed simulation is explored that simulates a more detailed play by simulating the cards play in each game. This naïve approach was initially undertaken as being simple and hypothesized as having the potential to, at minimum, classify more obvious cases of cognitive difficulty.

To make the synthesized data consumable by the ML classifier, consider the general form of the data expected by the ML classifier. The ML classifier require gameplay data of at least two different classes. Typically, they are differing by performance. One class of gameplay is of higher performance than the other. In the subsequent sub-sections, different ways of making two or more classes are explored. In the naïve case, a single game is represented by (reduced to) a single variable. This variable may refer to a Win vs. a Loss or may be referred to Following a strategy vs. Not Following a strategy. The term Following a strategy refers to the case where the player plays correctly with the right strategy yet they might lose the game due to ‘unlucky’ cards. That is, they might lose the game but not due to bad performance or they may win the game with a relatively bad performance. If the variable (game result) refers to Following a strategy or not, then this reduces some of the noise, making it easier for the ML classifier. However, if this approach is implemented, then further work needs to be done to determine the correct strategy of play and whether the player followed it or not.

One way of differentiating two classes is by the winning percentage, or the Following the strategy percentage. That can be computed from the “cognitive fingerprint” stream as in Figure 1-5 above. Another way of differentiating two classes is by averaging the scores of the games in the stream. In this case, the scores are what is calculated according to equation (1-1) above.
Next, different ways of generating synthetic data are explored.

4.1.2 Bernoulli (Simple Random Coinflip-analog)

In this simple model, the two classes can be differentiated by the value of the winning percentage. One class can refer to random play which yields a 50% winning percentage. The other class that represent a better performance can have a winning percentage higher than 50%. Since the game is stochastic, it is unreasonable to have a 100% winning percentage. So, a simple random bit generator of 0.5 probability of 1 and 0.5 probability of 0 similar to a random coin flip trial would generate one class. The other class can be generated by simply assigning a skewed probability of, for example, 0.7 probability for 1 and 0.3 probability for 0. Here 1 represent Win and 0 represent Loss.

$$Class1\ (Random\ Play)\ \begin{cases} P(1) = 0.5 \\ P(0) = 0.5 \end{cases}$$

$$Class2\ (Better\ Play)\ \begin{cases} P(1) = 0.7 \\ P(0) = 0.3 \end{cases}$$

4.1.3 Burst-Error

As a reasonable first extension to the Bernoulli generating gameplay approach, a bursty error model is introduced in detail here. This model was inspired by early models used for error correction associated with transmitting data over noisy channels. In these cases, the disturbance in the channel extends over many bits or symbols emulating moments of memory lapse or forgetfulness.

The premise here is that a person may undergo a degree of confusion that is dependent upon previous hands played. When a person forgets the bot’s strategy, it takes a few games to re-learn the bot’s strategy and play in a manner to counter it. Similarly, if a person plays what they believe to be an on-average winning strategy, but loses by the stochastic nature of the game, they may become confused momentarily and struggle to return to the previous strategy. Thus, these several games would show in the “cognitive fingerprint” stream as burst of losses (or bad performance). Hence, the motivation for the use of the burst model to mimic bursty periods of confusion or memory lapse, as a form of cognitive impairment. The error model is based on independent Bernoulli trials, and is similar to that produced by flipping a coin but with varying probabilities. In this case, a head would represent playing the correct strategy whereas a tail would be a random strategy. Then, after the strategy has been decided, another independent Bernoulli trial is done similar to flipping a coin with varying probabilities. In this second case, a head would represent a win whereas a tail would represent a loss. So, the first coin determines which strategy to play with, and the second coin determines the win or loss. However, when looking at the rates of wins and losses, they are not necessarily independent. As a first attempt to more realistically model play and degrees of impairment, a simple Gilbert-Elliott two state Markov process [36] for generating errors is considered. A diagram is shown below.

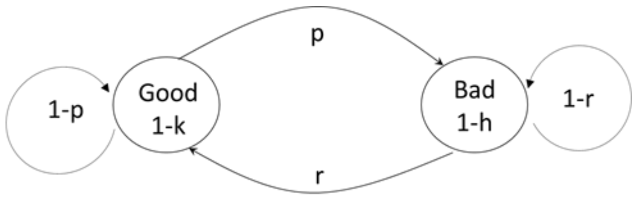


Figure 4-1 Gilbert-Elliott model generating a 2-state Markov modulated error process

The probability of winning in the Good state which represents playing with correct strategy is k and the probability of winning in the Bad state which represents playing with random strategy is h. Similarly, errors are generated at rates of 1-k while in the Good and 1-h while in the Bad state. Furthermore, p is the probability of moving from the Good state into the Bad state, and r is the probability of moving from the Bad state into the Good state. The steady state error is given by:

$$p_e = (1-k)\frac{r}{r+p} + (1-h)\frac{p}{r+p} \tag{4-1}$$

When the probability of leaving a state is less than 0.5, that state tends to be more likely to be followed by a play of similar state aka similar play strategy, thus leading to a burst of that state. An illustration of the burstyness of the error patterns generated is shown below:

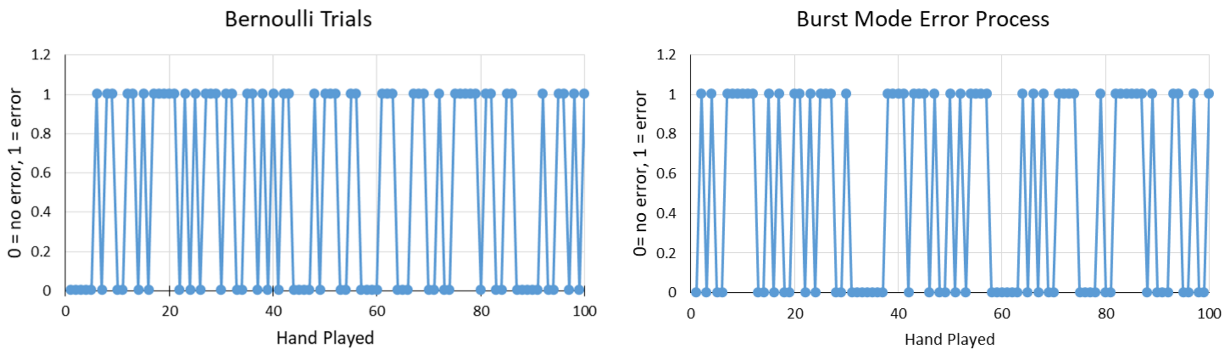


Figure 4-2 Burstyness of the Bernoulli and Burst Mode modulated error processes (p=0.25, r=0.25, k=0.9, h=0.1)

In Figure 4-2 above, the Bernoulli trials were generated using a fair coin with p=0.5, The Burst Mode "loss of set" or error in play was generated with parameters p=0.25, r=0.25, k=0.9, and h=0.1. The number of errors generated was 51 and 50 out of 100 for the Burst Mode and Bernoulli trials error generating modes of play respectively. The number of games is used here is increased from 50 to 100 in order to slightly improve the data. However, the work here holds if the level or the stream of games is 50, 100, or 150. It's worth noting that although both these two methods of generating data produced data with similar number of errors, one method had the errors in bursts while the other method had the errors in random independent positions. This gives options to classifiers to pick up on features other than the average number of wins.

4.2 The Classification Models

The main goal here is to develop a way of classifying gameplay data to the class of mechanism that generated them. Initially, we explore classifying two classes. Then as a next step, we can explore classifying multiple classes within a range of performance. Performance can refer to the winning percentage, score average, or others. Multiple classes with gradual differing level of performance can be a gateway to having a model that can predict the level of confusion i.e. how divergent play is from perfect play. The approach of trying different classification models is to first look at simple statistical model first, then consider the ML models. Also, with the ML models, simpler ML models are explored. Models such as Dense Neural Networks (DNN) are followed by exploration of Convolutional Neural Networks (CNN), which can more easily make classifications based on more features.

4.2.1 Simple basic model (statistical)

The simple basic model is a non-ML classification method. If this attempt turns out to be unfruitful, then it can serve as a reference for the subsequent ML models. The prediction here is based on the winning percentage. Simply, if the winning percentage is above a threshold then it would be classified as one class, otherwise it would be classified as the other class. One of the two classes taken here is random play which yields a winning percentage of 0.5. The other class represent "smart" play with a winning percentage of above 0.5. That is winning percentage throughout a stream of 100 games. The number of games is used here is increased from 50 to 100 in order to slightly improve the data. However, the work here holds if the level or the stream of games is 50, 100, or 150.

These two classes can be generated using a random number generator of 1 or 0 (Bernoulli Trials) with a given probability as in 4.1.2 above. For example, let P(class1) = 0.5 and P(class2) = 0.7 then we can set the cutoff threshold at 0.6 being the middle. Then the prediction model:

$$\text{Class prediction } = \begin{cases} class1 & \mathrm{W/(W+L)\ ratio} < 0.6 \\ class2 & \mathrm{W/(W+L)\ ratio} > 0.6 \end{cases} \tag{4-2}$$

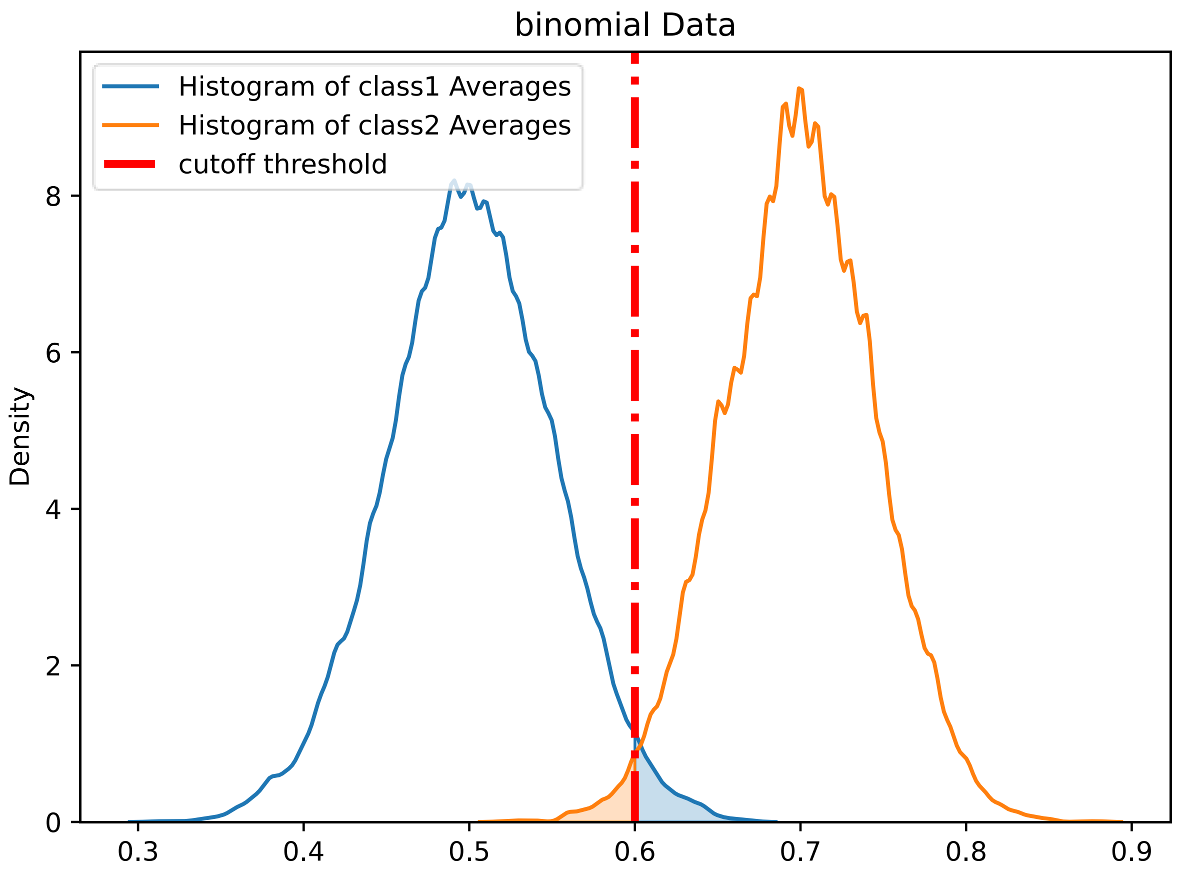


Figure 4-3 Histogram of two classes with winning averages of 0.5 and 0.7 (n=10000 for each class)

The prediction accuracy depends on a several parameters used in the data generation including:

- How far apart the Bernoulli probabilities of the generated two classes are from each other. In the example in Figure 4-3 above, they were 0.5 & 0.7. This yielded an accuracy of 98% but it drops to 84% when P(class2) = 0.6 and the cutoff is 0.55 as shown in Figure 4-4 below.

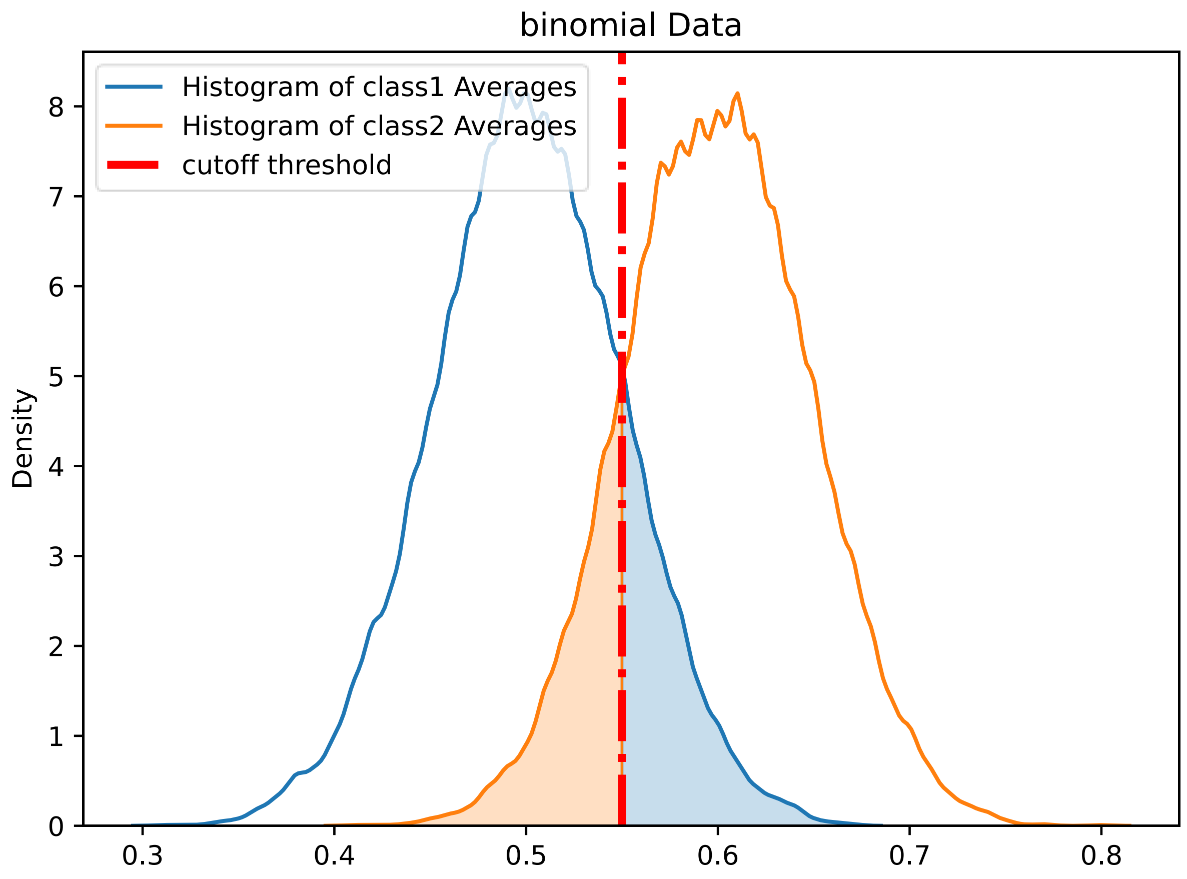


Figure 4-4 Histogram of two classes with winning averages of 0.5 and 0.6 (n=10000 for each class)

- The number of games N whose winning percentage is the input. In the example above, it was 100. In Figure 4-3 above, the accuracy is 98% but it drops to 93% when N = 50.

