# Abstract

A key design principle of modern trading card games (TCGs) is diversity between cards used by the players. This principle leads to different ways of playing within a single TCG; different cards promote different lines of achieving the game’s goal – which is usually to increase or reduce a game value to achieve a win. Artificial intelligences in digital TCGs are usually designed to be good at playing the simplest decks, or those that follow the “normal” or intended game plan. However, some decks will try to outpace others, slow others down, or circumvent the regular play patterns altogether. At this point AI can perform much worse. Part of the issue is that decks in TCGs are created from large pools of cards, and so having a strict set of rules to play each possible deck is not feasible.

# Table of contents

# Introduction

This project is about developing an Artificial Intelligence algorithm for a complex card game, Magic: The Gathering (MtG). MtG is a popular trading card game that pits two players against each other in the roles of duelling wizards. The players summon creatures and cast spells in an attempt to reduce the opposing players life total down to 0. Cards are used to represent these spells in the game, and players can make their own choice of which cards to include in their decks.

Forge [5] is an open source project to provide a digital environment to play MtG. The goal of the project is to develop an AI that can be released for use alongside Forge and its current AI, which is lacking in certain areas. In this way, the project is for the Forge community and user-base.

There is currently a lack of competitive AI opponents for MtG players, and whilst the game is primarily played between two human players, it would be useful for players to have an AI opponent available to train against and benchmark themselves against. One of the more difficult aspects of designing AI for trading card games (TCGs) is that there is a high variation in deck contents, and the game plan of these decks. Having an AI that can identify a deck’s speed or focus within the game can help the Forge AI to make decisions that better suit a deck’s game plan.

This report will outline the background and methodologies used to develop the project, as well as detailing design choices that impacted the project implementation.

## Aim

The aim of this project is to provide an AI that can learn to classify Magic: The Gathering decks such that it provides a tool for the Forge AI to change its decisions based on the speed and plan of the deck it is given.

## Objectives

1. To produce a deck classifying AI capable of taking a deck list from the Forge client and returning a classification for that deck.
2. The AI should be able to learn from training data what attributes of a card are common across different classifications of decks.
3. The AI should be able to take deck lists that include new or unseen cards and return a concise output classification.
4. The AI should be able to be linked into the Forge client to impact the Forge AI’s decision-making process.

# Motivation

There has been a recent surge in studies looking at developing learning AIs for games, both digital and physical. Some good examples of this are AlphaGo [1] for the board game Go, as well as Deep Blue [2] for Chess; an example of a digital game AI is OpenAI Five [3] for the video game Dota 2. These AI are developed with different pursuits in mind, but most look into training AI for real world application, using game contexts as substitutes due to their complex states and decision-making requirements.

Magic: The Gathering (MtG) is a strategic card game between two players, who take the roles of duelling wizards, using a variety of spells represented by cards to reduce the other’s life total to zero. Unlike traditional card games, MtG uses custom made cards in place of traditional playing cards. The game also makes use of imperfect information, in a similar vein to Bridge, where the exact contents of an opponent’s hand are usually unknown. Each player uses a custom deck of cards made up from a selection of thousands of cards, which makes it hard to guess a new opponent’s cards until they are played or otherwise seen.

I find MtG an interesting game due to its variety in cards, and the variety of playstyles this wide selection provides, with many cards altering how the game is played significantly. This variety also makes it an interesting case for developing AI to play MtG, as a set strategy will not always provide similar results against opposing strategies. Another facet to MtG is due to its imperfect information, there is commonly no computable optimal move, as unknown cards might make a usually optimal move sub-optimal, and vice versa.

Recent studies have shown that MtG is Turing Complete [4]; this is not directly relevant to developing an AI for the game, but it does mean that MtG is more computationally complex [4] than Chess and Go, and speculated to be the most computationally complex game in literature [4]. The cards required to induce a Turing machine state in the game will likely not be used in this project, due to complexity issues, as well as being an unrealistic scenario. However, the ability to prove MtG is Turing Complete shows the robustness of the game’s logic. A lot of game AIs use various types of decision trees in combination with neural networks including AlphaGo, Deep Blue, and others [6][8], and there has also been research into using Monte Carlo Tree Search for card selection in order to play MtG [7], albeit with some heavy limits to the game’s complexity.

