2020

Cheat Detection using Machine Learning within Counter-Strike: Global Offensive

Harry Dunham
The College of Wooster, hdunham20@wooster.edu

Cheat Detection using Machine Learning within *Counter-Strike: Global Offensive*

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Department of Mathematical and Computational Sciences
at The College of Wooster

by
Harry Dunham
The College of Wooster
2020

Advised by:

Dr. Nathan Fox (Computer Science and Mathematics)

THE COLLEGE OF
WOOSTER

Abstract

Deep learning is becoming a steadfast means of solving complex problems that do not have a single concrete or simple solution. One complex problem that fits this description and that has also begun to appear at the forefront of society is cheating, specifically within video games. Therefore, this paper presents a means of developing a deep learning framework that successfully identifies cheaters within the video game *CounterStrike: Global Offensive*. This approach yields predictive accuracy metrics that range between 80-90% depending on the exact neural network architecture that is employed. This approach is easily scalable and applicable to all types of games due to this project's basic design philosophy and approach.

This work is dedicated to my father who unfortunately passed away and did not get to see this paper to its conclusion.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

Cheating is a social construct that has been routinely discouraged and regulated throughout human history. Cheating can be thought of as a means of gaining advantages, goods, or other worldly possessions in an unfair and anti-competitive manner.

1.1 Cheating in Video Games

Cheating exists within video games for a multitude of reasons. For example, it can be used to simply win a game when a player otherwise cannot, to have fun (that is to say, some players derive enjoyment through the use of cheats), and many more depending on the game and the context. However, cheating is typically a very damaging entity that can hurt the success of a game. Therefore, this lends itself towards the creation of anti-cheat methods within video games that actively prevent and prohibit players from being able to gain an unfair advantage over other players. Currently for the majority of games within the first-person shooter genre, cheating is very apparent. Therefore, there is a large need and desire to develop better methods that successfully detect and combat cheats.

Cheats can be boiled down to two basic categories: knowledge-based cheats and game assistance cheats. The former category focuses on players using cheat

methods to gain otherwise unobtainable information/knowledge, while the latter encompasses players who utilize automated programs/software in order to ameliorate their ability to play the game. Cheating methods typically involve the manipulation of a game's memory; this is due to the fact that all the relevant information about the game state (i.e. where other players are, what items they have, etc.) is stored within it. Therefore, the development of new anti-cheat methods that circumvent a reliance on memory monitoring is essential.

1.2 Counter-Strike

Counter-Strike is a first-person shooter video game, which was originally released in 1999. *First-person shooters* (FPS) are a genre of video games whereby players are pitted against each other and incentivized to eliminate opponents in order to win. First-person shooters typically allow players to use weapons such as guns in order to eliminate the opposition. Due to this gameplay dynamic, most of the cheats that are prevalent within the game revolve around the augmentation of player accuracy, which allows players to have an unfair advantage in eliminating enemy players. However, cheats do not stop there. Additionally, there are cheats that allow players to access information that is otherwise not available to the player, such as the location or exact health of enemies. The former cheat is typically called an "AimBot," since the task of aiming is giving to a "robot" or application rather than being performed by the player-entity. The latter cheat is typically realized through "WallHacks," whereby players are able to visually see the enemy players through walls. There are many more cheats within *Counter-Strike*, but these two are the most renowned and prevalent within the game.

1.3 Why is Cheat-Detection a Suitable Task for Machine Learning?

Cheating is typically compared to a game of cat and mouse. Developers of anti-cheating software can fix exploits and other methods hackers/cheaters use to violate the rules of the game and subsequently push out a game update. However, eventually hackers/cheaters either find new exploits or find exploits within the patches that the developers released. This continues ad infinitum, and eventually it becomes less and less profitable for the game-developers to spend time, money, and effort fixing issues and exploits that would eventually be circumvented.

This basic dynamic between cheaters and anti-cheat developers leads to the research and creation of new anti-cheat methods that do not have these same weaknesses. One of these involves the use of machine learning algorithms to correctly identify when player-entities are cheating. This circumvents and avoids all the vulnerabilities that are associated with having games stored in memory, since we can instead evaluate players in real-time to determine whether they are cheating. However, it raises concerns such as how to properly determine whether players are or are not cheating, how to mitigate false-positives and false-negatives, and similar concepts.

1.4 Outline of Remaining Chapters

This paper begins by outlining the fundamentals of the game of *Counter-Strike*, which includes a high-level explanation of how the game is played and the objectives that players have during the game. Additionally, we outline the two core interdependent objects that this project's data hinges upon, those being ticks and subsequently demo files (which are large compilations of ticks).