The class representing better play is given a probability of winning below 1 because the player may lose some of the games due to the nature of the game WarCAT. The cards are dealt randomly which may sometimes lead to the player being dealt 'bad' (unlucky) cards that result in a loss even if the player played well.

It is worth noting that the results of this simple model are somewhat inconclusive. That is because the accuracy depends on how far apart the winning percentages of the two classes are. At present, it is not clear how far apart to set them to credibly emulate a real gameplay.

As for classifying the data that came from the burst-error, the model yielded similar results. This is not surprising because this classification model has one feature which is the winning percentage. Thus, the burstyness feature is ignored by this model. This is a serious flaw in this simple model as it doesn't distinguish between Bernoulli trials data and bursty-error data so long they have the same winning percentage as shown in Figure 4-5 below.

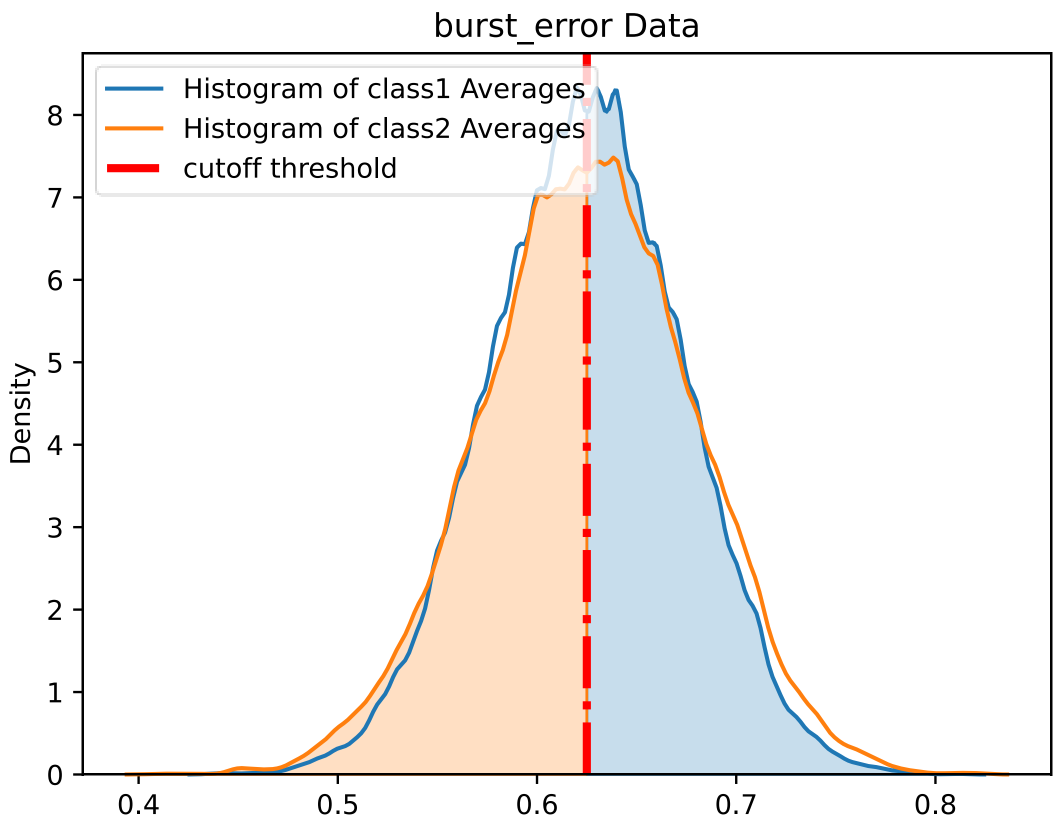


Figure 4-5 Two classes with similar winning percentages but differing bursts

In Figure 4-5, the two classes are only differing by the values of p and r in Figure 4-1 above. Class1 has 0.5 for both p and r whereas Class2 has 0.2 for both. The winning percentage the same for both classes. The resulting accuracy is 50.20% which is no better than randomly guessing. This motivates the exploration of ML models' abilities to extract patterns from the data that may be less obvious.

4.2.2 DNN

Neural Networks have proven to be quite successful classification models in various fields and applications. They have been used in areas such as image classification and object recognition [27], recognizing hand writing [37], audio recognition [38], [39], text translation [40], object tracking in video [41], etc. Multi-Layered Neural Networks can even do a good job at detecting hierarchical levels of features [42]. However, the area where Neural Networks have shown the most solid progress with reference to human performance is in playing games [43]. Typically, Neural Networks work well in applications where the problem inputs are either in numerical representations such as image pixels and sound wave amplitudes or can be imbedded in numerical representations such as gameplay, or text representation.

A Dense Neural Network (DNN) model used here is roughly based on the MNIST handwritten digit classification model [44]. DNN is chosen as an initial attempt for an ML classifier because it is one of the simplest neural network models. It uses a fully connected Neural Network made of three layers with numbers of neurons 150-500-2. The first layer represents the input layer which takes in the stream of 150 numbers representing a win or loss. The middle layer is made of 500 neurons each is connected to all the 150 neurons in the first layer. Each neuron in the middle layer can extract a feature. The third layer or the output layer has two neurons representing the number of classes (better play strategy or random strategy).

ML models have an inherent problem that need to be remedied which is overfitting. When an ML model trains on data, its performance gradually improves. At the beginning, it starts learning the general features in the data. Then, when the model trains for too long, it starts to memorize the noise specific to that dataset. However, when the model attempts to predict new data with different noise, it performs less than optimal. One of the widely used techniques to solve overfitting is cross-validation [45]. Two datasets are used in the process that typically come from splitting a parent dataset. One is typically 8-10x larger than the other. The large one is used for training the ML model while the small one is used to evaluate or validate the model performance. The validation dataset is not used in the training thus can serve as an objective evaluator.

In this DNN model, two classes are used which both have the same winning percentage but differing in the burstyness of play. Two datasets were generated per Section 4.1.3 Burst-Error above. The training dataset is made of 100k games while the validation dataset is made of 10k games. Each level is made of 50 or 150 games of play per the corresponding class strategy.

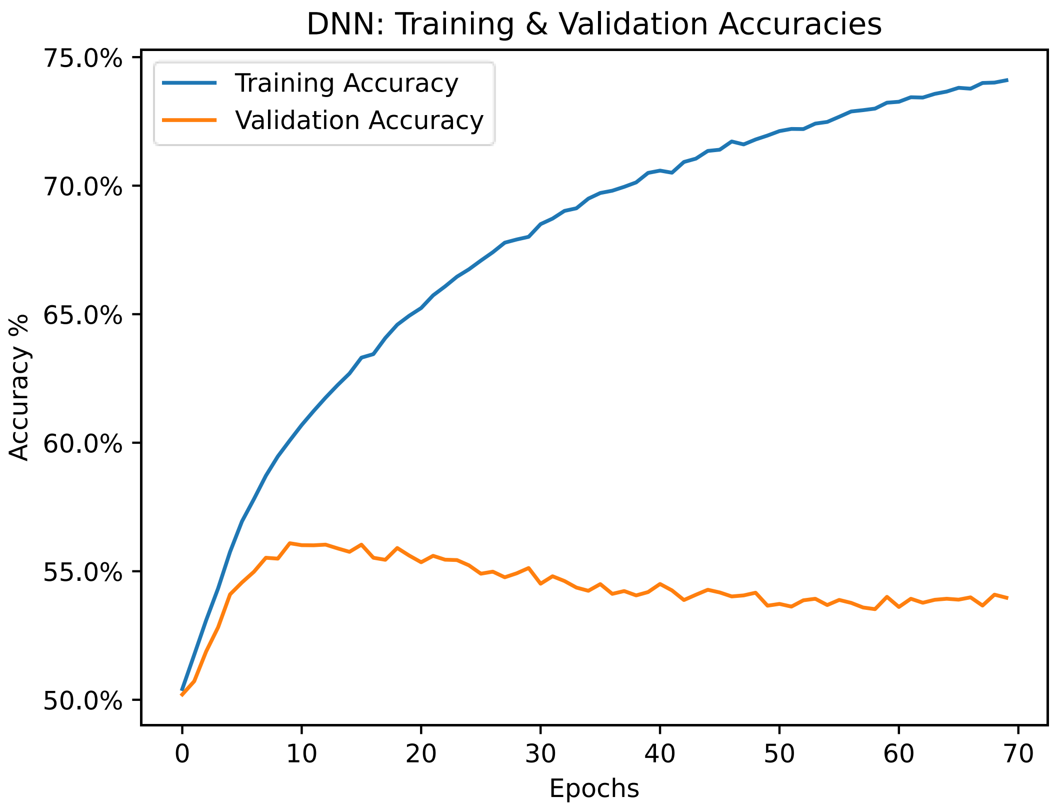


Figure 4-6 DNN training and validation accuracies.

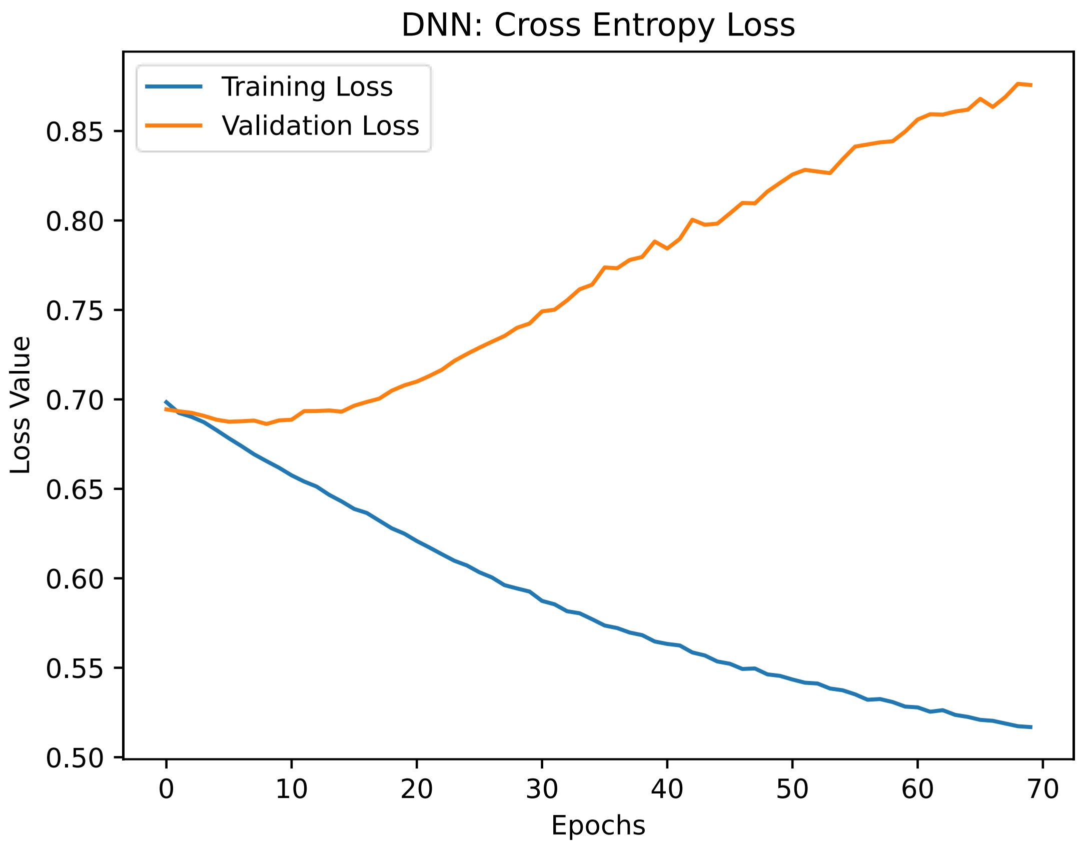


Figure 4-7 The Training and Validation Cross-Entropy Losses

In the two figures above, the two classes only differ by the values of p and r in Figure 4-1 above. P is the probability of moving from Good state into the Bad state and r is vice versa. Class1 has 0.5 for both p and r whereas Class2 has 0.2 for both. The winning percentage is the same for both classes. The resulting validation accuracy is 55.84% which is slightly better than statistical model but still impractical. However, when Class2 has 0.1 for both p and r, the resulting validation accuracy rises to 62.27%. It is worth noting that both classes have the same winning percentage. Although, the accuracy is still low, there is about a 5 to 12 % rise in the validation accuracy due to the use of ML. The magnitude of this rise depends on the amount of burstyness in the data; in other words, the magnitude of this rise depends on the values of p and r used. This ML model has achieved this based solely on the burstyness. Therefore, ML model is preferable to the Simple statistical model for this high level classification.

The two classes represent a person with MCI and a healthy person or one with normal age-related cognitive decline. As they are not likely to get the same winning percentage, the eventual accuracy is likely to get higher than 62.27%. In fact, the validation accuracy is 87.01% when the two classes have winning percentages of 70% and 80%. The 80% winning percentage representing normal healthy play which is based on the average winning percentage of real data from about 30 undergrad and graduate students who helped with assessing the game and its aesthetic and feel. The 70% winning percentage would most probably represent play with some degree of cognitive difficulty. However, there were no real data available from people with MCI at the time of this work. The data collected was not part of a clinical trial, but rather an early effort to assess the game's potential to be engaging.

4.2.3 DNN with FFT Preprocessing

Another approach of model design was attempted. A Fast Fourier Transform (FFT) was implemented over the raw input to get a representation of the different frequencies associated with the cognitive fingerprint matrix. The motivation for this approach is that the different burstinesses in the streams of games would manifest as different frequencies in the streams. The result is shown in Figure 4-8 below, with slight improvement to 58.73% validation accuracy compared to the 55.84% in bare DNN, and 50.20% for the statistical model. Furthermore, the validation accuracy is 87.27% when the two classes have winning percentages of 70% and 80%.