Originally, I had wanted to design an AI based on learning methods to play MtG in its entirety. However, after a couple of months of research and preliminary design, this plan was deemed infeasible for this project. This was due to the complexity of MtG, as well as the lack of resources and time that an individual can provide.

The new inspiration for the project: Is it possible to write an AI for classifying decks in MtG? The game has multiple viable strategies and options for deck building, and each value certain decisions higher than others. I hope to provide a foundation in the Forge client to manipulate the Forge AI’s decision making based on the deck it is playing.

This is the approach I will likely be taking, using tree search to determine what neural network is appropriate for the current game state. The most challenging aspect of this project would be defining appropriate inputs and structure for the neural network(s).

# Related Work

There have been numerous studies into teaching AI to play traditional board games, AlphaGo [1] and Deep Blue [2] being more successful examples of this. There have also been studies specific to writing an AI to play MtG [7], but these usually involved heavy restrictions on what could be played, severely limiting the game’s complexity. This project aimed to create an AI that can play within a less restricted version of the game but switched focus once it became apparent that the original goal was infeasible. Instead the project focused on classifying the decks used to play the game to help make more appropriate decisions. There have not been any high-profile deck classification projects for MtG, and so the project was mostly built from nothing.

The utility of knowing what kind of deck a player is playing has many advantages. For example, knowing that you are playing a fast deck helps when deciding how aggressive to play, and conversely knowing if the opponent will be able to stop you quickly enough can help decide how to prepare your board-state for their defences. Another important factor in competitive play is when two similar decks face each other: identifying who is the aggressor is vital for a player to win. This concept is discussed further in the article “Who’s the Beatdown?” [9].

The company behind MtG have several of their own digital clients for playing the game, the two most notable being MtG Online [11]and MtG Arena [12], however developing for these platforms is not a plausible route. MtG Online does not currently have an AI built-in, and so play is entirely between human players. MtG Arena has primitive AI for teaching the game to new players, but not a competitive AI for playing against, and so gameplay is again mainly between human players. Both games feature anti-cheat measures, which prevent “botting”, or automating inputs, and so developing an AI for these platforms is not plausible. However, the lack of in-depth AI in both games indicate that this is a hard problem which the game developers believe to be too difficult to be worth implementing.

Two alternative open-source digital clients include Cockatrice [13] and Xmage [14]. Cockatrice lacks a rules engine for game and relies on the players communicating their intentions correctly. Thus, an AI algorithm would not be able to know what to do, as all actions must be performed manually. Xmage is a viable alternative to Forge, being both open source and having a full rules engine. However, Xmage is used more often for online games between human players, whereas Forge is strictly against AI opponents, so it makes more sense to develop for the latter.

Forge [5] is an open source project to provide a digital environment to play MtG. It currently features a basic AI with a function set to get game information. This AI is not that proficient, nor complex, and can be frustrating or dull to play against if a player is sufficiently skilled. An AI that is more challenging would be more engaging to advanced players and is something I think worth developing for this reason, and this is what I initially decided to do for my project.

Tensorflow [10] is a python library written to aid in the creation and use of neural networks. I used this library as writing my own neural network framework would be cumbersome and difficult. Tensorflow also makes use of Nvidia CUDA technology, which can utilise GPU processing power to speed up the machine learning process. Tensorflow is used for a wide variety of machine learning implementations, namely image classification and reinforcement learning algorithms.

Deep Learning 4 Java is another library for using neural networks. I only used part of DL4J’s tools, to load the network model generated in python into the Forge client using Java. I could have used DL4J for the entire project, but Tensorflow is a more lightweight solution and can be run and modified relatively quickly.