Next, in order to be able to understand the types of cheats that are being targeted and studied under this project's scope, it is essential to provide a baseline definition of cheating. Therefore, Chapter 3 focuses on the means with which cheating presents itself within *Counter-Strike*. This requires the introduction of cheat "families," which are two broad and distinct subgroups of cheats that are commonly found in *Counter-Strike* as well as the majority of other first-person shooter video games. Finally, the theory and implementation of the most common cheats within *Counter-Strike* are presented in order to provide a theoretical background. This allows our solutions to various cheats to be grounded on the theoretical underpinnings that these cheats found themselves upon.

Following our discussion on cheating, we discuss deep learning on a theoretical basis as well as the implementation that this project uses in order to tackle cheaters within *Counter-Strike*. Therefore, we begin by outlining the various types of learning that are present within machine learning and the biological influences that permeate within the field of machine learning. Next, this paper covers recurrent neural networks and why they are the most logical network structure for the task of cheat detection within *Counter-Strike*.

The next chapter is this project's use of graphics cards (GPUs) for neural network training, which includes various differences between their functionality to that of central processing units (CPUs). Additionally, this chapter includes the various implementation specifics for this project.

Chapter 6 covers the data component of this project and its associated manipulation and implementation. This includes discussion of the means by which data is sourced, the method in which data is converted into meaningful and useable representations of the data, and the crafting of metrics and features that allows the network to efficiently extrapolate what cheating is and how it presents itself within *Counter-Strike*.

Finally, the following chapter is the results that this project attains. This includes a multitude of different testing methods which all yield similar results. However, some of these testing methods prove to be far more logical and efficient than others in ways that are distinct from simply predictive accuracy.

The conclusion follows our results chapter and focuses on the future work to be done within this field and this project. Additionally this chapter serves to provide insight into various errors within the scope of this project as well as potential improvements that could have ameliorated the results that this project obtains.

CHAPTER 2

Counter-Strike

Counter-Strike is a series of *first-person shooters* (FPS), whose first iteration, *Counter-Strike*, was originally released in November 1999 (see Figure 2.1). The current game in the series is *Counter-Strike: Global Offensive*; however, we refer to *Counter-Strike: Global Offensive* as simply *Counter-Strike*. The game revolves around two basic teams, attackers and defenders (labelled in-game as *terrorists* and *counter-terrorists* respectively). The attackers typically have two objectives, which are delineated and separated by game modes: destroy an objective or rescue a non-player entity (labelled in-game as *defuse* and *hostage-rescue* respectively). These objectives are determined by the game mode the user is playing. However, elimination of the other team is often a plausible objective as well.

The game mode that is most popular and used competitively is defuse; therefore, this project only sources data from this game-mode, as it is the most abundant. At the beginning of a match, players are randomly assigned to either the attacking team or defending team. The match is separated into rounds, whereby the attacking team has a limited amount of time to attempt to succeed in fulfilling one of their objectives. The defending team attempts to prevent them from accomplishing their goals. Should the time run out for that current round, the defending team is awarded the round-win, since they have successfully deterred the attacking team. Otherwise, should one team attain one of their objectives, they are instead awarded the round-win. The goal of this game mode is to be the first team to reach 16

Figure 2.1: In-game rendering of *Counter-Strike*

round-wins. Additionally, after 15 rounds are played, the teams switch roles, i.e. the attacking team gets to defend for the next 15 rounds, while the defending team gets to attack for the next 15 rounds. This means that the game is inherently symmetric.

One small caveat and difference between *Valve Software*'s matchmaking system and other third-party systems is that the former does not allow for overtime, which is an extension/addition of more rounds to decide who wins the match. Overtime occurs when both teams reach 15 round-wins over the course of the match. Overtime only differs from regular match-play in two ways. First, rather than being a best of 30-round match, it is best of 6 rounds. This means that the first team to reach 4 round wins subsequently wins the entire game. The second difference is that players start overtime with the maximum amount of in-game cash, which is typically earned through completing objectives within each round during the regulation period of the game. Cash can be used to buy items, which include Kevlar armor to reduce the amount of damage a player sustains, weapons, and utility items (e.g. smoke grenades, flash grenades, etc.). The most powerful items tend to be the most expensive items within the game; therefore, overtime presents the opportunity for players to buy and equip the best items. Finally, overtime still

adheres to the inherent symmetric nature of regular match play by forcing players to switch from the attacking team to the defending team and vice versa after the first three rounds are played. This means that it is still possible for the game to be tied after overtime, which, when this type of situation occurs, results in another subsequent overtime. However, this system only typically occurs in third-party matchmaking systems and large scale tournaments; therefore, when a match within *Valve Software's* matchmaking system ends in a tie, the match is terminated and no winner is determined through overtime.