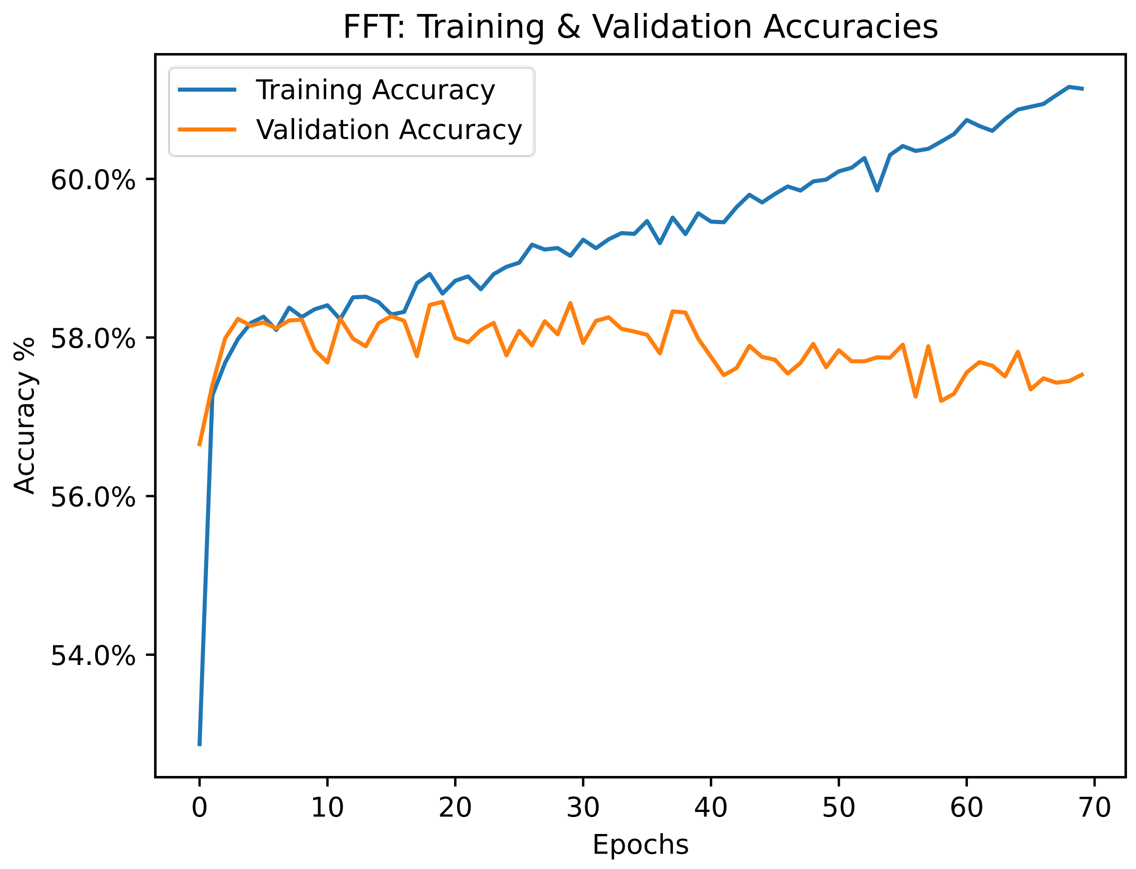


Figure 4-8 Training and Validation Accuracies of DNN using FFT preprocessing

4.2.4 CNN

Although the promise of ML techniques for classification is enormous, there is onus on the implementer to process the data in such a manner that the classifier can more easily accommodate. One type of classifier that has been identified for use here is a convolutional neural network (CNN) [27]. CNNs are traditionally used for image classification which typically are of 2-dimensional data but they also have corresponding 1-dimensional variants [46]. An important aspect that needs to be captured from the gameplay stream (cognitive fingerprint) of data is when the player scores poorly (forgets the strategy and needs to relearn the strategy again) aka the burstyness. Thus, a CNN model can capture how often that happens and at what frequency (for example: every ten games or twenty games) similar to the FFT. Additionally, the CNN model can encode the position of these bursts relative to each other which FFT does not. CNN model encodes these features though taking the convolution of the data stream. Different filters are convolved with the data. A filter is convolved by multiplying the filter with the input. Then the filter is shifted one step and multiplied by the input again and so forth as shown in Figure 4-9 below.

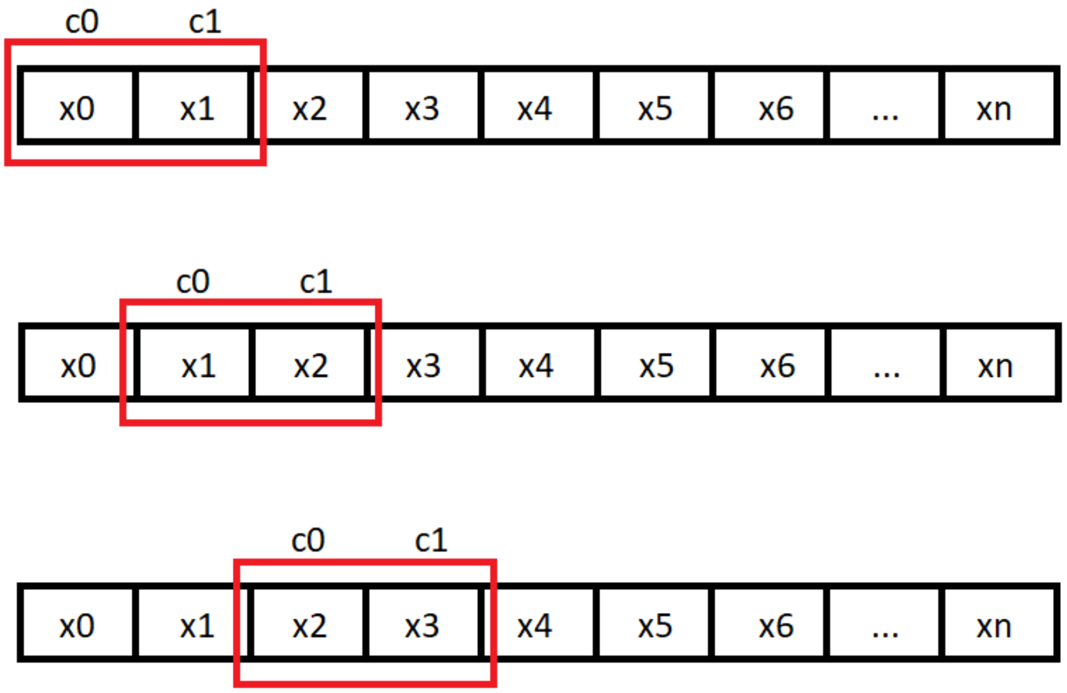


Figure 4-9 An Example Diagram of Convolving a Filter with a 1-Dimensional Data

In the figure above, c0 and c1 are the values of the filter. At each multiplication instance of filter being multiplied by an input, the output is calculated per (4-3) below.

$$convolution\ output\ of\ step_n = (\mathrm{c}_n * x_n) + (\mathrm{c}_{n+1} * x_{n+1}) \tag{4-3}$$

The input data of Convolutional Neural Network (CNN) were similar to that of DNN. The model is designed as in the figure below.

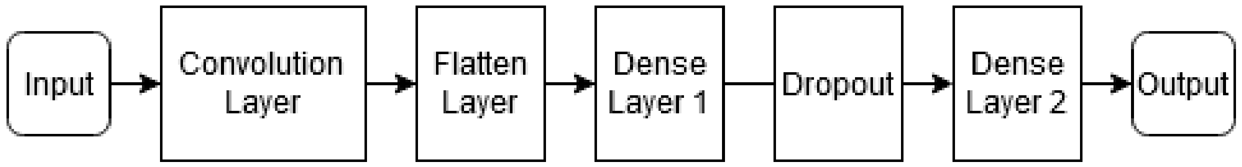


Figure 4-10 CNN Model Layers Diagram

The input is 150 neurons. The convolution layer uses 32 filters with a kernel size of 8. Dense Layer 1 has 200 neurons. The dropout rate used is 0.25. Dense Layer 2 has 2 neurons which are the classifier outputs. The dropout is one of the simpler techniques used to counter overfitting. In this technique, a randomly selected small number of neurons are ignored during training. In each training step, different neurons are randomly selected to be ignored. This leads to a robust model where it is not highly dependent on certain neurons. It is worth noting that the dropout is only used in training. It is not used when calculating the validation accuracy or when the model is used in deployment.

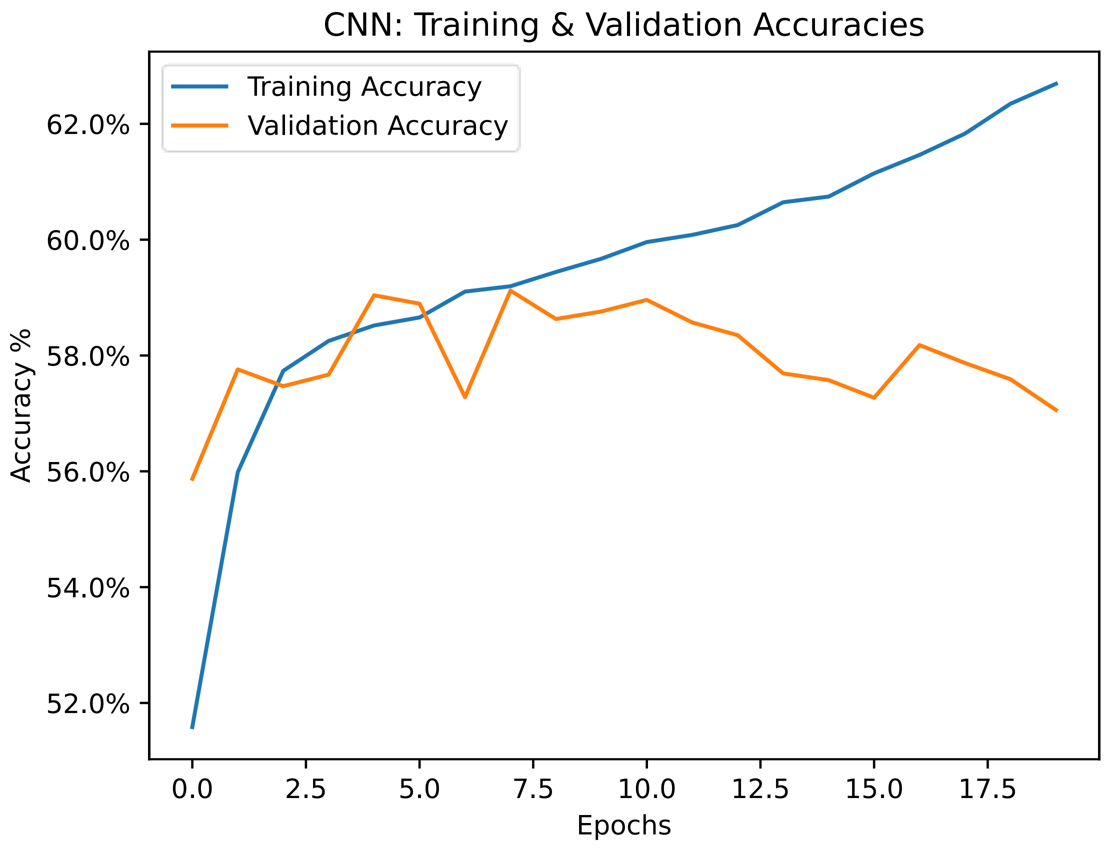


Figure 4-11 The Training and Validation Accuracies for CNN Model

In the figure above, the best validation accuracy is 59.11% which is just slightly better than the 58.73% of the DNN with FFT preprocessing. Furthermore, the validation accuracy is 87.33% when the two classes have winning percentages of 70% and 80%.

On the other hand, another possible approach is worth highlighting. Now, when the player is in the Good state, the player has a higher probability of winning but that probability is not certain (not 1). Thus, by looking at the data, seeing that a player had won does not mean that the player is in the Good state. This adds a noise to the data. Assume, there is an intermediary system that labels a play of Win/Loss into Following/Not-Following a strategy. Then the resulting data would have

1 for Following a strategy and 0 for Not-Following a strategy. In this case, both neural network models DNN and CNN yielded accuracies higher than 99% when the data are labelled as either Following or Not-Following a strategy. However, these models require an intermediary model that can pre-process/classify the gameplay data of Win/Loss into Following/Not-Following a strategy. Such an intermediary model Figure 4-12 was proposed in a separate work [47] which checks whether the player's strategy resembles any of the statistically most likely winning strategies.

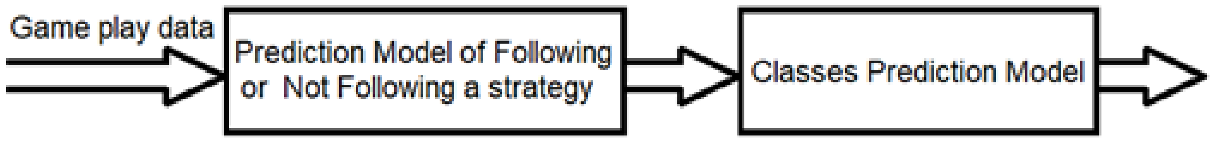


Figure 4-12 Inference of following or no following a successful strategy

4.3 Summary

In this chapter, first preliminary ways of synthesizing data were explored. The main purpose was to provide input so we can explore the classification models in depth, particularly the ML models. However, ML models did not yield conclusive classification accuracies because the accuracy depends on how far apart the winning percentages of the two classes are. The accuracy is around 87% when the winning percentages of the two classes are 80% and 70%. However, in the next chapter, the work will focus mainly on improving the synthetic data by focusing on exploring low level simulations and data preprocessing.

Chapter 5: Advanced Data Synthesis & Classification

This chapter explores data synthesis in a granular manner as opposed to the naïve approach in 4.1. Then, those synthesized data are used to train the classifier models displayed previously in 4.2. The main work in this chapter is about the data synthesis.

5.1 Motivation and Data Analysis

The data are in the form of a stream of 150 games which constitute a level. This is referred to earlier in this work as a \`Cognitive Fingerprint\` as shown in Figure 1-5 and Figure 4-2 above. Each single game in the stream can be represented by a one bit variable as in the naïve approach in 4.1. However, in this chapter, each game is simulated in a more detailed manner. In this approach, a game can be represented by a more meaningful number. For instance, a game can be represented by a score that gauges the level of performance in that one game. In another instance, a game can be represented by a 1 or 0 where the player might be assigned 1 even if they lost the game if they displayed good performance relative to the cards dealt. Another advantage of this granular simulation is that the game implicitly captures the exact amount of noise coming from the fact that this game, WarCAT, involves cards being dealt randomly. The amount and frequency of this noise might affect the burstyness conjectured in the previous chapter. Thus, here, this and possibly other overlooked dynamic factors are being taken into consideration with more granular simulation. Also, this will reduce the amount of surprises when the model connects with real data in any future deployment.

5.1.1 Gameplay Analysis

In the game WarCAT, the player can demonstrate their level of performance by playing with the goal of maximizing the score or the score average of the stream of games. There is more than one way of playing the game that would lead to, on average, successfully countering the opponent's strategy. These ways can be thought of as of either one of two types: fixed strategies and adaptive strategies. In the fixed strategy, the cards are played with regard to opponent's strategy alone and without regard to the opponent's card at that specific game. An example of a fixed strategy is playing by sacrificing one card or two cards as discussed before in 1.3.1. In the adaptive strategy, the player might modify their play based on what information they glean about the opponent's cards. For instance, the player might decide to sacrifice two cards if the opponent's first card less/more than a certain value. Another example of an adaptive play is when a trained artificial model or agent is doing the play. Such model or agent can be trained where the actions are taken with regard to the opponent's play. There are many algorithms that train models in this manner. Each way of playing gives different performances or score averages ranging from the totally random to the maximum performance that manner of playing can give. As such, there are many ways of playing giving similar score averages. Given that the synthetic data are a substitute to real data, the synthetic data need to be similar to real human play not just in the score average but also in capturing the same underpinnings of play.