## Description of a Magic Card

Since this project involves the properties of a magic card, this section will explain briefly the various aspects of a magic card. Despite there being multiple types of cards, for the purposes of this project, all cards are processed the same way. This process still accounts for the type of card each card is, so no details are lost.

Each magic card has several important features:

* Name: each card has a unique name which the game uses to identify it.
* Mana cost: each card has a cost which a player must pay using an in-game mechanic called mana. In “fair” magic mana is usually accrued at a rate of 1 per turn, and so there is an expectancy to be able to cast a 4-mana card on turn 4. Mana costs contain both “coloured” and “generic” mana, with coloured mana being more restrictive on a card.
* Converted mana cost: this is the total amount of mana required to spend on a card, regardless of colour requirements.
* Type: each card has a type, which dictates how the card can be played. There are currently ~ 370 unique card types; a card can be almost any combination of types.
* Power and Toughness: these values are mostly integers, and are used for in game combat between cards, dictating how much damage a card can deal and receive in combat.
* Keywords: these words give cards extra abilities, varying gameplay from the base rules.

These described features will be used to process cards and deck lists for the neural network. There are other features, such as art or number within a set, but these are not relevant for the project goal and thus left out.

Some cards have alternate casting costs, which can turn a 6-mana card into something that can be played on an earlier turn, but these are harder to process generically, and will not be specifically targeted by the network.

## Description of a Magic Deck

Magic decks generally fall into one of three main archetypes: “Aggro”, “Midrange”, and “Control”. These archetypes describe how the deck aims to win a game of MtG.

Aggro decks aim to play cheap, fast spells that create efficient creatures or damaging effects, and end the game before the opponent has a chance to establish their own spells in defence. Midrange decks aim to play medium costed creatures that are able to weather an Aggro deck’s early aggression and using creatures or spells that generate extra efficiency in the form of abilities or recurring effects, create a board-state that gradually overwhelms the opponent. Control decks generally aim to slow the pace of the game down, removing or block cheaper creatures, before playing one or two large creatures that are difficult for the opponent to remove.

Each archetype has preferences for card features, such as mana costs or card types, that help identify what archetype a deck belongs to. For example, a deck with mostly 1 or 2 costed creatures, with a few direct damage spells, is likely to be an Aggro deck. A deck with a range of mana costs, and a mix of removal spells and creature spells, is likely to be a Midrange deck. A Control deck is likely to feature “counterspells” (cards that prevent an opponent’s spell having any effect), a few high cost creatures, and several card drawing spells. With these examples in mind, it should be possible to create an AI that is able to correctly identify from a deck’s features, what archetype a deck is.

# Description of the Work

The result of the project is a neural network which can accurately predict the speed of a given MtG deck, which is accessed by the Forge client upon AI deck selection. It modifies existing Forge weighting variables to cause the AI to value different decisions higher based upon the deck’s rated speed.

In addition to this there is a GUI which is usable to format Forge client deck lists, retrieve information about each card from both online and within the Forge client, and manually rate decks. This GUI was developed to provide training and test data for the deck classifier, as there was previously no data available.

# Methodology

There are many methods used within AI research, and a wide spread of them are present in game playing research. Here I outline some of the possible technologies usable for the project and discuss why they were plausible options. Many of these options were considered primarily for the initial project goal of playing MtG, instead of the final goal of classifying decks within the game.

## Convolutional Networks

Convolutional networks using an operation called convolution that takes parts of an image as groups and performs some operator function – the convolution – to them. These outputs are either a value or set of values that can be used to identify and classify features within an image.

For some games, such as Go [6], or Chess, convolutional networks can be used to recognise board patterns and compare these to standard board states, which the AI can use to make a decision. This image/pattern recognition approach can work well for board games that have a rigid board structure, e.g. a square grid, as patterns are likely to appear and repeat themselves between games. As well as this, the position and orientation are meaningful in these games, and so this provides more focused patterns or board states to be recognised.