2.1 Ticks

A tick is a packet of information that is sent from the client to the server and includes things such as player position, where the player is looking, what team the player is currently on, etc. *Counter-Strike* has approximately 400 different game events, whereby each game event is a structured packet of data that is sent to the server. For example, the event `player_hurt` occurs when a player entity loses health points within the game (see Listing 2.1). The event includes the in-game name of the hurt client, the in-game name of the player who inflicted the damage, the remaining amount of health for the hurt player, etc.

Within *Counter-Strike*, there are two possible tick-rates, which is the number of times ticks are exchanged between clients and the server per second. The two rates are 64-tick and 128-tick. For this project, it does not matter which type of tick-rate the server uses, since the neural network considers each tick individually and normalizes values across the entire demo. However, this does mean that two matches of the same duration (e.g. 30 minute long games) that differ in tick-rate have a large disparity in the amount of retrievable data from these matches (i.e. large differences in the number of ticks). This means that 128-tick matches are

objectively superior to those of 64-tick, in terms of the total feasible amount of retrievable information.

However, it is important to note that the majority of matches that contain cheaters are 64-tick games. This is due to the basic design infrastructure of *Counter-Strike: Global Offensive's* online matchmaking system. All online matches played on *Valve* servers are 64-tick, while tournament matches, third-party matchmaking systems, and community servers tend to use 128-tick servers. Additionally, there is typically less incentive for cheaters to play on the latter and higher-tick servers due to their lack of rewards and better anti-cheat methods; therefore, there is a higher propensity to see players cheating on the lower tick-rate servers. This is something that is inherently unavoidable due to the aforementioned reasons, but it should still be acknowledged.

Ticks are the smallest incremental data point that this project uses. Ticks contain important game-state meta-information that is used in the construction of an input vector that is passed to the neural network. Ticks are the main reason why we can avoid focusing on an anti-cheat implementation that focuses purely on how memory is accessed and altered on a client's computer, since we can instead analyze what every player is currently doing in the game at any given point. This is analogous to analyzing a video by going through every frame within the video.

2.2 Demo Files

Demo files are the means with which *Counter-Strike* saves matches for viewing within the game. They are the compilation of all the ticks within the game. These files come in an unreadable format, but in 2014, *Valve* released a *GitHub* repository containing the means of creating a simple application that converts `.dem` files to plain text documents [29].

Demo files can be viewed and obtained in many different ways. For example, every competitive match of *Counter-Strike: Global Offensive* is displayed upon one of the game's various dashboard elements with the option to save and download the `.dem` file and re-watch the match. Additionally, there are many online forums and services that save various demos. For example, all of the professional matches that are played (online, on LAN, etc.) are stored on the match page on a site like `hltv.org`.

```
player_hurt
{
 userid: s1mple (id:23)
  position: 283.655396, -213.244171, -0.403509
  vecs: -168.293594, -104.493011, 0.000000
  tickbase: 109280
  facing: pitch:0.071411, yaw:101.420288
  entity: 8
  team: T
 attacker: ZywOo (id:21)
  position: 75.123627, 455.732880, 0.031250
  vecs: 231.874176, -30.520855, 0.000000
  tickbase: 109277
  facing: pitch:0.291138, yaw:287.671509
  entity: 6
  team: CT
 health: 85
 armor: 92
 weapon: hkp2000
 dmg_health: 15
 dmg_armor: 7
 hitgroup: 4
}
```

Listing 2.1: A `player_hurt` event featuring the game's most renowned professional players, s1mple and ZywOo

CHAPTER 3

Cheating

Cheating is the cornerstone that this project hinges upon. Therefore, it is essential to thoroughly define cheating both in an abstract sense and within *Counter-Strike* itself.

3.1 Cheating Theory

In order to understand cheating, we must first give a thorough definition that provides the framework or foundation for later discourse. Thus we will define it as follows:

Any behaviour that a player uses to gain an advantage over his peer players or achieve a target in an online game is cheating if, according to the game rules or at the discretion of the game operator (i.e. the game service provider, who is not necessarily the developer of the game), the advantage or the target is one that he is not supposed to have achieved [21].

Cheating within the current personal computer era stems from the basic fact that games are stored within memory on players' personal computers. Since users have free access to their own personal hardware, there is a multitude of different ways for users to access and manipulate this memory. This lends itself to the creation

of applications that utilize the memory related to the game in ways that both the developers never intended and that the typical user never experiences.