5.2 Models for In-Depth Gameplay Synthesis

Multiple models of generating granular gameplay data are explored next. The simpler models are explored first to establish a reference, then more advanced models that utilize ML are explored. The ideal goal is having a model that is capable of playing similarly if not better than the human performance. To gauge human performance, a small tournament was held where a group of ~30 undergraduate and graduate students played the WarCAT game on mobile phones. The person who won the tournament achieved a score average of 18.5 and a winning percentage of 84.6%. This person had played six games with a score average of 18.3 and a winning percentage of 83.8%. However, the score average of all 30 students is 15.69 and the winning percentage is 80.56%.

5.2.1 Simple Winning Strategies

Since the computer always plays with the same strategy throughout the 150 *games*, we can find a counter play strategy throughout those *games*. 150 games are used here instead of 50 games to improve the data (lower the noise). Also, 150 can reflect a gameplay of 3 levels of 50 games where a longer-term goal would have been for the computer to alter their strategy from level to level. However, currently the computer plays with the same strategy throughout the 150 games. For example, let the computer strategy be playing its cards in a descending order high to low. Then, one counter strategy would be play the lowest card first, then playing the four remaining cards high to low as shown in Figure 5-1 below.

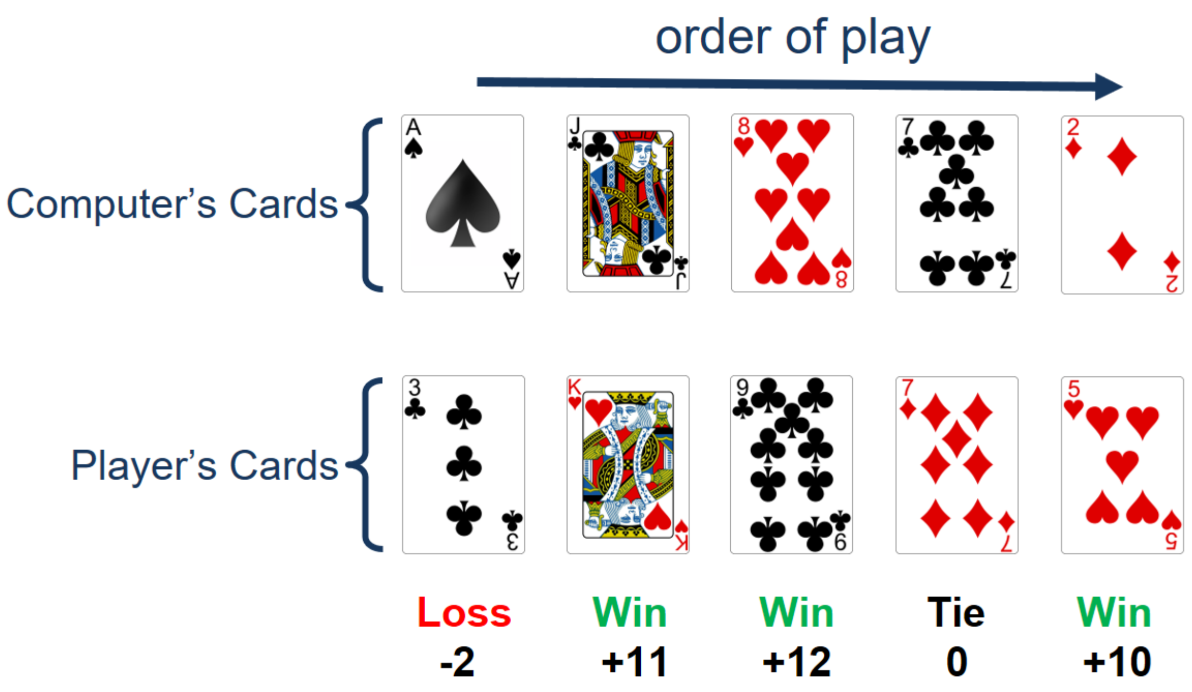


Figure 5-1 an example of computer's strategy and a player's counter strategy

The *rounds* are played consecutively left to right. After playing all five *rounds*, the player or the computer with the highest aggregate points wins the *game*. The points are awarded per equation (1-1) above. In fact, this counter strategy of playing the lowest card first gives a score average of 14.71 points and a winning percentage of 77.1%. Another winning strategy is sacrificing the lowest two cards first, then playing the rest high to low. This second strategy gives a score average of 9.19 and winning percentage of 85.74%. These score averages are below the top human level of 18.5. These different styles of play can be considered different classes to train the ML classifier in the later sub-part of this chapter. They can even be mixed with the probability of random play to produce a similar type of play but with lower score average or winning percentage.

The second strategy is safer with winning percentage but yields less score average than the first strategy. However, these two strategies can be used either one or the other depending on the cards dealt per game. For example, here is an adaptive strategy that uses either one of both depending on the situation. Make the first strategy the default and thus playing the lowest card first. However, if opponent's first card indicates a risky situation, then switch to the second strategy and thus play the second lowest card. Risky situation means when the opponent happens to be dealt high cards and the player happens to be dealt low cards. An example of detecting that would be if the opponent's first played card (opponent's highest card) is an Ace and the player's highest card is less than an Ace. In fact, this laid out example yields a score average of 15.72 and winning percentage of 80.62%.

5.2.2 Expected-Value-Based Play

Here is a purely statistical model that calculates the expected card value from all the possible cards in the deck. For the first card, the expected value is 12.3 if the opponent is playing with strategy of playing high to low. This is the expected value of the maximum of five cards. For the other cards, the expected values are shown in the table below.

Table 5-1 The Expected Value of

Card Number	Statistical Expected Value (Not Adaptive)	Average Number of Combinations
1	12.30	371,280 exactly
2	10.17	~81,126
3	8.00	~14,134
4	5.83	~2,345
5	3.70	N/A

These expected values are calculated empirically from checking all different combinations of 5 cards where each card can have a value of 2 up to 14 (Ace). There are 371,280 different combinations. Each one of these combinations is made of 5 cards that are sorted. When it comes to playing the first card, the expected value is calculated out of those 371,280 combinations. Then when it is the second card's turn, at this point the opponent's first card is known. Then, out of those 371,280 combinations, only a portion of them have the first card that matches the opponent's first card. Then, the same for third and fourth cards. Then, the fifth card is played as it is the only one left.

Since each play move is calculated on the spot, the model does not require training. However, a drawback of this approach is large amount of calculations required for each move.

However, the achieved expected score with this approach is ~12.67. This is not better than what was achieved with simply following a constant strategy and not better than the human level of ~18.5. Perhaps, the poor performance is a result of this model looking only one step ahead. Only one card is considered when deciding the move. This strategy is to maximize the instantaneous current card score without regard for a larger strategy that maximizes the final 5-cards score.

5.2.3 Reinforcement Learning (RL)

The adaptive strategy in 5.2.1 above outperformed the fixed strategy because it can have two options to deal with a situation. A trained model or agent can learn even greater options to deal with a situation with even more nuance. Therefore, such model will, in general, have the potential to achieve better performance.

Furthermore, one means to establish the credibility of ML for classification is to generate a dataset of synthetic data that would have similar characteristics as a data derived from a real person playing the game. With this objective in mind, techniques to generate synthetic data focused on

Reinforcement Learning were explored. The basic idea was not to create synthetic player bots with artificial intelligence superior to human performance, but rather bots that exhibited artificial but intelligent behavior. This was demonstrated through training player bots with a range of different numbers of training patterns. Those trained with fewer patterns ostensibly would represent players with cognitive decline. Those trained with a large number of training patterns represent players with no cognitive difficulty. The basic idea is to consider RL as a learning means similar to human learning. In general, the more training, the better the RL bot's performance. Analogously the less training, the lower the RL bot's performance. Specifically, here we consider the well-trained bot as not displaying any form of cognitive decline whereas the lesser trained bot as consequently suffering some form of cognitive difficulty.

Generating synthetic data to emulate cognitive difficulties or decline is ideally suited to the ML technique of RL. Of all ML strategies, RL is the one that most closely emulates the means in which humans learn many tasks. In general, a stimulus is presented, an action taken, and a reward provided as illustrated in Figure 5-2 below.

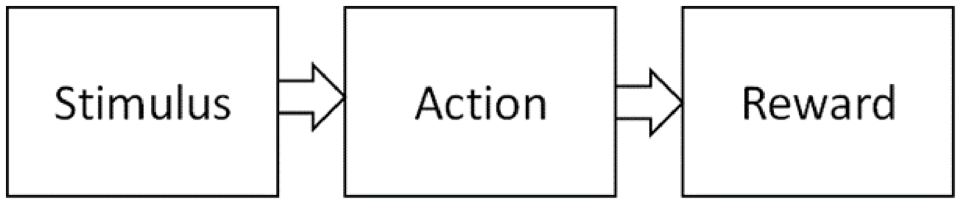


Figure 5-2 Stimulus-Action-Reward Paradigm

In the case of WarCAT, the stimuli are the cards played by the computer (game bot) and the action is the card played by the player (player bot). The objective is then to build player bots to emulate players with various levels of cognitive decline. Although RL has a long history and various mathematical manifestations, the principles are simple. Player bots learn through repeated play where positive rewards reinforce actions that emit further positive rewards, unless there is a tie, in which case the score for that play is 0. A positive reward is associated with winning, while a negative reward or score with losing. With this scoring in place, it is easier to learn a winning strategy through reinforcement if play undertaken wins more often.

The approach takes its lead from AI techniques being applied to game agents or bots [48]. In the case of AI agents in games, typically the objective is not to outsmart players. Rather, the objective is to have the agents be as smart as necessary to maintain the human player's engagement and sense of challenge and fun. The objective is not to push the limits of ML within the bot, as would be the case in many ML optimization instances. For our purposes, the ML bot needs to be imperfect (and further, displays varying degrees of imperfection), thereby, imitating a human-like behavior. More specifically the objective is for the bots to display artificial behavior as opposed to superhuman-like superior artificial intelligence. With this objective, the goal is to train bots for various periods of time or for various number of training patterns or iterations. Bots trained with minimal iterations will essentially have little knowledge of what to do, while those that have been trained longer will have learned strategies that enable them to win consistently. As such, bots with various levels of training will represent human play with various levels of impairment or cognitive decline.

5.2.3.1 Q-Learning

The size of all possible stimuli (choosing 5 cards out of the deck) for the first card played is all the combinations with repetition allowed, but order is not important. Its notation is as follows:

$$Combinations\ (repetition\ allowed, order\ notimportant) = {}_{n+r-1}C_r \qquad \text{(5-1)}$$

n = the number of different cards, r = the number of selected cards.

$$Card1\ States\ Size\ (n = 13, r = 5) = {}_{13+5-1}C_5 = 6{,}188$$

For the second card, we take all the possible combinations as in (5-1) with *r* = 4. Then, since the opponent's last card is considered as well, Card2 state size is multiplied by the different cards in the deck. Also, since only four cards are available of each nominal card in the deck, the states that are made of five similar cards are not possible. Therefore, 13 is subtracted out of each state size of the five cards. The state sizes of all cards are shown in Table 5-2 below. Card5 does not require a model as there is only one card left to play.

Table 5-2 State Size for Each Card

Card Played	**State Size**
Card1	6,175
Card2	~23,627
Card3	~5,902
Card4	~1,170

The numbers are relatively small, not requiring a deep network. Therefore, the tabular Q-learning algorithm is a suitable choice for the AI bot. However, the actions size is not constant as it is 5 for the first card, then decreases by 1 every time a card is played. Figure 5-3 below illustrate a visual map of the states and actions for each of the cards played.

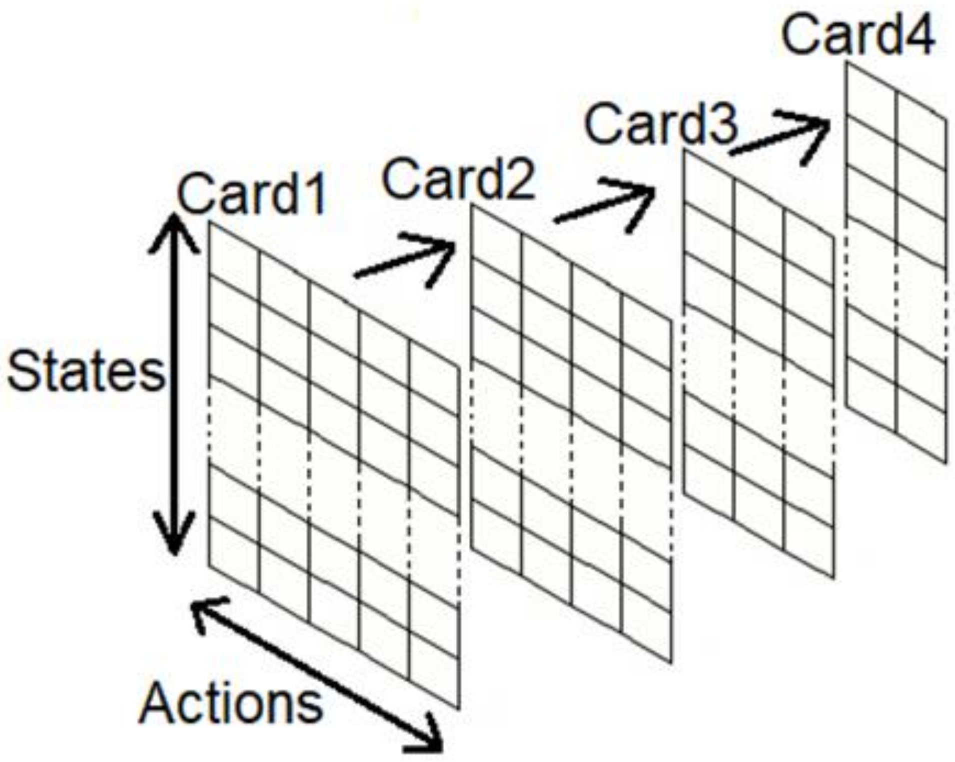


Figure 5-3 Illustration of Actions and States as a Game Progresses

There are four different Q-tables for each of the cards played. The Q value updates according to (5-2), (5-3) where the Q value of table n+1 is used to update the Q value of table n.

$$\underbrace{Q^{new}}_{Table\ Card\ n} \leftarrow \underbrace{Q^{old}}_{Table\ Card\ n} + \underbrace{\alpha}_{Learning\ Rate} .(learned\ value) \tag{5-2}$$

$$learned\ value = \underbrace{r}_{reward} + \underbrace{\gamma}_{discount\ factor} .max \underbrace{\underbrace{Q}_{Table\ Card\ n+1}}_{\substack{estimate\ of\ optimal \\ future\ value}} \tag{5-3}$$

The r (reward) in (5-3) corresponds to the score of the card played which is calculated according to (1-1) in Section 1.3.1 above. The Q value of card-4-table gets updated where the *learned value* in (5-2) is the final aggregate score of that game according to (1-2) in Section 1.3.1 above.