A good example of using convolutions is in [6], where they detail the way that common Go play patterns impacted the shapes of convolutions. They attempt to combat “Ladders” by using diagonal convolutions to recognise if a Ladder is escapable or not, by looking for an additional piece that can break the Ladder.

MtG does not have a rigid board structure; it does not matter what order cards are positioned on the table, nor in the hand. This means that an image-based algorithm would not suit a game playing AI, as it would not be able to read any meaningful data from the images.

## Branching Decision Trees

Decision trees are used frequently [1][8] to determine what state a game is in, so that an AI can make choices specific to that scenario. This helps in games where actions can change in priority as the game progresses; in Real Time Strategy (RTS) video games, it is common to have resource gathering phases before combat phases, and so an AI would need to be able to determine if it should aim for increasing resources or using them to fight.

An example of the difficulties of using Tree Search algorithms, such as Monte Carlo tree search, can be seen in a study on writing an AI for the RTS game StarCraft [15], where the amount of complexity within the game is shown. Monte Carlo tree search is also used by the AlphaGo [1] study, where they iterate through multiple solutions using a function to maximise an “action value” and requires a huge 48 CPUs and 8 GPUs to process 40 search threads. This kind of resource is not available for this project, so tree search is not a practical option.

For MtG, decision trees seem like a sensible idea: the game features a resource system, which can be used to create stronger combat units. The game also plays differently depending on a player’s strategy, and so identifying that strategy could be important to maximising success. An issue with decision trees, though, is that they can be slow to iterate through possible game states. This is because with each additional possible game state, the number of possible paths increases exponentially, and the processing power required is too large, as evidenced by the AlphaGo requirements above. This issue is particularly pertinent within MtG, as the board states can vary greatly between games, and strategies can appear very similar to start, and only become apparent as different several turns into the game.

For the reviewed goal of classifying decks, decision trees could be used by sorting each card within a deck and generating a new branch for each combination. Again, this is infeasible due to complexity issues, with a combination set of 20000 cards leading to approximately 1.268e176 permutations.

## Neural Networks

Neural networks are very common within AI development [1][2][6], and can be used in conjunction with other techniques, such as convolutional matrices, to either classify or provide outputs for a given set of inputs. These networks are good at being trained for a particular task, through reinforcement learning, where the network attempts to perform its purpose, e.g. classification, and by comparing its answers to training data of correct answers, can self-correct its decision variables to improve itself on the next run. This self-correction algorithm is known as back-propagation [16].

Back-propagation takes the current weights in the neural network and reinforces the weights in order to align the output value(s) with the training data target values. This weight change usually effects a gradient descent approach, in order to encourage gradual change to prevent outliers from breaking a model.

For this project, a neural network could be useful, as reinforcement learning is a desired part of the final AI, and a neural network is an efficient way to do this, as the reinforcement can be automated with enough training data. However, neural networks require specific inputs, and so deciding what inputs these should be can be difficult. In the case of MtG, it is not especially apparent what all the inputs should be, due to the large quantity of possible inputs available.

For the initial goal, this included player information, such as life totals or cards in hand, as well as information about certain cards, properties of cards on the board, and cards in decks. For the revised goal, this included a more concise set of each card’s properties. Inputting all this information into a network might result in a network that is difficult to train, and not very good at playing or learning to play. To ensure that I understood the network and did not over convolute the problem, I chose to select certain inputs, instead of inputting all possible inputs.

# Design

## The Original Design

The original design was as follows: a card selection AI, that uses attributes about cards in hand, as well as limited board state information, to choose which card to play when available.

This would be achieved by writing a neural network in python using the TensorFlow library. This network would take the inputs from the Forge client and return a value to the client mid-game. The program would also save the scores and inputs matching those scores to potentially be used as training data in the future. The forge client also would make use of two evaluative functions to provide additional inputs to the neural network.