Therefore, while our definition extends to methods of cheating that fall outside the realm of the game world (i.e. collusion between teams or players, match-fixing, etc.), we narrow the scope of that definition to cheating methods that rely on exploitation of the game-state and its associated information. This hinges upon the fact that those other aforementioned methods of cheating tend not to present themselves within the game-state itself. For example, if a tournament fixes matches in order to give one team favorable matches to win the tournament, the players within each match do not exude any unnatural behavior or cheating-like behavior, since they still have to necessarily play *Counter-Strike* in their typical fashion to win the match. This inherently differs from a player who, for example, can see all the enemy players through walls, since they change their play-style around the fact that they have an atypical advantage.

3.1.1 Cheating "Families"

Based off our aforementioned narrowed scope in terms of cheating within *Counter-Strike*, we can now define the basic "families" of cheats which exist within *Counter-Strike*: mechanic assistance and knowledge assistance. However, it is also important to note that *Counter-Strikes*'s Overwatch system breaks down cheating into three basic categories: "Aiming Assistance," "Vision Assistance," and "External Assistance." Overwatch is an in-game system where users are given the opportunity to review matches of players who are suspected of cheating. After the user finishes reviewing the match, they are required to submit a verdict, which includes options to select whether the cheater in question utilized any of the previously mentioned cheating categories. These categories are not as precise and tend to be unwieldy due to the fact that "External Assistance" is a catch-all category for all cheats that do not

involve a player's aim or vision. Therefore, we define and use more stringent categories that avoid the potential nuances and pitfalls that are associated with those of *Counter-Strike*.

3.1.1.1 Mechanic Assistance

"*Game mechanics* are methods invoked by agents, designed for interaction with the game state" [24], where agents are simply players interacting with the game. Mechanics can also be thought of as something analogous to different skills that sports rely on. For example, one of the main mechanics that is present within first-person shooters is obviously shooting. Players are given weapons that they can aim with and fire. We can view this as something akin to how quarterbacks within American football have the ability to throw the football. Therefore, we can think of something like *aim-assistance* as some sort of application that aids the player in shooting more accurately and effectively. This would be analogous to a quarterback having some sort of technology that ameliorates his ability to accurately throw passes to his receivers.

Therefore, we can define *mechanic assistance* as a family of cheats whereby players use outside applications or software to improve the way in which they interact with the game world. In general, this is a very large group of cheats within games due to the fact that players are typically given many different ways to interact with the game world. For *Counter-Strike*, the two main methods of interaction involve moving/movement, and aiming. However, while those are the main two and are typically involved with most cheats, there exist many different inputs that a player can use to affect the game environment (e.g. destroy in-game objects, drop or pick up in-game objects, open doors, etc.).

3.1.1.2 Knowledge Assistance

The next family of cheating involves gaining an advantage and assistance over other players with knowledge or information pertaining to the game state. Typically, the most desirable information within *Counter-Strike* involves the locations of enemy players on the map. This stems from the fact that one of the main objectives of the game is to eliminate the opposition; therefore, knowing their exact coordinates grants players massive advantages. This can take many forms and appearances. Whether it be by rendering the enemy player models so that they are visible through all objects (i.e. walls) within the game environment or by rendering the enemy players to the in-game map of the game environment (see Figure 3.1), it is easily the most prevalent form of knowledge assistance.

Figure 3.1: The radar or mini-map of *Counter-Strike*, with red dots being enemy players and blue dots being teammates

3.2 *Counter-Strike* Specifics

While *Counter-Strike* is certainly not the first FPS game to exist, it has existed long enough for all the hegemonic hacks, which are present within nearly every

first-person shooter, to be created by outsider programmers/hackers. These are "AimBot" and "WallHack," which can be thought of as the representatives of the two distinct families of cheats.

3.2.1 Aim Assistance - "AimBot"

"AimBot," which is an abbreviation for "Aiming Robot," is an application that automatically corrects where a player-entity is aiming so that their crosshair falls directly over top of an enemy player. The crosshair can be thought of as a visual representation and visual aid that is rendered on the screen to help players determine the exact pixel location of where they are looking within the game environment. This cheating method requires a lot of math based upon trigonometry in order to calculate these values.

First, *Counter-Strike* player entities are multi-dimensional entities, where the principal values are their position in terms of an x, y, and z component, and their pitch and yaw, which are the angles about which they are looking (see Figure 3.2). Pitch, yaw, and roll are used within aeronautics and describe the axes with which a plane rotate about.

Pitch can be thought of as the wing-to-wing axis, and rotations about this axis alter the position of the plane's nose in terms of an upward or downward direction. Yaw is the axis defined through a vertical intersection of the aircraft, and rotations about this axis alter the position of the plane's nose in terms of side-to-side directions (i.e. left and right). Finally, roll can be thought of as the nose-to-tail axis and rotations about this axis result in barrel-roll type rotations, i.e. rotations that alter to orientation of the cabin and crew of the plane. Roll is not an entity or value that players can freely alter, since that would allow players to rotate themselves into upside-down orientations and other strange positions.