To play with a more advanced level of play, it is important to make the process of selecting an action to also consider the previous cards played by the opponent. So, the opponent's last played card would need to be involved in the stimuli or the states. The state size of the first card is all the possible combinations of the AI bot's 5 cards as none of the opponent's cards are revealed yet. For the subsequent cards, the size of the states is all possible combinations of the remaining cards plus the opponent's last played card. To ensure that no two states share the same Q-value, each state is assigned a hash value (address). This hash includes in its encoding the all the cards that define the state.

```
Q-Learning Algorithm (for 5 cards)
1:   Initialize $S_{i=0 \text{ to } 4}$ = number of all possible combinations of hand of cards, (i=0 → first card)
2:   Initialize $A_{j=0 \text{ to } 4} = 5 - j$, which is the number of available actions
3:   Initialize 5 Q ($S_i$, $A_j$) tables to store Q-values. One table for each of the cards
4:   Initialize $\alpha$ = Learning rate
5:   Initialize $\gamma$ = discount factor
6:   while Q not converged do
7:       for step t ← 1 to T do
8:           simulate a WarCAT game
9:           reward = $\begin{cases} \text{cards 1 to 4: score of the single card played} \\ \text{card5: final score} \end{cases}$
10:          From card 1 to card 5:
11:              New $Q(S_i, A_j) \leftarrow$ Old $Q(S_i, A_j) + \alpha * (reward + \gamma * \max(Q(S_{i+1})))$
12:      end
13:  end
```

Figure 5-4 Pseudocode of the version of Q-Learning used in this work

5.2.3.2 Complete Solution (brute force)

Another approach is not implemented in this work but it is worth mentioning. The approach is brute force search solution. There are 13^ (5 player cards+5 opponent cards) = about 138 billion possible cases. There are databases that can handle such size. Also, this size is not impossible for a cloud computation to perform operations. For example, GPT-3, which is language model developed by the AI company OpenAI, had 175 billion parameters [49]. Meanwhile the cost of computation is decreasing with time. However, this requires too much computational resources that are beyond the size of this work.

5.2.3.3 Results

The highest average score possible is ~25 if perfect knowledge is provided about the opponent's cards and the order in which they are played. On the other hand, the human's highest average score was about 18.5 from a tournament of about 30 participants. This average score is exceeded by the AI bot as shown in Table 5-3 and Figure 5-5 at around 275 million Epochs. Each score average shown is the average of 1 million games played using the AI bot that was trained with that number of Epochs.

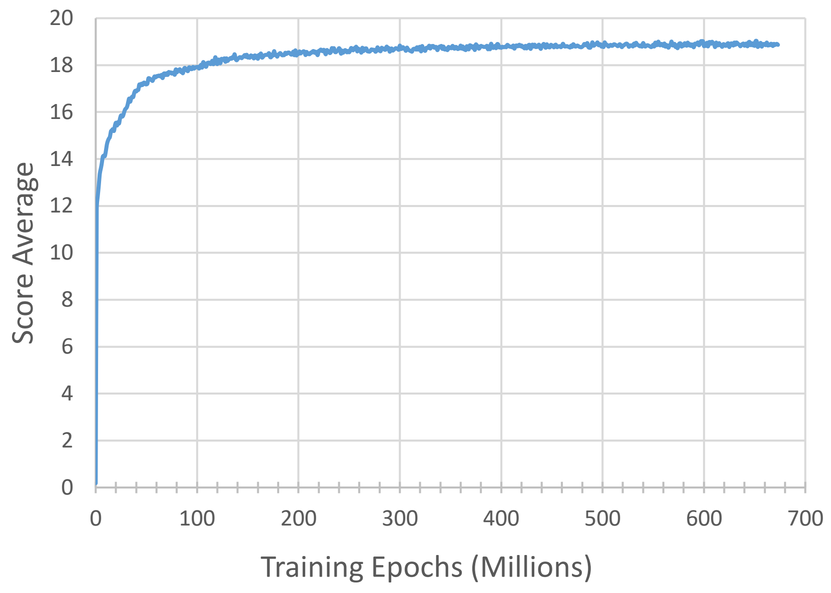


Figure 5-5 The Score average performance of the RL bot

Table 5-3 Levels Used to Emulate the AI Bot's Score Average

Epochs (Millions)	**Average Score**	**Epochs (Millions)**	**Score Average**
0	0.18304	50	17.17993
1	11.86888	100	17.9266
2	12.40905	200	18.47558
5	13.58131	230	18.58362
10	14.33171	275	18.64247
15	15.17609	400	18.73655
30	16.16207	660	18.8828

The table above shows the various learning periods represent prototypes of various cognitive abilities based upon their degree of training.

5.3 Classification

5.3.1 Introduction

The CNN classifier (discussed in Section 4.2.4 above) will be used here as it yielded slightly better results that its counterparts such as: simple statistical model, DNN, and FFT models.

5.3.2 Improving the data

One way of improving the data is lowering the variance. As the game is stochastic in nature, there is considerable variation in the score per game. This variation will present a challenge for any ML classifier. The variation can be mitigated to a large degree by normalizing play against the best score possible, which is, on average, calculated at a value of ~25, again with some variation associated with the hands dealt. Figure 5-6 illustrates an example of normalizing the scoring against the maximum possible per hand for each player. In effect, the result is the “residue” or remainder, or points left on the table for a given hand. Thus, the variance can be reduced by approximately 60% as shown.

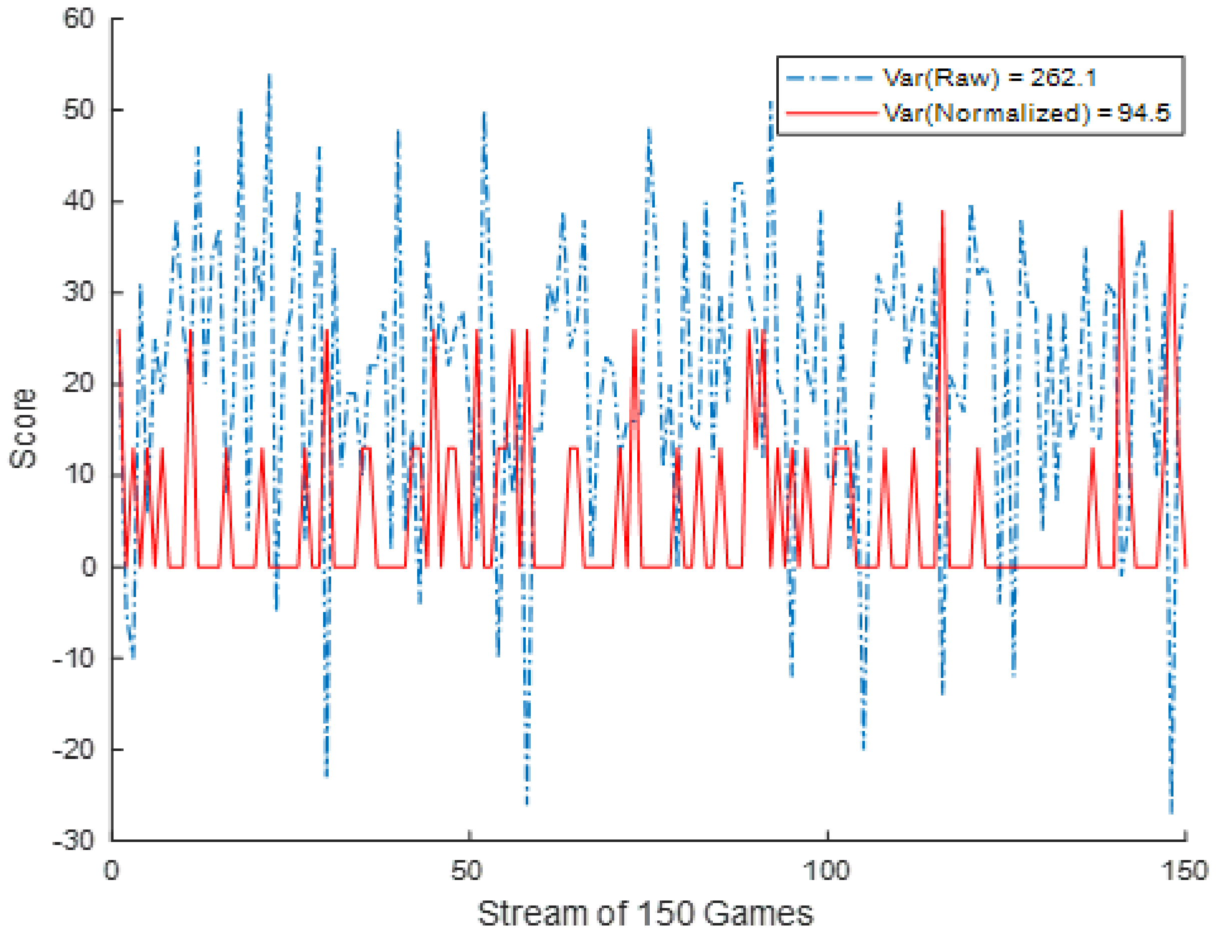


Figure 5-6 Reduction in variance using the residue from best possible score for each hand

The reduction in variance seen in Figure 5-6 is a result of the difference between two Gaussian distributions (one associated with maximum scores possible given the cards that were dealt and the score the player bot achieved after learning). The variance is reduced as there is a high covariance between the best score obtainable and the score the player bots achieved. This observation lends additional credibility to a ML classifier as the classification of players with various levels of training should form more recognizable clusters with less overlap.

From these residual data of 150 games, one constructs a fingerprint of play. Each fingerprint consists of the score residue of each game (residue from maximum score), the average of the 150 scores, and the average of the 150 maximum scores. This leaves an input vector of ML features totaling 152 inputs as representing a person's play.

These patterns of play of "cognitive fingerprints", represent play associated with various levels of learning or cognitive acuity. The conjecture is that the differences in these "cognitive fingerprints" will be significantly easier to classify using the normalized score per hand than the actual scores. An interesting artifact of play is that although the scores appear highly erratic (normalized data in Figure 5-6), they are significantly quantized when compared to the maximum score achievable for that game. The reason for this effect is that the quantization is directly related to patterns of play associated with misplayed cards. Basically, these represent permutations of (at most) three misplayed cards, which results in score differences of 0, 13, 26 or 39 points.
Using RL bot developed in Section 5.2.3.1, five different stages of training were used to generate five different classes of gameplay data, as shown in Table 5-4 below.

Table 5-4 RL Bots at 5 Different Learning Stages and Their Average Scores

Class	*Epochs (Training)*	*Score Average*
RL Class1	1,000,000	11.88046
RL Class2	5,000,000	13.59851
RL Class3	15,000,000	15.0722
RL Class4	50,000,000	17.23785
RL Class5	660,000,000	18.92467

Based on these observations and new features, another attempt was made at classifying the now-quantized synthetic data using a CNN. Five classes were extracted from the AI or RL bot at different stages of learning as shown in Table 5-4, where each class represents a different degree of a player's cognitive impairment. For each class, over $8e10^6$ games were simulated from those RL bots. Every 150 games were grouped to make a stream of games emulating individual players' cognitive fingerprints, thus totaling a respectable 55,000 games or synthetic players. Those were cast into training set patterns of 50,000 and validation set patterns of 5,000. First, the CNN model was trained on two classes: RL Class4 and RL Class5 from Table 5-4. Figure 5-7 show the cross-validation accuracy of 83.5%. This can be interpreted as the classifier's ability to distinguish two neighboring degrees of cognitive impairment with an accuracy of 83.5%.

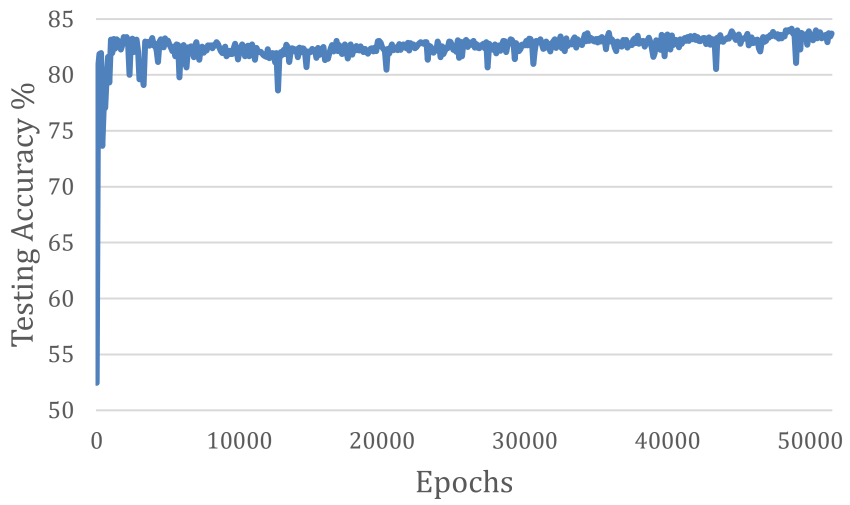


Figure 5-7 (RL Class4, RL Class5), Testing size = 5000, Training size = 50000

It is expected that it would be more difficult for the model to classify two neighboring classes from Table II than to classify two classes that are further apart. Therefore, Figure 5-8 shows a higher accuracy of 97.9% when the model was training on the two classes: RL Class3 and RL Class5. This can be interpreted as the classifier's ability to distinguish two very different degrees of cognitive impairment with an accuracy of 97.9%.

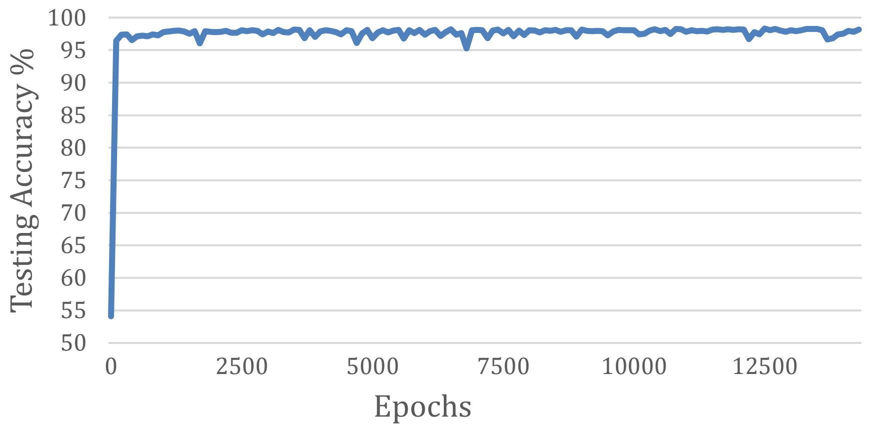


Figure 5-8 (RL Class3, RL Class5), Testing size = 5000, Training size = 50000

A sixth class added was the "Basic Strategy" associated with always sacrificing the lowest card first then playing the remainder high to low. This strategy gives an average score of 14.71. This class, along with RL Class3, were used to train the model since these two classes share close score averages. The testing accuracy result was 65.3% as illustrated in Figure 5-9. This can be interpreted as the classifier's ability to distinguish between medium-level cognitive impairment and the rote implementation of a memorized strategy with no hand-by-hand judgement with an accuracy of 65.3%.

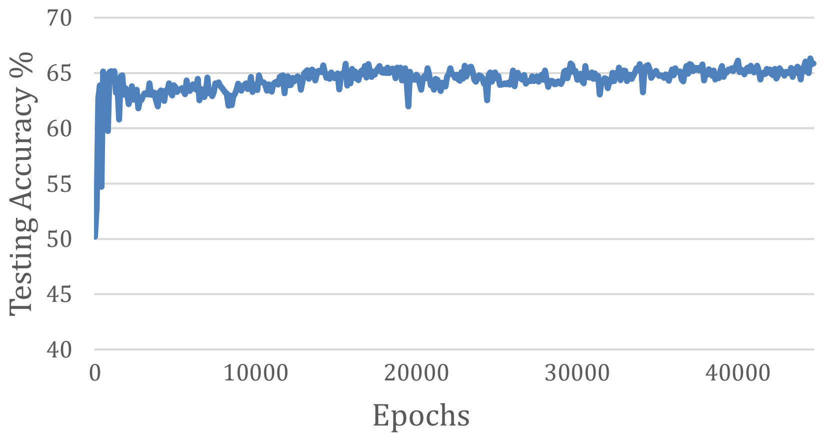


Figure 5-9 (Basic Strategy, RL class3), Testing size = 5000, Training size = 50000

It is interesting to note that the accuracy was higher when the same model in Figure 5-9 was trained on a randomly selected subset of its dataset. Figure 5-10 shows a testing accuracy of 75.5% when the training set was 10,000 and the validation set was 1,000. The reason for this unexpected increase in accuracy is currently unknown. It also manifests when the models shown in Figure 5-7 and Figure 5-8 use smaller datasets. The model was also trained on the five RL classes at once. Figure 5-11 illustrates the testing accuracy.