The Forge client is very complex and features a wide set of classes to deal with the MtG rules engine. Fortunately, the code base is well written and thus working out an injection point to put my project’s behaviour was made easier. Unfortunately, the code base is not well documented so working out what certain functions and classes did was slower than anticipated. Part of the reason the project is designed around a card score-based selection is down to how the Forge client currently has AI for its games.

## Decision Inputs

After consideration of different ways to read the game state, I had decided that a card selection method would be the best way forward. This was because ultimately, actions are performed by playing cards, and so rating which card is best to play is the most sensible way to choose an action. In order to determine which card to play, certain inputs are needed. These include what kind of card each card is, as well as attributes about the card, like what effects it has when played, or if it will have effects later in the game, e.g. a creature card can provide benefits to a player over multiple turns, until it dies. This card selection AI would have been a neural network, take inputs that describe a card, as well as limited board state information, and output a score for the card. This score would be generated based on how effective the card would be to play in the current moment, with higher scores reflecting a better play. Since training data is unavailable for the initial tries, any successful games would have their scores saved as good data, and lost games would have a chance to have their scores saved as well, in case good decisions were made despite losing the game.

In order to input a card’s attributes, I would have had an algorithm that uses the Forge client’s card database API, which contains all the relevant attributes and information, and saves it into a state that can be input into the neural network. The algorithm would give a value between 0 and 1 for each possible attribute, and this would reflect what qualities or effects a card has, and how much value each attribute provides for the card.

Another important piece of information is whether a player is on the offensive or defensive. Certain strategies lend themselves better to playing offensively, and so certain cards will be better in an offensive position. To determine this, I would have written an algorithm that read the game state values such as life totals, and identified how much of an offensive position the AI player is in. This would have then fed into the neural network and contribute to scoring each card.

## Revised Design

After realising the original project was infeasible, the design was remodelled to accommodate the revised project goal. Since the main difference between the new and old projects was the data being used, the biggest redesign was in that area. Due to lowering the scope of the AI, the data required went from a subset of a large set of optional data inputs, to a clear concise set of inputs: the cards’ properties. The issue of no training or test data carries across, and so this data will be generated manually by aggregating multiple volunteers’ ratings into a mean rating. This will help reduce the uncertainty that a single opinion would generate.

# Implementation

## Original Implementation

The implementation was to be written in part in java, and in part in python. The reasoning behind this is because the Forge client is written in java, and so writing the AI in java is practically required. The neural network part will be written in python using the TensorFlow [10] library. This is because I do not have the knowledge or time to effectively write my own neural network structure for this project. To interface between the Forge client and the neural network I was to use the TensorFlow java library.

Figure 1The information flow of the project. Data is moved from the Forge client to the evaluative functions, which return a card score.

I had so far been using Git to manage the development between two machines, only one of which can use the Nvidia CUDA library to perform machine learning tasks quickly. The other machine is a laptop and so being able to develop on either is important to maintain smooth development.

It became apparent after January that the original project was infeasible due to computational restrictions, and that the focus of the project was switched to the Deck Classifier.

# Revised Project Implementation – Deck Classification Network

The revised design is split into three parts: the GUI for manually assessing and rating decks to provide test data, the neural network classifier which processes decks and parses them to output a trained model, and the integration of the model into the Forge client.

## Classification GUI

The classification GUI was made in order to generate training data for the neural network. The GUI presents a visual display of a deck from the Forge client and allows the user to select a speed on a scale from 1 to 10. These scores are then saved to a text file, ready to be used by the neural network.

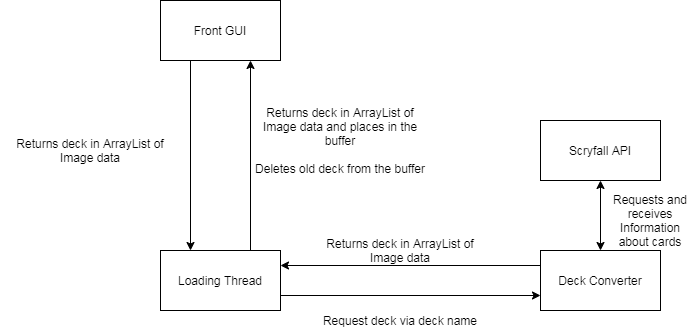
The classification GUI tool has two distinct parts: the front-end GUI display, and the back-end card data processor. The front-end uses Java’s Swing library and uses a separate thread to run the back-end process to load and unload deck data.