All of these values, plus the location of the positions of the enemy players,

Figure 3.2: Changing view orientation from looking at the ground to looking at the ceiling (i.e. initial pitch and yaw as 89° and 0° respectively, and final values as −89° and 0°).

are all the requisite components needed to calculate the proper pitch and yaw in order to aim at the enemy/enemies. Therefore, we construct two vectors, one for the player whose aim we are correcting, $\overrightarrow{p} = \langle x_p, y_p, z_p \rangle$ and the second for an enemy player, $\overrightarrow{e} = \langle x_e, y_e, z_e \rangle$. From here we construct a difference vector, which is simply the difference between each component of the two players position vectors: $\overrightarrow{d_\Delta} = \langle x_p, y_p, z_p \rangle - \langle x_e, y_e, z_e \rangle = \langle x_p - x_e, y_p - y_e, z_p - z_e \rangle$. Therefore, the total distance between the two player entities is simply the magnitude of $\overrightarrow{d_\Delta}$, i.e. $\|\overrightarrow{d_\Delta}\| = \sqrt{(x_p - x_e)^2 + (y_p - y_e)^2 + (z_p - z_e)^2}$. This is the first crucial value needed in order to derive the pitch and yaw.

One essential idea is that this cheating method roots itself within basic trigonometry by constructing/identifying various lengths and metrics associated with the

two players and constructing two triangles, which are then used to derive the necessary angles. Therefore, with the distance between the two players, and the difference between the heights of the players (i.e. $d_{\Delta z} = z_p - z_e$, which happens to unconventionally be defined about the z-axis within *Counter-Strike*), we can use $\arcsin\left(\frac{d_{\Delta z}}{\|\vec{d_\Delta}\|}\right) = pitch_p$ (see Figure 3.3).

This same method can be applied in order to derive the necessary yaw, but instead we form a triangle based upon the differences in the x and y displacements between the two players, and additionally use $\arctan\left(\frac{d_{\Delta y}}{d_{\Delta x}}\right) = yaw_p$.

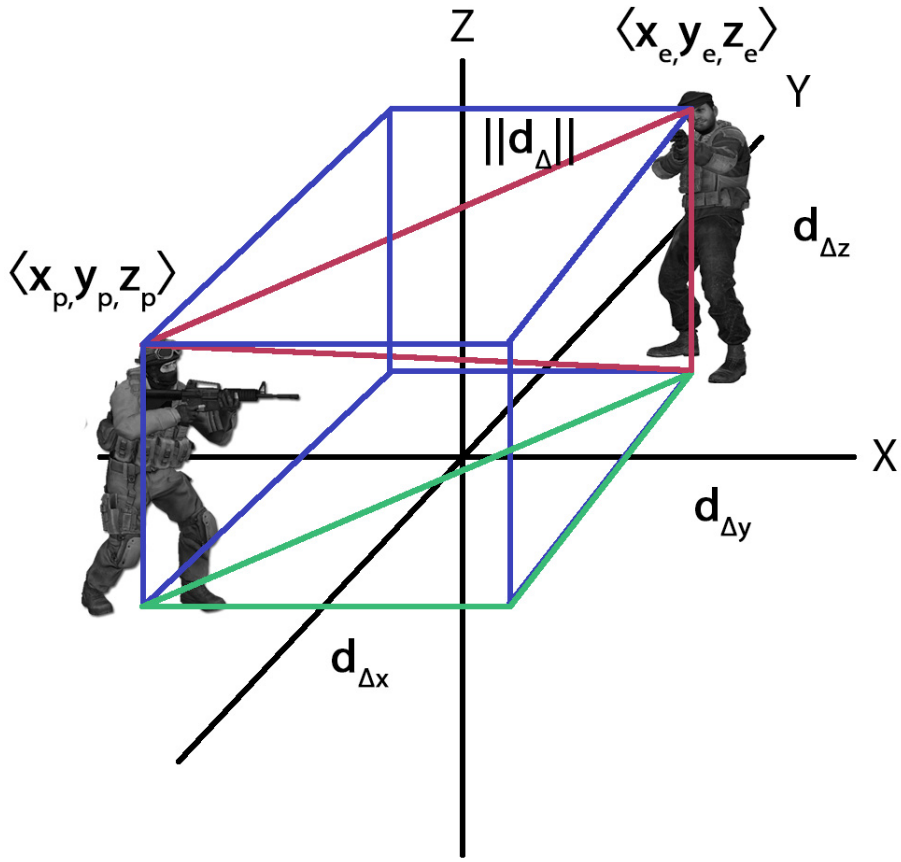

Figure 3.3: Visualizing correcting a player's aim to an enemy player-model

The last step for these calculations is properly bounding the values and converting them to either degrees or retaining them as radians, which depends upon the storage