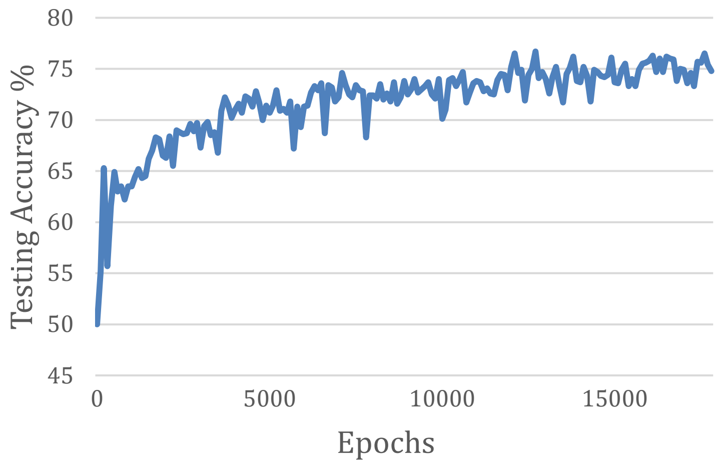


Figure 5-10 Two classes (Basic strategy, RL class3), Testing size = 1000, Training size = 10000

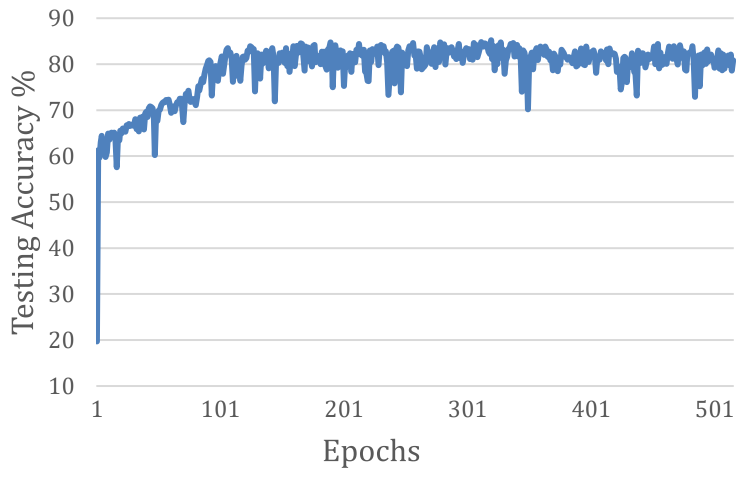


Figure 5-11 Five RL classes, Testing size = 1000, Training size = 10000

5.4 Results Summary

The results of the RL endeavor is that a 1.6 difference in score average leads to around 83% to 86% in accuracy in classification (without intermediary model). This is an encouraging result given that such a difference in score average exists among healthy students from the 30 student tournament. It is likely then that such difference would exist between a healthy person and a person with MCI.

Chapter 6: Extensions to This Work

6.1 Getting gameplay data from people with MCI

A lot the results in this work depend on how different the two classes of data are. It is not clear how much a person with MCI differs in performance in the WarCAT game from a healthy person. Therefore, getting real human data from healthy people and crucially from people with varying levels of MCI can illuminate much of the importance of the results. Even, a small amount of real data can serve as baseline similar to the 30-person tournament data used here. Besides giving an idea of the significance of the applicability of this work, real data can test how engaging the game is with people with MCI. While technically feasible, challenges include finding enough participants in each demographic category, and the human ethics implications of gathering data from human subjects. Real human data (if available) can also be used as a part in the training data set of the ML models. Its role can then gradually increase as more data becomes available.

Chapter 7: Summary & Conclusion

To potentially detect MCI, ML algorithms are utilized in conjunction with the WarCAT SG in two ways. First, an RL bot was trained until it surpassed human performance to generate several classes of artificial player data. Second, CNN classifiers were trained on data from the RL bots at different levels of performance. Currently, they can classify two RL learning stages with 1.69 average score performance difference with 83.5% accuracy. A higher average score difference gives even greater discrimination accuracy. These results bode well in justifying ML approaches within SGs to help in the early detection of MCI. The next step will be to induce distractions, “loss of set’, or momentary lapses of concentration within the RL bots’ play to emulate memory slips or “senior moments”. Future work will also investigate the use of Generative Adversarial Networks (GANs) for data synthesis and Recurrent Neural Networks (RNNs) for classification.

This application of using WarCAT to potentially detect subtle cognitive impairment is analogous to signal processing techniques used to correct for errors in a communication channel. Error control coding is directly analogous to MCI and the brain’s ability to adapt and correct. This capability of the brain is also one of the reasons detecting MCI is so difficult. Take a familiar QR code as an example as shown in Figure 7-1 below.

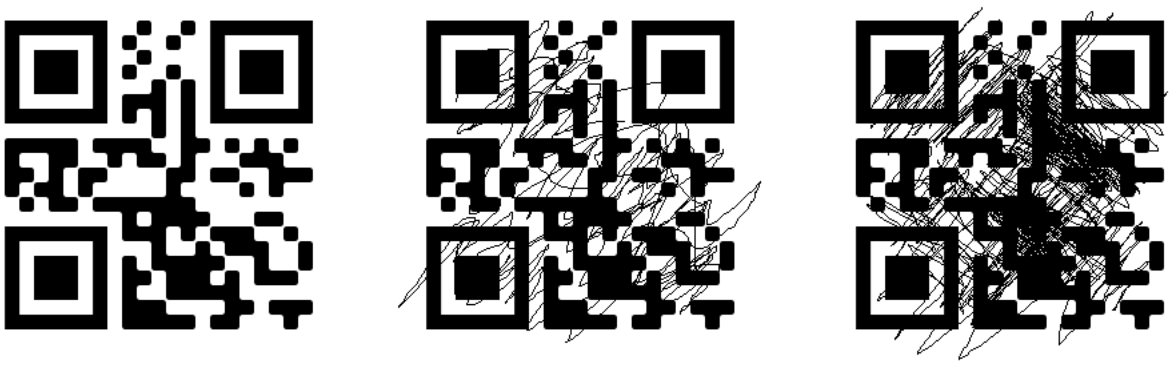


Figure 7-1 QR codes without error, with minimal defects, and with considerable defects

The left hand QR code has no defects, analogous to no cognitive decline and easily captured by a QR code reader, the middle has discernable defects but still easily decoded by a QR code reader as consequence of the ECC, the right side QR code has significant number of defects making it difficult or impossible to decode the QR code.

As in the case with cognitive impairments, often when a positive clinical diagnosis is made, there is already too great a degree of cognitive impairment for therapeutic interventions to be useful. The ML approaches presented here on gameplay data are akin to building error monitoring tools that attempt to recognize subtleties with ML classifiers before they are obvious to the 'user' or others around them. This type of instrumentation is available within ECC hardware although rarely exploited,

In a communication channel, signal processing techniques are used to correct for errors in much the same manner as the human brain corrects for degradation in function. Similar to a noisy communication channel, the person listening to music on their phone may not even know that the channel was noisy, as the errors were corrected by software and hardware before the music was presented to the listener and thus remain beyond the user's detection or awareness. It is only when the channel noise increases to such a degree that it cannot be corrected, that a person would realize that the communication channel was seriously degraded, perhaps beyond repair or correction. Modern communication service and the devices themselves maintain channel condition reports and error rate statistics. These concepts provide direction that instrumenting serious games and other surveillance systems might employ to help detect MCI.

Presently, there is limited work on serious games that can detect MCI or any pre-dementia stage of abnormal cognitive decline. If ML is to be effective, considerable data are required. Many areas in which ML techniques have demonstrated utility are those where data are already available, or if not, where data can be generated synthetically. Ideally, one would like to have labeled data such that a supervised ML algorithm can be deployed. Of the more successful of these are CNNs, where they have been incredibly successful for image, music and speech processing. The cognitive fingerprints – based off real data as well as the synthetic data - have a very close similarity to one-dimensional images or spectrograms and as such may bear similar fruit in ML classification by CNNs. Another strong ML contender for this type of data are RNNs as well as hybrids of both. RNNs are well suited to time series data and as such worth consideration as gameplay can be posited as a type of time series data.

Although serious games and the challenges of data collection and processing was presented here, it certainly is not the only platform that could be employed to help detect MCI. Many games are likely more suitable, but all will face similar challenges. Others, perhaps more invasive and nefarious, may be built upon Google's, Amazon's, and Apple's personal assistants. It should also be noted that RL is only one of several means to generate synthetic data; RL just happens to be more widely used in games than alternative techniques.

REFERENCES

[1] K. M. Langa and D. A. Levine, "The Diagnosis and Management of Mild Cognitive Impairment," *JAMA*, vol. 312, no. 23, p. 2551, Dec. 2014, doi: 10.1001/jama.2014.13806.

[2] A. C. Tricco, C. Soobiah, E. Lillie, L. Perrier, M. H. Chen, B. Hemmelgarn, S. R. Majumdar, and S. E. Straus, "Use of cognitive enhancers for mild cognitive impairment: Protocol for a systematic review and network meta-analysis," *Systematic Reviews*, vol. 1, no. 1, pp. 1–6, 2012, doi: 10.1186/2046-4053-1-25.

[3] R. Roberts and D. S. Knopman, "Classification and epidemiology of MCI," *Clinics in Geriatric Medicine*, vol. 29, no. 4, pp. 753–772, Nov. 2013, doi: 10.1016/j.cger.2013.07.003.

[4] "World Alzheimer Report 2018," 2018. [Online]. Available: https://www.alz.co.uk/research/world-report-2018

[5] R. Gallassi, F. Oppi, R. Poda, S. Scortichini, M. S. Maserati, G. Marano, L. Sambati, "Are subjective cognitive complaints a risk factor for dementia," *Neurological Sciences*, vol. 31, no. 3, pp. 327–336, 2010, doi: 10.1007/s10072-010-0224-6.

[6] A. Nordlund, S. Rolstad, O. Klang, Å. Edman, S. Hansen, and A. Wallin, "Two-year outcome of MCI subtypes and aetiologies in the Göteborg MCI study," *Journal of Neurology, Neurosurgery &amp;amp; Psychiatry*, vol. 81, no. 5, pp. 541 LP – 546, May 2010, doi: 10.1136/jnnp.2008.171066.

[7] M. Ganguli, H. H. Dodge, C. Shen, and S. T. DeKosky, "Mild cognitive impairment, amnestic type," *Neurology*, vol. 63, no. 1, pp. 115 LP – 121, Jul. 2004, doi: 10.1212/01.WNL.0000132523.27540.81.

[8] M. Ganguli, B.E. Snitz, J.A. Saxton, C-C. H. Chang, C-W. Lee, J. Vander Bilt, T.F. Hughes, D. A. Loewenstein, F.W. Unverzagt, and R.C. Petersen, "Outcomes of mild cognitive impairment by definition: A population study," *Archives of Neurology*, vol. 68, no. 6, pp. 761–767, 2011, doi: 10.1001/archneurol.2011.101.

[9] P. E. J. Spaan, "Episodic and semantic memory impairments in (very) early Alzheimer's disease: The diagnostic accuracy of paired-associate learning formats," *Cogent Psychology*, vol. 3, no. 1, pp. 1–25, 2016, doi: 10.1080/23311908.2015.1125076.

[10] R. C. Petersen, G. E. Smith, S. C. Waring, R. J. Ivnik, E. G. Tangalos, and E. Kokmen, "Mild Cognitive Impairment," *Archives of Neurology*, vol. 56, no. 3, p. 303, Mar. 1999, doi: 10.1001/archneur.56.3.303.

[11] 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*, vol. 3, no. 4, pp. E419–E427, Nov. 2015, doi: 10.9778/cmajo.20150057.

[12] 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*, vol. 2, no. 1, 2018, doi: 10.15761/rri.1000128.

[13] S. McCallum, "Gamification and serious games for personalized health.," *Stud Health Technol Inform*, vol. 177, pp. 85–96, 2012, [Online]. Available: http://www.embase.com/search/results?subaction=viewrecord&from=export&id=L365722648%0Ahttp://limo.libis.be/resolver?&sid=EMBASE&issn=09269630&id=doi:&atitle=Gamification+and+serious+games+for+personalized+health.&stitle=Stud+Health+Technol+Inform&title=St

[14] 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*, vol. 7, p. 26, Mar. 2015, doi: 10.3389/fnagi.2015.00026.

[15] T. Tong, M. Chignell, M. C. Tierney, and J. Lee, “A Serious Game for Clinical Assessment of Cognitive Status: Validation Study,” *JMIR Serious Games*, vol. 4, no. 1, p. e7, 2016, doi: 10.2196/games.5006.

[16] S. Hagler, H. B. Jimison, and M. Pavel, “Assessing executive function using a computer game: Computational modeling of cognitive processes,” *IEEE Journal of Biomedical and Health Informatics*, vol. 18, no. 4, pp. 1442–1452, 2014, doi: 10.1109/JBHI.2014.2299793.

[17] H. Jimison and M. Pavel, “Embedded assessment algorithms within home-based cognitive computer game exercises for elders,” *Annual International Conference of the IEEE Engineering in Medicine and Biology - Proceedings*, vol. 1, pp. 6101–6104, 2006, doi: 10.1109/IEMBS.2006.260303.

[18] C. Basak, M. W. Voss, K. I. Erickson, W. R. Boot, and A. F. Kramer, “Regional differences in brain volume predict the acquisition of skill in a complex real-time strategy videogame.,” *Brain Cogn*, vol. 76, no. 3, pp. 407–14, Aug. 2011, doi: 10.1016/j.bandc.2011.03.017.

[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*, vol. 12, no. 5, pp. 1–14, 2017, doi: 10.1371/journal.pone.0175999.

[20] O. Thompson, S. Barrett, C. Patterson, and D. Craig, “Examining the Neurocognitive Validity of Commercially Available, Smartphone-Based Puzzle Games,” *Psychology*, vol. 03, no. 07, pp. 525–526, 2012, doi: 10.4236/psych.2012.37076.

[21] S. Verschueren, C. Buffel, and G. vander Stichele, “Developing Theory-Driven, Evidence-Based Serious Games for Health: Framework Based on Research Community Insights,” *JMIR Serious Games*, vol. 7, no. 2, p. e11565, May 2019, doi: 10.2196/11565.

[22] J. R. Cockrell and M. F. Folstein, “Mini-Mental State Examination (MMSE).,” *Psychopharmacol Bull*, vol. 24, no. 4, pp. 689–92, 1988, [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/3249771

[23] 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 US Preventive Services Task Force (Structured abstract).,” *Database of Abstracts of Reviews of Effects*, no. 107, pp. 1–411, 2013, [Online]. Available: http://ovidsp.ovid.com/ovidweb.cgi?T=JS&PAGE=reference&D=dare&NEWS=N&AN=00125498-100000000-39011

[24] “Tip Sheet 4 – The Clock Drawing Test (CDT),” 2011. https://www2.health.vic.gov.au/about/publications/policiesandguidelines/acas-dementia-clock-drawing-test-tip-sheet-4

[25] “Serious Games and Machine Learning for Detecting Mild Cognitive Impairment - SG-ANN2019 v2.”