Figure 2 Simplified flow of data within the Deck Classifier

### Front End GUI

The front-end GUI displays all the cards within a deck in a table format. When it starts, it loads the first set of decks into memory, then displays the GUI for the user. Whenever the user submits a rating for a deck by pressing a button, the back-end thread unloads the previous deck and loads the next one into the buffer. This allows memory to be used efficiently whilst still providing a buffer of decks loaded, should the user progress through the currently loaded decks quickly. The GUI features a set of radio buttons that allows a user to easily select a value for the deck’s speed, and since the buttons are grouped as a radio group, selecting any given value will deselect any previously selected value. When either the “Next” or “Finish” buttons are pressed, the program logs the currently selected speed value and saves it to a list. When the program is closed, either by clicking “Finish” or the window’s “X” button, the list is saved to a text format with the deck’s name and its speed value paired, for easier processing later, both manually and automated.

### Back End Data Processor

The data processor itself has two parts to it: the ListConverter class and the various data structures to hold the cards’ information. The ListConverter class contains functions to read a deck list from Forge – or otherwise specified folder – and convert it into a list. This list is then parsed, and the card names are looked up using the online Scryfall MtG card database API [18] with data and images taken and saved to a list of CardData objects. These objects contain all the relevant information about a card, and this list is what is handed back to the classification GUI to use and display. The image data is used primarily for the GUI classification, whereas the rest of the data is stored locally for use by the neural network later. This helped to eliminate the issue of double processing all the information for each card and deck.

## Neural Network Classifier AI

The network AI used to classify decks is written in python using the TensorFlow libraries and Keras libraries for TensorFlow. It includes two parts, a deck parsing script, and the neural network scripts.

### Deck Loader

The deck loader loads a given deck from its respective JSON file and converts the information within to a set of values, which are returned to the neural network script. These values are in three distinct formats: Unique Identifiers (UIDs), integer values, and binary values. The integer values are used in places where the conversion is simple, such as a cards’ converted mana cost, power, and toughness. The binary values represent all possible properties a card could have, such as if it is a certain colour, if it has certain keywords, or if it is a certain card type. UIDs are used for names and mana costs, since they are not easy quantifiable, and each encountered entry is recorded in a dictionary, and its index in the dictionary is used as its value. The end output of this pre-processing is a 60 row, 522 column 2d array which is then passed to the neural network script.

### Network Script

The neural network script has functions for both creating a new model and loading a model from a saved location. The new model function creates the model by generating a set of new models via k-fold cross validation and picks the one with the best accuracy on the test data. The model features 5 layers: an input layer with 60 nodes, three hidden layers with 40,30, and 20 nodes respectively, and an output layer with 5 nodes, giving an output of 5 confidence values representing which speed class the classify believes the deck to be. The original spread of speeds was 1 to 10, but these were condensed to 1 to 5, as there is not much meaningful distinction between decks on a wider scale, and the low amount of training data means that edge classes will not have a valid set of training cases.

## Forge Integration

The Forge integration uses the Deep Learning 4 Java libraries to load the Keras model and runs pre-processing on the selected deck. This pre-processing is the same as performed by the neural network, but in Java for the Forge client. The model outputs its array of confidence values for each class (speed 1 to 5). The strongest of each class is taken as the deck’s speed. This speed is then used to modify decision biases / weights that the Forge AI uses to make decisions and actions. This set of processes is run when a game in Forge is started.