and implementation of angles within the game. This is one of the most crucial steps in this entire process, due to the nature of anti-cheat systems, since if we improperly calculate the angle such that it lies outside the bounds of possible angles that a player can have, then this will immediately trigger the anti-cheat system and will result in a ban. For example, while we might typically envision turning ourselves in the real world with bounds from 0 degrees (i.e. where we are currently looking) all the way up to 360, which is an entire rotation that lands us back to where we were originally looking, *Counter-Strike* bounds its yaw (i.e. side-to-side angles) between $-180 \leq yaw_p \leq 180$. Additionally, the game does not allow players to look completely up or completely down, which restricts pitch (i.e. down-to-up angles) $-89 \leq pitch_p \leq 89$. Therefore, after checking that our calculated and corrected pitch and yaw values fall within the bounds of the game, we can rewrite the local memory that is responsible for storing our players pitch and yaw values.

One important thing to note about this method is that it is inherently obvious and quite easy to visually distinguish between players employing this method and players who are not. This is due to the fact that the adjustment of pitch and yaw values results in a linear adjustment (i.e. a straight line) between where the player was originally aiming and where the enemy player currently is. It is obvious that when players are naturally using a mouse they tend to exhibit more entropic adjustments between those two aiming locations. Therefore, many of the more advanced and prevalent "AimBot" cheats attempt to add entropy to the adjustment to make it appear more human-like.

3.2.2 Knowledge Assistance - "WallHack"

"WallHacks" typically take on a plethora of different shapes and forms, but they share the same core idea of accessing all of the enemy player's locations and using that information to render some visual guide to the cheater in order for them to

see where the enemies are through walls. The simplest of these forms is a simple bounding box that encapsulates the enemy's player model, while the most complex methods render complete outlines of the enemy player models with associated meta-data. For example, one of the more prevalent complex methods of "WallHacking" involves color-coding the outlines so that the cheater can determine how far away enemies are based upon their outline color, or based upon how much health the enemy has left (see Figure 3.4).

Figure 3.4: Wall hack with color-coded outlines [1].

3.2.3 Mechanic Assistance - "B-Hop"

"B-Hop," which stands for "Bunny-Hop," is a unique game mechanic that stems from the physics within *Counter-Strike* (i.e. the *Source Engine*). In order to understand a "Bunny-Hop," we must first define strafing, which is the act of a player-entity moving sideways in relation to where they are looking. The next important idea to introduce is friction, which, when we consider *Counter-Strike*, is only ever applied to

[1] This source is available from the author upon request; it is redacted from this paper due to concerns about raising awareness of illicit or otherwise dishonest content.

player-entities that are physically touching the ground. This means that players that are currently airborne either through jumping or falling do not experience any form of friction, which can simply be conceived of as a slowing force within the game.

The final core idea that "Bunny-Hopping" founds itself upon is the basic fundamentals of how movement operates within *Counter-Strike*. Players have two main methods of interacting with the game-environment, which are moving and looking around. When a player inputs movement through a keyboard press, their player model accelerates in a certain direction within the game world. Likewise, when a player presses two keys that correspond to movement within the game, their player model moves at an angle that bisects the two angles, or no movement should they be in opposite directions (see Figure 3.5).

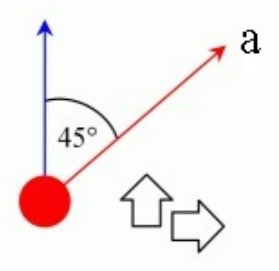

Figure 3.5: Pressing the forward and right directional keys [13]

It is important to note that this acceleration value is a constant, and the maximum speed with which a player-entity can move within *Counter-Strike* is 250 `units` per second, where `units` is simply an arbitrarily defined distance within the game environment. Next, the crux of "Bunny-hopping" is within how this bound is calculated by the game's engine. First, this speed bound is only calculated in the direction that the player-entity is currently travelling. For example, if a player is moving at 300 `units` per second in the forward direction and presses the forward key, this key press has no effect since the player is already moving at a speed that exceeds the maximum. Therefore, friction, our slowing force, is applied to reduce

the player's speed to a value that is less than the maximum value if the player is touching the ground. However, the player can still freely augment their speed with the other directional keys (i.e. within our example, the left and right keys could be pressed to augment the speed in those directions). This stems from the fact that *Counter-Strike*'s engine only considers players to have a single direction of movement, which we denote as $\overrightarrow{V}$. This is achieved through vector projections, whereby given a players current velocity, denoted $\overrightarrow{V_c}$, and the direction the player currently wants to move in, denoted $\overrightarrow{V_w}$, we (meaning the game engine) construct the projection of $\overrightarrow{V_c}$ onto $\overrightarrow{V_w}$, which we can denote $V_p = \|\overrightarrow{V_c}\| \cdot \cos\theta$ (see Figure 3.6) [13]. This is all in relation to the orientation of the player, which is denoted as $\overrightarrow{V_o}$.