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

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Commun ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

[28] 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. [Online]. Available: https://arxiv.org/pdf/1804.09699.

[29] H. Ide and T. Kurita, “Improvement of learning for CNN with ReLU activation by sparse regularization,” in *2017 International Joint Conference on Neural Networks (IJCNN),* May 2017, pp. 2684–2691. doi: 10.1109/IJCNN.2017.7966185.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, doi: 10.1038/323533a0.

[31] P. Probst and B. Bischl, “Tunability: Importance of Hyperparameters of Machine Learning Algorithms,” 2019. [Online]. Available: http://jmlr.org/papers/v20/18-444.html.

[32] M. X. Cohen and C. Ranganath, “Reinforcement learning signals predict future decisions,” *Journal of Neuroscience,* vol. 27, no. 2, pp. 371–378, Jan. 2007, doi: 10.1523/JNEUROSCI.4421-06.2007.

[33] R. Daniel and S. Pollmann, “A universal role of the ventral striatum in reward-based learning: Evidence from human studies,” *Neurobiology of Learning and Memory*, vol. 114. Academic Press Inc., pp. 90–100, 2014. doi: 10.1016/j.nlm.2014.05.002.

[34] M. de Bruijne, “Machine learning approaches in medical image analysis: From detection to diagnosis,” *Medical Image Analysis*, vol. 33. Elsevier B.V., pp. 94–97, Oct. 01, 2016. doi: 10.1016/j.media.2016.06.032.

[35] J. Drechsler and J. P. Reiter, “An empirical evaluation of easily implemented, nonparametric methods for generating synthetic datasets,” *Computational Statistics and Data Analysis*, vol. 55, no. 12, pp. 3232–3243, 2011, doi: 10.1016/j.csda.2011.06.006.

[36] E. O. Elliott, “Estimates of Error Rates for Codes on Burst-Noise Channels,” *Bell System Technical Journal*, vol. 42, no. 5, pp. 1977–1997, Sep. 1963, doi: 10.1002/j.1538-7305.1963.tb00955.x.

[37] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber, “A Novel Connectionist System for Unconstrained Handwriting Recognition.” [Online]. Available: www.e-Beam.com

[38] O. Abdel-Hamid, A. R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, “Convolutional neural networks for speech recognition,” *IEEE Transactions on Audio, Speech and Language Processing*, vol. 22, no. 10, pp. 1533–1545, Oct. 2014, doi: 10.1109/TASLP.2014.2339736.

[39] K. Choi, G. Fazekas, M. Sandler, and K. Cho, “Convolutional recurrent neural networks for music classification,” *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pp. 2392–2396, 2017, doi: 10.1109/ICASSP.2017.7952585.

[40] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation,” Sep. 2016, [Online]. Available: http://arxiv.org/abs/1609.08144

[41] G. Ning, Z. Zhang, C. Huang, Z. He, X. Ren, and H. Wang, “Spatially Supervised Recurrent Convolutional Neural Networks for Visual Object Tracking,” Jul. 2016, [Online]. Available: http://arxiv.org/abs/1607.05781

[42] C. Farabet, C. Couprie, L. Najman, and Y. Lecun, “Learning hierarchical features for scene labeling,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1915–1929, 2013, doi: 10.1109/TPAMI.2012.231.

[43] 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, & D. Hassabis, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: 10.1038/nature16961.

[44] L. Deng, “The MNIST database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012, doi: 10.1109/MSP.2012.2211477.

[45] M. Stone, “Cross-validatory Choice and Assessment of Statistical Predictions.”

[46] S. Kiranyaz, T. Ince, and M. Gabbouj, “Real-Time Patient-Specific ECG Classification by 1-D Convolutional Neural Networks,” *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 3, pp. 664–675, 2016, doi: 10.1109/TBME.2015.2468589.

[47] K. Gutenschwager, M. Aljumaili, R. D. Mcleod, and M. R. Friesen, “Visualizing ‘ Cognitive Fingerprints ’ from Simple Mobile Game Play,” in *CMBEC42*, 2019, vol. 42.

[48] R. van Eck, “Building Artificially Intelligent Learning Games,” in *Games and Simulations in Online Learning*, IGI Global, 2007, pp. 271–307. doi: 10.4018/978-1-59904-304-3.ch014.

[49] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin,

S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language Models are Few-Shot Learners,” 2020. doi: https://doi.org/10.48550/arXiv.2005.14165.

Appendix A WarCAT Gameplay Data Sample

Bot Card1	Bot Card2	Bot Card3	Bot Card4	Bot Card5	Player Card1	Player Card2	Player Card3	Player Card4	Player Card5	Win/ Loss	Score	Max Score	Win Percentage	Expected Score
13	10	6	5	2	3	14	14	6	4	1	34	34	0.7	5.4
12	11	10	4	3	3	13	11	11	10	1	31	31	0.7	5
7	6	5	3	2	4	13	13	10	4	1	18	44	0.7	12.8
9	9	8	6	5	2	13	12	8	7	1	34	34	0.7	5.4
14	8	7	5	4	2	13	6	5	4	-1	-5	21	0.1	-15.4
14	13	11	7	4	3	11	11	8	4	-1	-1	12	0.07	-14
13	12	10	7	3	2	9	5	3	3	-1	-29	10	0.05	-21.2
12	11	9	5	3	3	14	14	7	4	1	37	37	0.55	3.2
12	8	6	3	2	7	14	12	11	10	1	16	42	0.9	18.6
14	11	5	4	3	4	13	12	9	5	1	33	33	0.72	7
14	11	10	4	3	5	12	10	8	6	1	27	27	0.55	1
10	5	5	3	2	3	13	10	5	3	1	30	30	0.77	6.6
14	11	9	4	3	5	14	11	7	6	1	37	37	0.6	5.8
13	9	8	6	3	3	11	10	8	5	1	41	41	0.53	-0.6
8	8	7	6	5	4	13	9	9	6	1	32	32	1	13.8
14	14	13	10	4	3	12	10	7	4	-1	-33	6	0.1	-20
13	13	6	4	2	3	14	13	6	5	1	36	36	0.63	7.4
14	11	6	5	4	4	13	12	8	8	1	34	34	0.7	5.4

Appendix B 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. This code is not the whole code that was used. However, it should provide most of the picture of the used code.

requirements.txt (installed on python3.8)

```
numpy==1.22.3
matplotlib==3.5.1
seaborn==1.8.0
scikit-learn==1.0.2
tensorflow==2.8.0
```

simple_data_generator.py (used in Section 4.1)

```
import numpy as np
import random

class SimpleDataGenerator:
    def __init__(self, classes_parameters):
        self.game_size = []
        self.data_type = []
        self.label = []
        self.bin_p = []
        self.bur_p = []
        self.bur_r = []
        self.bur_k = []
        self.bur_h = []
        self.seed = []
        self.classes_num = 0
        for parameters in classes_parameters:
            assert parameters["data_type"] in ["binomial", "burst_error"]
            self.game_size.append(parameters["game_size"])
            self.data_type.append(parameters["data_type"])
            self.label.append(parameters["label"])
            self.bin_p.append(parameters["bin_p"])
            self.bur_p.append(parameters["bur_p"])
            self.bur_r.append(parameters["bur_r"])
            self.bur_k.append(parameters["bur_k"])
            self.bur_h.append(parameters["bur_h"])
            self.seed.append(parameters["seed"])
            self.classes_num += 1

    def generate(self, data_size):
        data = []
        labels = []
        for i in range(self.classes_num):
            if self.seed[i] is not None:
                np.random.seed(self.seed[i])
                random.seed(self.seed[i])

            for _ in range(data_size):
                if self.data_type[0] == "binomial":
                    data.append(np.random.binomial(n=1, p=self.bin_p[i], size=self.game_size[i]))
                else:
                    games = []
                    state = bool(random.randint(0, 1))
                    for _ in range(self.game_size[i]):
                        if state:
                            x = int(random.random() < self.bur_k[i])
                            state = not bool(random.random() < self.bur_p[i])
                        else:
                            x = int(random.random() < self.bur_h[i])
                            state = bool(random.random() < self.bur_r[i])
                        games.append(x)
                    data.append(np.asarray(games))
                labels.append(self.label[i])

        data = np.asarray(data, np.float32)
        labels = np.asarray(labels, np.float32)
        return data, labels

    def get_burst_error_rates(self):
        error_rates = []
        for i in range(self.classes_num):
            error_rates.append((1 - self.bur_k[i]) * (self.bur_r[i] / (self.bur_r[i] + self.bur_p[i])) + (
                    1 - self.bur_h[i]) * (self.bur_p[i] / (self.bur_r[i] + self.bur_p[i])))
        return error_rates


if __name__ == "__main__":
    class1_params = {"game_size": 4,
                     "data_type": "binomial",
                     "label": 0,
                     "bin_p": 0.5,
                     "bur_p": 0,
                     "bur_r": 1,
                     "bur_k": 0.75,
                     "bur_h": 0.5,
                     "seed": None}
    class2_params = {"game_size": 4,
                     "data_type": "binomial",
                     "label": 1,
                     "bin_p": 0.65,
                     "bur_p": 0.13333,
                     "bur_r": 0.43,
                  "bur_k": 0.75,
                  "bur_h": 0.5,
                  "seed": None}
    classes_params = [class1_params, class2_params]
    class_gen = SimpleDataGenerator(classes_params)
    print(class_gen.generate(3))
    print(class_gen.get_burst_error_rates())
```

average_based_2classes.py (used in Section 4.2.1)

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

from simple_data_generator import SimpleDataGenerator

seed = 0

class1_params = {"game_size": 100,
               "data_type": "burst_error",  # "binomial" or "burst_error"
               "label": 0,
               "bin_p": 0.5,
               "bur_p": 0.5,  # p to good state
               "bur_r": 0.5,  # p to bad state
               "bur_k": 0.75,  # p of 1 in good state
               "bur_h": 0.5,  # p of 1 in bad state
               "seed": None}
class2_params = {"game_size": 100,
               "data_type": "burst_error",
               "label": 1,
               "bin_p": 0.65,
               "bur_p": 0.2,
               "bur_r": 0.2,
               "bur_k": 0.75,
               "bur_h": 0.5,
               "seed": None}

prms = [class1_params, class2_params]
data_gen = SimpleDataGenerator(prms)
data, labels = data_gen.generate(10000)
bur_errors = data_gen.get_burst_error_rates()
print(bur_errors)

cutoff_threshold = np.mean([prms[i]["bin_p"] for i in range(len(prms))]) if prms[0]["data_type"] ==
"binomial" else 1 - np.mean(bur_errors)

preds = [int(np.mean(data[i]) > cutoff_threshold) for i in range(len(data))]
tn, fp, fn, tp = confusion_matrix(labels, preds).ravel()
sensitivity = tp / (tp + fn)
```

```
specificity = tn / (tn + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)

print(f"Sensitivity = {sensitivity}")
print(f"Specificity = {specificity}")
print(f"Accuracy = {accuracy}")

avg1 = [np.mean(data[i]) for i in range(len(labels)) if labels[i] == 0]
avg2 = [np.mean(data[i]) for i in range(len(labels)) if labels[i] == 1]
print("class 1 avg: " + str(np.mean(avg1)))
print("class 2 avg: " + str(np.mean(avg2)))

fig, ax = plt.subplots()

sns.kdeplot(avg1, bw_method=0.1, label="Histogram of class1 Averages")
sns.kdeplot(avg1, bw_method=0.1, clip=[cutoff_threshold, 1], fill=True)
sns.kdeplot(avg2, bw_method=0.1, label="Histogram of class2 Averages")
sns.kdeplot(avg2, bw_method=0.1, clip=[0, cutoff_threshold], fill=True)
plt.axvline(x=cutoff_threshold, color='r', linewidth=3, linestyle='dashdot', label="cutoff threshold")
plt.legend(loc="upper left")
plt.title(f"{prms[0]['data_type']} Data")
plt.show()
```

dnn.py (used in Section 4.2.2)

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import numpy as np
import time

from simple_data_generator import SimpleDataGenerator

inputs = keras.Input(shape=(100,), name="digits")
x = layers.Dense(200, activation="relu", name="dense_1")(inputs)
x = layers.Dense(100, activation="relu", name="dense_2")(x)
outputs = layers.Dense(2, activation="softmax", name="predictions")(x)

model = keras.Model(inputs=inputs, outputs=outputs)



class1_params = {"game_size": 100,
            "data_type": "burst_error",  # "binomial" or "burst_error"
            "label": 0,
            "bin_p": 0.5,
            "bur_p": 0.5,  # p to good state
            "bur_r": 0.5,  # p to bad state
            "bur_k": 0.75,  # p of 1 in good state
```

```
                "bur_h": 0.5,  # p of 1 in bad state
                "seed": None}
class2_params = {"game_size": 100,
                "data_type": "burst_error",
                "label": 1,
                "bin_p": 0.65,
                "bur_p": 0.2,
                "bur_r": 0.2,
                "bur_k": 0.75,
                "bur_h": 0.5,
                "seed": None}

prms = [class1_params, class2_params]
data_gen = SimpleDataGenerator(prms)
bur_errors = data_gen.get_burst_error_rates()
print([1 - x for x in bur_errors])

x_train, y_train = data_gen.generate(100000)
x_test, y_test = data_gen.generate(10000)

y_train = y_train.astype("float32")
y_test = y_test.astype("float32")

model.compile(
    optimizer=keras.optimizers.Adam(),  # Optimizer
    # Loss function to minimize
    loss=keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)

start_time = time.time()
print("Fit model on training data")
history = model.fit(
    x_train,
    y_train,
    batch_size=1000,
    epochs=70,
    # We pass some validation for monitoring validation loss and metrics at the end of each epoch
    validation_data=(x_test, y_test),
    shuffle=True
)
run_time = int(time.time() - start_time)

fig = plt.figure()
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss Value')
plt.title('DNN: Cross Entropy Loss')
plt.show()
#fig.savefig(f'DNN_Cross_Entropy_Loss_Adam_2c_(run_time={run_time}_sec)_0.2.svg',
format="svg")


fig = plt.figure()
ax = plt.axes()
plt.plot(np.array(history.history['sparse_categorical_accuracy'])*100, label='Training Accuracy')
plt.plot(np.array(history.history['val_sparse_categorical_accuracy'])*100, label='Validation Accuracy')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy %')
ax.yaxis.set_major_formatter(mtick.PercentFormatter())
plt.title('DNN: Training & Validation Accuracies')
plt.show()
#fig.savefig(f'DNN_Training_&_Validation_Accuracies_Adam_2c_(run_time={run_time}_sec)_0.2.s
vg', format="svg")

max_val = np.max(history.history['val_sparse_categorical_accuracy'])
print(f'max val accuracy: {max_val}')
```

fft.py (used in Section 4.2.3)