# Evaluation and External Aspects

To evaluate the performance of the classification network, I had originally planned to aggregate multiple volunteers’ ratings of a large set of deck lists from the Forge client. I built a GUI tool in order to collect this data. Unfortunately, before data collection could commence, the COVID-19 global pandemic prevented data from being gathered. Instead, I generated training data manually which created a data set which is less reliable than preferred, as it is subjective data. The GUI displayed a few stability issues to start, and so to ensure that this would not impede data generation, I added a feature to resume deck rating from a given deck. This helped reduce efficiency lost if the client were to crash during data generation. Since the GUI was unfortunately not used by others, these stability issues were only minor, and did not impact development. If the GUI were to be used in the future for more data generation, I would ensure these stability issues were fixed beforehand.

The neural network created was run on a 10-fold cross validation process, and the best model created from this process had a 63% accuracy on unseen data, which is not a high accuracy, but is still a good rate. The lack of data for the extreme classes, 1 and 5, is likely to have caused issues when attempting to evaluate these decks.

The alternative to reduce this problem could be to switch the classes range from 1 to 3, representing 3 styles of decks: “Aggro”, “Midrange”, and “Control”, with each being slower than the last. Originally, I had not wanted to use these classifications as there are some decks which fall into multiple of the categories, as well as speed discrepancies between decks within the same class. However, I recreated the data set, this time classifying each deck on its archetype rather than its relative speed. This was easier, and likely created more reliable data as the distinctions between archetypes are easier to judge. Due to this change, the range of classes was reduced again to 3, one for each archetype. Unfortunately switching the training data alone did not solve the low accuracy, and additional changes had to be made to the network model.

One of the first changes was to increase the number of nodes within each layer, as the processing power was available to be utilised. The layers were increased to 300,250,180,100, and 3 for each respective layer. The final layer node count was reduced to 3 to match the new number of outputs, and the other node counts increased. This increased the accuracy within each fold during cross-validation to around 80-95%, but validation accuracy was still low, at around 50-70%. This is a sign of overfitting, where the network trains too much to the provided training data set.

There are a few methods available to combat overfitting, the easiest of which is increasing the data set’s size. Unfortunately, I was unable to do this, due to the time constraints of the project, and the lack of availability of such data. One method that I was able to use, however, is called regularisation, and is used to reduce overfitting by adding a constant error into the network. Regularisation adds a constant offset to prevent the network from developing an overly complex weight set.

After adding and changing the last details to the model, it had an approximate 76% accuracy rate, which was better, but not perfect. I decided to look at which classes it was struggling with. When I looked at the data, it was apparent that the issue was a lack of data; the Control class “3” was severely under-represented in the training data, being only 38 of the 269 decks listed. The other two classes, Aggro and Midrange have a population of 137 and 114 respectively. The unfortunate lack of Control deck data means that to improve the network, more data would have to be obtained, and the network retrained. However, ignoring the Control decks due to their low representation shows a network with 80% accuracy on both Aggro and Midrange decks.

I then decided to dismiss Control as a class within the training data. Since there was not enough data for it, it only hindered the network’s accuracy. The network now only predicted Aggro or Midrange decks at an 86% accuracy, correctly predicting 122/127 Aggro decks and 76/104 Midrange decks. A likely reason that the network is more able to correct predict Aggro decks than Midrange decks is due to the variation within the archetypes. Midrange allows for more varied cards to be played and a wider spread of mana costs, whereas Aggro decks demand the most efficient, cheaper creatures. Another factor is that cards used in Aggro will overlap with those in Midrange, but not vice versa, and so the network will be biased towards guessing Aggro.

By removing Control, the validity of the classifier is reduced, but with extra data the network could be retrained to account for it. In addition, there are several other common archetypes in MtG: Tempo, Combo, and Prison. Tempo is similar to Aggro, seeking to quickly beat down the opponent, but often trades direct damage and efficiency for ways to clear the board for small creatures to continually attack. Combo decks seek to obtain 2 or more cards that interact with each other, usually to create an infinite loop of game actions, often winning the game for the Combo player immediately. Prison decks aim to create board states such that the opponent is severely limited in what actions they can take, achieved by removing the cards in the opponent’s hand, or creating costs for otherwise free game actions. These archetypes are harder to define than the original 3 used in the project, and for that reason were not initially included. However, given enough sample deck data, these archetypes could also be accounted for.