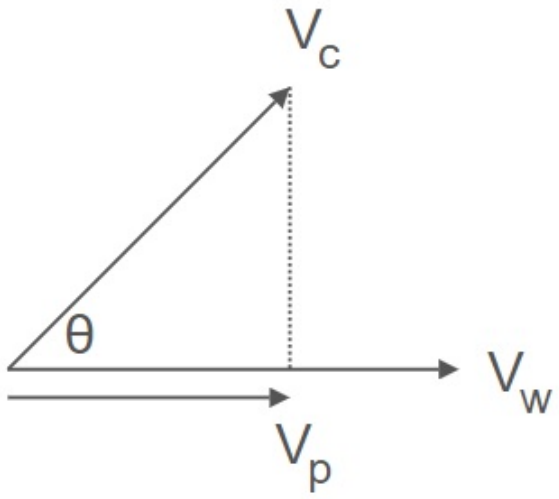

Figure 3.6: Projecting V_c onto V_w to form V_p

Finally, we denote the direction with which a player accelerates as $\overrightarrow{V_a}$, with an associated magnitude $\|\overrightarrow{V_a}\| = a$. This can be simply conceived of as a scalar. The acceleration is then added to the magnitude of the projection, i.e. $speed = \|\overrightarrow{V_p}\| + a$, which yields another scalar, which is then truncated or reduced if it exceeds the maximum speed. However, this convention now allows for $\overrightarrow{V_c}$ to exceed the

maximum velocity, since the magnitude of its projection is checked rather than its own magnitude. This stems from the earlier convention of players only having a single direction of movement.

Therefore, a "B-Hop" is a culmination and combination of all of these aforementioned ideas. We can circumvent the cap that is placed on a player's movement by jumping in the air and strafing (i.e. clicking the side directional keys) and continually changing the direction of movement that our player is moving in so that it coincides with the direction key that we are pressing. This means that gradually looking left while pressing the left directional key augments the magnitude of $\overrightarrow{V_c}$ while circumventing the maximum speed cap within that specific direction, since the magnitude of $\overrightarrow{V_p}$ is capped rather than $\overrightarrow{V_c}$ (see Figure 3.7). The final caveat to this is that the source engine does not immediately apply friction to players when they land on the ground. Instead players are given a short period of time (specifically a single `tick`, which is between 0.0078125 or 0.015625 seconds depending on the tick-rate of the server) before friction is applied. This allows players to input another jump in order to avoid getting slowed down, and they subsequently gain speeds that far exceed the maximum value imposed by the game engine. This behavior results in its name "bunny-hopping," since players who execute this technique are constantly zigzagging and jumping around.

In terms of implementation, the prototypical method that cheaters employ is artificially writing to the game's memory to input a jump that coincides within the small timeframe before friction is applied in order to easily achieve high speeds. However, as *Counter-Strike* has evolved, the difficulty of "B-Hopping" has increased dramatically, and at the game's current release/state it is currently impossible to consistently achieve this technique without cheats. This stems from the timeframe to input a player's next jump being too small to properly time. This means that players who achieve consistent "B-Hops" are typically cheating, since it is entirely

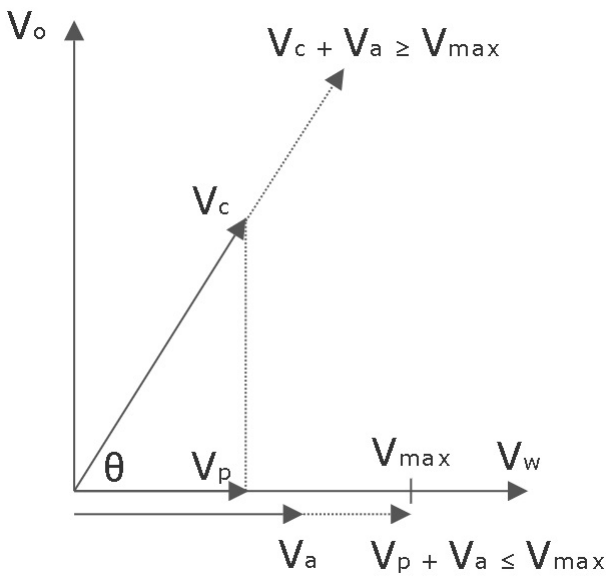

Figure 3.7: Difference between the unbounded current velocity vector and the bounded projection velocity vector.

luck-based. However, it is important to note that it is not impossible to "Bunny-hop", it is just humanly impossible to consistently perform. In other words, players can sometimes still string together multiple "Bunny-hops," but it just relies entirely on luck.