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import numpy as np
from scipy.fftpack import fft
import time

from simple_data_generator import SimpleDataGenerator

inputs = keras.Input(shape=(100,), name="digits")
x = layers.Dense(200, activation="relu", name="dense_1")(inputs)
x = layers.Dense(100, activation="relu", name="dense_2")(x)
outputs = layers.Dense(2, activation="softmax", name="predictions")(x)

model = keras.Model(inputs=inputs, outputs=outputs)

class1_params = {"game_size": 100,
            "data_type": "burst_error",  # "binomial" or "burst_error"
            "label": 0,
            "bin_p": 0.5,
            "bur_p": 0.5,  # p to good state
            "bur_r": 0.5,  # p to bad state
            "bur_k": 0.75,  # p of 1 in good state
```

```
              "bur_h": 0.5,  # p of 1 in bad state
              "seed": None}
class2_params = {"game_size": 100,
              "data_type": "burst_error",
              "label": 1,
              "bin_p": 0.65,
              "bur_p": 0.2,
              "bur_r": 0.2,
              "bur_k": 0.75,
              "bur_h": 0.5,
              "seed": None}

prms = [class1_params, class2_params]
data_gen = SimpleDataGenerator(prms)
bur_errors = data_gen.get_burst_error_rates()
print([1 - x for x in bur_errors])

x_train, y_train = data_gen.generate(100000)
x_test, y_test = data_gen.generate(10000)

x_train = np.abs(fft(x_train))
x_test = np.abs(fft(x_test))

y_train = y_train.astype("float32")
y_test = y_test.astype("float32")

model.compile(
    optimizer=keras.optimizers.Adam(),  # Optimizer
    # Loss function to minimize
    loss=keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)

start_time = time.time()
print("Fit model on training data")
history = model.fit(
    x_train,
    y_train,
    batch_size=1000,
    epochs=70,
    # We pass some validation for monitoring validation loss and metrics at the end of each epoch
    validation_data=(x_test, y_test),
    shuffle=True
)
run_time = int(time.time() - start_time)

fig = plt.figure()
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss Value')
plt.title('FFT: Cross Entropy Loss')
plt.show()
# fig.savefig(f'FFT_Cross_Entropy_Loss_Adam_2c_(run_time={run_time}_sec)2.svg', format="svg")

fig = plt.figure()
ax = plt.axes()
plt.plot(np.array(history.history['sparse_categorical_accuracy'])*100, label='Training Accuracy')
plt.plot(np.array(history.history['val_sparse_categorical_accuracy'])*100, label='Validation Accuracy')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy %')
ax.yaxis.set_major_formatter(mtick.PercentFormatter())
plt.title('FFT: Training & Validation Accuracies')
plt.show()
#fig.savefig(f'FFT_Training_&_Validation_Accuracies_Adam_2c_(run_time={run_time}_sec)2.svg',
format="svg")

max_val = np.max(history.history['val_sparse_categorical_accuracy'])
print(f'max val accuracy: {max_val}')
```

cnn.py (used in Section 4.2.4)

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import numpy as np
import time

from simple_data_generator import SimpleDataGenerator

inputs = keras.Input(shape=(100, 1,), name="digits")
conv1 = layers.Conv1D(32, 8, activation="relu", name="Conv1")(inputs)

flatten = layers.Flatten()(conv1)  #pool2)
x = layers.Dense(200, activation="relu", name="dense_1")(flatten)
drop1 = layers.Dropout(0.25)(x)

outputs = layers.Dense(2, activation="softmax", name="predictions")(drop1)

model = keras.Model(inputs=inputs, outputs=outputs)

class1_params = {"game_size": 100,
        "data_type": "burst_error",  # "binomial" or "burst_error"
        "label": 0,
                "bin_p": 0.5,
                "bur_p": 0.5,  # p to good state
                "bur_r": 0.5,  # p to bad state
                "bur_k": 0.75,  # p of 1 in good state
                "bur_h": 0.5,  # p of 1 in bad state
                "seed": None}
class2_params = {"game_size": 100,
                "data_type": "burst_error",
                "label": 1,
                "bin_p": 0.65,
                "bur_p": 0.2,
                "bur_r": 0.2,
                "bur_k": 0.75,
                "bur_h": 0.5,
                "seed": None}

prms = [class1_params, class2_params]
data_gen = SimpleDataGenerator(prms)
bur_errors = data_gen.get_burst_error_rates()
print([1 - x for x in bur_errors])

x_train, y_train = data_gen.generate(100000)
x_test, y_test = data_gen.generate(10000)

y_train = y_train.astype("float32")
y_test = y_test.astype("float32")

model.compile(
    optimizer=keras.optimizers.Adam(),  # Optimizer
    # Loss function to minimize
    loss=keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)

start_time = time.time()
print("Fit model on training data")
history = model.fit(
    x_train,
    y_train,
    batch_size=1000,
    epochs=20,
    # We pass some validation for monitoring validation loss and metrics at the end of each epoch
    validation_data=(x_test, y_test),
    shuffle=True
)
run_time = int(time.time() - start_time)

fig = plt.figure()
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss Value')
plt.title('CNN: Cross Entropy Loss')
plt.show()
# fig.savefig(f'CNN_Cross_Entropy_Loss_Adam_2c_(run_time={run_time}_sec).svg', format="svg")

fig = plt.figure()
ax = plt.axes()
plt.plot(np.array(history.history['sparse_categorical_accuracy'])*100, label='Training Accuracy')
plt.plot(np.array(history.history['val_sparse_categorical_accuracy'])*100, label='Validation Accuracy')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy %')
ax.yaxis.set_major_formatter(mtick.PercentFormatter())
plt.title('CNN: Training & Validation Accuracies')
plt.show()
#   fig.savefig(f'CNN_Training_&_Validation_Accuracies_Adam_2c_(run_time={run_time}_sec).svg',
format="svg")

max_val = np.max(history.history['val_sparse_categorical_accuracy'])
print(f'max val accuracy: {max_val}')
```

qlearning.py (used in Section 5.2.3.1)
NOTE: training here takes a long time. It takes about 3 days on an average desktop computer to get a score average > 18.5

```
import os
import time
import random
import pickle
import datetime
import numpy as np


def getHand():
    deck = list(range(2, 15)) * 4
    hand = random.sample(deck, 5)
    return hand


def getHash(state):
    hash = str(state[0])
    for bit in state[1:]:
        hash += "0" + str(bit)
    return hash


class warcatGame():
    def __init__(self):
        self.player_hand = []
        self.bot_hand = []
    def reset(self):

        self.player_hand = sorted(getHand(), reverse=True)
        self.bot_hand = sorted(getHand(), reverse=True)
        return self.player_hand[:]

    def step1(self, action):
        diff = self.player_hand.pop(action) - self.bot_hand[0]
        score = np.sign(diff) * 13 - diff
        next_state = np.append(self.player_hand[:], self.bot_hand[0])
        return next_state, score

    def step2(self, action):
        diff = self.player_hand.pop(action) - self.bot_hand[1]
        score = np.sign(diff) * 13 - diff
        next_state = np.append(self.player_hand[:], self.bot_hand[1])
        return next_state, score

    def step3(self, action):
        diff = self.player_hand.pop(action) - self.bot_hand[2]
        score = np.sign(diff) * 13 - diff
        next_state = np.append(self.player_hand[:], self.bot_hand[2])
        return next_state, score

    def step4(self, action):
        diff = self.player_hand.pop(action) - self.bot_hand[3]
        score = np.sign(diff) * 13 - diff
        next_state = np.append(self.player_hand[:], self.bot_hand[3])
        return next_state, score

    def step5(self):
        diff = self.player_hand[0] - self.bot_hand[4]
        score = np.sign(diff) * 13 - diff
        return score

    def render(self):
        return 0


class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.gamma = discount_factor
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.learning_rate = learning_rate
        self.q_table = dict()
    def act(self, state):
        hash = getHash(state)
        if hash in self.q_table.keys():
            act_values = self.q_table[hash]
        else:
            act_values = np.zeros(self.action_size)
            self.q_table[hash] = act_values
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        return np.argmax(act_values)
    def predict(self, state):
        hash = getHash(state)
        if hash in self.q_table.keys():
            act_values = self.q_table[hash]
            return act_values[:]
        else:
            self.q_table[hash] = np.zeros(self.action_size)
            return np.zeros(self.action_size)[:]

    def load(self, name):
        with open(name, 'rb') as handle:
            self.q_table = pickle.load(handle)

    def save(self, name):
        with open(name, 'wb') as handle:
            pickle.dump(self.q_table, handle, protocol=pickle.HIGHEST_PROTOCOL)


def train(iterations, save_model=False):
    if save_model:
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)
    scoreAverage = []
    for e in range(iterations):
        state1 = env.reset()
        action1 = agent1.act(state1)
        next_state1, score1 = env.step1(action1)
        reward1 = constant_reword * score1
        hash1 = getHash(state1)
        old_qs1 = agent1.q_table[hash1]
        next_q = old_qs1[action1] + agent1.learning_rate * (reward1 + (agent1.gamma * 
np.amax(agent2.predict(next_state1))) - old_qs1[action1])
        old_qs1[action1] = next_q
        agent1.q_table[hash1] = old_qs1

        state2 = next_state1
        action2 = agent2.act(state2)
        next_state2, score2 = env.step2(action2)
        reward2 = constant_reword * score2
        hash2 = getHash(state2)
        old_qs2 = agent2.q_table[hash2]
        next_q = old_qs2[action2] + agent2.learning_rate * (reward2 + (agent2.gamma *
np.amax(agent3.predict(next_state2))) - old_qs2[action2])
        old_qs2[action2] = next_q
        agent2.q_table[hash2] = old_qs2

        state3 = next_state2
        action3 = agent3.act(state3)
        next_state3, score3 = env.step3(action3)
        reward3 = constant_reword * score3
        hash3 = getHash(state3)
        old_qs3 = agent3.q_table[hash3]
        next_q = old_qs3[action3] + agent3.learning_rate * (reward3 + (agent3.gamma *
np.amax(agent4.predict(next_state3))) - old_qs3[action3])
        old_qs3[action3] = next_q
        agent3.q_table[hash3] = old_qs3

        state4 = next_state3
        action4 = agent4.act(state4)
        next_state4, score4 = env.step4(action4)
        reward4 = constant_reword * score4
        hash4 = getHash(state4)
        old_qs4 = agent4.q_table[hash4]

        score5 = env.step5()
        totScore = score1 + score2 + score3 + score4 + score5
        next_q = old_qs4[action4] + agent4.learning_rate * (totScore - old_qs4[action4])
        old_qs4[action4] = next_q
        agent4.q_table[hash4] = old_qs4

        totalScore = score1 + score2 + score3 + score4 + score5
        scoreAverage.append(totalScore)
        if len(scoreAverage) > 500:
            scoreAverage.pop(0)

        if agent1.epsilon > agent1.epsilon_min:
            agent1.epsilon *= agent1.epsilon_decay
        if agent2.epsilon > agent2.epsilon_min:
            agent2.epsilon *= agent2.epsilon_decay
        if agent3.epsilon > agent3.epsilon_min:
            agent3.epsilon *= agent3.epsilon_decay
        if agent4.epsilon > agent4.epsilon_min:
            agent4.epsilon *= agent4.epsilon_decay

        if e % print_every_n == 0:
            print("episode: {}/{}, moving average: {:.2f}, score: {}, e: {:.2}".format(e,
n_episodes,np.mean(scoreAverage), totalScore, agent1.epsilon))

        if save_model:
            if e % save_every_n == 0:
                agent1.save(output_dir + "agent1_weights_" + '{:04d}'.format(e) + ".hdf5")
            agent2.save(output_dir + "agent2_weights_" + '{:04d}'.format(e) + ".hdf5")
            agent3.save(output_dir + "agent3_weights_" + '{:04d}'.format(e) + ".hdf5")
            agent4.save(output_dir + "agent4_weights_" + '{:04d}'.format(e) + ".hdf5")


def validation_test(iterations, load_model = False, model_dir = '', iteration_number=0):
    if load_model:
        agent1.load(model_dir + "agent1_weights_" + '{:04d}'.format(iteration_number) + ".hdf5")
        agent2.load(model_dir + "agent2_weights_" + '{:04d}'.format(iteration_number) + ".hdf5")
        agent3.load(model_dir + "agent3_weights_" + '{:04d}'.format(iteration_number) + ".hdf5")
        agent4.load(model_dir + "agent4_weights_" + '{:04d}'.format(iteration_number) + ".hdf5")

    def getScore(pCard, bCard):
        diff = pCard - bCard
        return np.sign(diff) * 13 - diff

    alltScore = []
    for k in range(iterations):
        p = sorted(getHand(), reverse=True)
        b = sorted(getHand(), reverse=True)
        tScore = 0
        act1 = np.argmax(agent1.predict(p))
        tScore += getScore(p[act1], b[0])

        p = np.delete(p, act1)
        p = np.append(p, b[0])
        act2 = np.argmax(agent2.predict(p))
        tScore += getScore(p[act2], b[1])

        p = np.delete(p, act2)
        p = np.delete(p, len(p) - 1)
        p = np.append(p, b[1])
        act3 = np.argmax(agent3.predict(p))
        tScore += getScore(p[act3], b[2])

        p = np.delete(p, act3)
        p = np.delete(p, len(p) - 1)
        p = np.append(p, b[2])
        act4 = np.argmax(agent4.predict(p))
        tScore += getScore(p[act4], b[3])

        p = np.delete(p, act4)
        tScore += getScore(p[0], b[4])

        alltScore.append(tScore)
    print("Accuracy=  " + '{:04f}'.format(np.mean(alltScore)) + "    from testing "+str(iterations)+" 
iterations    by loading: " + str(model_dir + "agentx_weights_" + '{:04d}'.format(iteration_number) + 
".hdf5 " + str(iteration_number)))

env = warcatGame()
state_size1 = 5
```

```
state_size2 = 5
state_size3 = 4
state_size4 = 3
state_size5 = 2

action_size1 = 5
action_size2 = 4
action_size3 = 3
action_size4 = 2

#parameters
discount_factor = 1.0
epsilon = 1.0 #exploration
epsilon_decay = 0.999999999
epsilon_min = 0.01
learning_rate = 0.00075
constant_reword = 1

batch_size = 32
n_episodes = 500000001
n_episodes = 50000001
print_every_n = 10000
save_every_n = 1000000
current_time = datetime.datetime.now().strftime('%Y-%m-%d_%H;%M;%S')
model_dir = "saved models/"
if not os.path.exists(model_dir):
    os.makedirs(model_dir)

output_dir = model_dir + "/time=({})_iterations={}/".format(current_time, n_episodes)

agent1 = DQNAgent(state_size1, action_size1)
agent2 = DQNAgent(state_size2, action_size2)
agent3 = DQNAgent(state_size3, action_size3)
agent4 = DQNAgent(state_size4, action_size4)

# uncomment to load a trained model at some iteration number
# iteration_number = 500000000
# load_model_dir = model_dir + "time=(2021-06-06_15;32;40)_iterations=500000001/"
# agent1.load(load_model_dir + "agent1_weights_" + '{:04d}'.format(iteration_number) + ".hdf5")
# agent2.load(load_model_dir + "agent2_weights_" + '{:04d}'.format(iteration_number) + ".hdf5")
# agent3.load(load_model_dir + "agent3_weights_" + '{:04d}'.format(iteration_number) + ".hdf5")
# agent4.load(load_model_dir + "agent4_weights_" + '{:04d}'.format(iteration_number) + ".hdf5")

train(n_episodes, save_model=True)

# uncomment and correct the time below to match the one in the name of the trained model and to do a
validation test
#validation_test(100000, load_model=True, model_dir=model_dir + 'time=(2021-06-
06_15;32;40)_iterations=500000001/', iteration_number=500000000)
```