# Summary and Reflections

Overall, the project was a success, despite mediocre classification performance by the network. After having to switch the project’s focus, the new goal was still achieved in good time, showing that a stronger deck classifier could be produced given more time and data.

Looking back, I think I would have switched the classes used to instead represent the 3 archetypal decks, Aggro, Midrange, and Control, as the definitions as to what makes a deck one of these 3 archetypes is much more concise and reliable than an arbitrary speed rating. This would help prevent any issues with subjective data collection.

# Bibliography

[1] Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. (2016). Mastering the game of Go with deep Neural Networks and tree search. *Nature*, 529(7587), pp.484-489.

[2] Campbell, M., Hoane, A. and Hsu, F. (2002). Deep Blue. *Artificial Intelligence*, 134(1-2), pp.57-83.

[3] OpenAI. (2019). *OpenAI Five*. [online] Available at: <https://openai.com/blog/openai-five/> [Accessed 9 Oct. 2019].

[4] Churchill, A., Biderman, S. and Herrick, A. (2019). Magic the Gathering is Turing Complete. [online] Available at: <https://arxiv.org/abs/1904.09828> [Accessed 9 Oct. 2019].

[5] Slightlymagic.net. (2019). *Forge - Slightly Magic*. [online] Available at:  <https://www.slightlymagic.net/wiki/Forge> [Accessed 9 Oct. 2019].

[6] Barratt, J. and Pan, C. (2017). Playing Go without Game Tree Search Using Convolutional Neural Networks. [online] Available at: <http://cs231n.stanford.edu/reports/2017/pdfs/603.pdf> [Accessed 16 Oct. 2019].

[7] Ward, C. and Cowling, P. (2009). Monte Carlo Search Applied to Card Selection in Magic: The Gathering. [online] Available at: [https://web.archive.org/web/20160528074031/http://scim.brad.ac.uk/staff/pdf/picowlin/CIG2009.pdf](https://web.archive.org/web/20160528074031/http:/scim.brad.ac.uk/staff/pdf/picowlin/CIG2009.pdf) [Accessed 16 Oct. 2019].

[8] Silver, D. and Huang, A. (2012). Mastering the Game of Go with Deep Neural Networks and Tree Search. [online] Available at: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Publications_files/unformatted_final_mastering_go.pdf> [Accessed 16 Oct. 2019].

[9] Flores. M 1999, “Who’s the Beatdown?” [https://web.archive.org/web/20190329100828/http://starcitygames.com/magic/fundamentals/3692\_Whos\_The\_Beatdown.html](https://web.archive.org/web/20190329100828/http:/starcitygames.com/magic/fundamentals/3692_Whos_The_Beatdown.html) [Accessed 1 December]

[10] TensorFlow <https://www.tensorflow.org/> [Accessed 1 December]

[11] Magic: The Gathering Online <https://magic.wizards.com/en/mtgo> [Accessed 4 December]

[12] Magic: The Gathering Arena <https://magic.wizards.com/en/mtgarena> [Accessed 4 December]

[13] Cockatrice <https://cockatrice.github.io/> [Accessed 4 December]

[14] Xmage <http://xmage.de/> [Accessed 4 December]

[15] Uriarte, A. and Ontan˜on´, S. (2019). Improving Monte Carlo Tree Search Policies in StarCraft via Probabilistic Models Learned from Replay Data. [online] Available at: <https://www.aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/viewFile/13984/13601> [Accessed 4 Dec. 2019].

[16] Rumelhart, D., Hinton, G. and Williams, R. (2019). *Learning representations by back-propagating errors*.

[17] Nvidia CUDA <https://developer.nvidia.com/cuda-zone> [Accessed 5 December]

[18] Scryfall API https://scryfall.com/docs/api [Accessed 20 January]

# Appendices