3.2.4 Forms of Cheating

While there are many different types of cheats which aid players in their ability to perform and win within *Counter-Strike*, all earlier discourse ignores how players actually cheat within the game. For example, if we were viewing a game, what does cheating look like, and how does it present itself?

The first inherent problem with cheat detection through analysis of in-game player behavior is that, for the most part, a lot of the more prevalent cheats within *Counter-Strike* are not immediately obvious and/or visually apparent. Instead, and particularly when a player is actively attempting to hide their usage of cheats, the

determination of whether a player is cheating relies upon an understanding of the game and how normal players would play given a specific set of scenarios within the game. For example, one of the most suspicious types of behaviors that can be seen with cheaters is how they always appear to be lucky, i.e. the player always has their crosshair in the correct position, the player always predicts abnormal behavior by opponents, etc. For players of every level (professionals included), this type of behavior does not present itself for the entire duration of a game; instead it can happen in spurts or bouts. However, it is obvious that abnormal luck is not an immediate indication of whether a player is cheating. It is inherently conceivable for a player to be extremely lucky and not be cheating. Therefore, it is essential to admit and acknowledge the fact that when we consider this gradient of cheating, we can only source and use data points that fall within the binary extrema of the game. That is to say, while it may appear that a player is exuding behavior that is indicative of cheating, unless there is evidence beyond a reasonable doubt, we cannot include that data for the sake of preventing false-positives and false-negatives.

It might seem that we have devised an outlook and understanding of cheating that includes only the most obvious forms of cheating, since we can only consider the most visually blatant examples of cheating and non-cheating. However, this problem can be circumvented through the creation of our own cheating methods to obtain otherwise unobtainable data. Additionally, we can source data from willing cheaters without succumbing to the potential of mislabeling data.

The second problem is that cheating is not inherently a binary behavior that players exude; instead it can be conceived of as a gradient, whereby players can cheat for some percentage of the game and subsequently play cleanly (i.e. without cheats) for the remainder of the game. However, to further complicate this matter, there are also degrees of cheating, which stems from the fact that multiple cheats can be used in union or parallel to further ameliorate the players' ability to win and

complete game objectives. For example, a player could cheat for a single round (i.e. potentially only $\frac{1}{30}$ of the entire game) with only some form of knowledge assistance. The cheater then continues to play cleanly from that point forward. However, later in the game the cheater's team is about to lose, which results in them deciding to activate multiple different cheats to ensure that their team wins the match. This example is meant to elucidate the sheer complexity that is associated with cheating in first-person shooters.

The final problem that is associated with cheating is the fact that we, as outside viewers of the game, do not see the visual components of various cheats. Cheats alter the local players' rendering of the game-state, meaning that we do not visually see the game alterations that cheaters perform on their local memory when we observe matches or view `demo` files. For example, when cheaters render the enemy's player-models through walls so that they always know their opponents' locations, we do not see the same renderings within `demo` files or when spectating the game (see Figure 3.8).

Figure 3.8: Outsider's view of a cheater versus the cheater's local game render.

Overall, cheating is a very complex and multi-faceted problem that roots itself within the ability to manipulate memory, which allows players to bend the physics of the game engine to their will and artificially access information that is

otherwise unobtainable. While this may seem to hinder our ability to properly source and combat cheating, we can avoid most of the aforementioned problems by breaking down the problem into smaller pieces that are more easily palatable and understandable.

CHAPTER 4

Deep Learning

Machine learning is a subfield of artificial intelligence that attempts to develop algorithms that mimic human abilities and capacities. Deep learning is, in-turn, a subfield of machine learning, and can be conceived of as "a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations" [7]. The addition of the word "deep" within deep learning lends itself to confusion since it implies a form of deeper understanding from machine learning algorithms. However, this is simply not the case and is merely meant to highlight the fact that deep learning emphasizes neural network constructions that utilize multiple layers of nodes, where neural networks are simply the most common form of these aforementioned representations. We can formally define neural networks as directed acyclic graphs that are composed of layers of nodes or neurons that take in an input (i.e. data) to produce an output (typically some form of prediction). For example, Chollet constructs a basic network to predict the handwritten number that is depicted within an image, which uses four successive layers before producing a final output value (see Figure 4.1).

Machine learning also encompasses multiple different learning strategies and outlooks which each have their own benefits and consequences. The typical deciding factor between which type of learning method to employ is based upon the dataset and how it is organized and labeled. The three types of learning are unsupervised, semi-supervised, and supervised. While the concept of "supervising" a neural

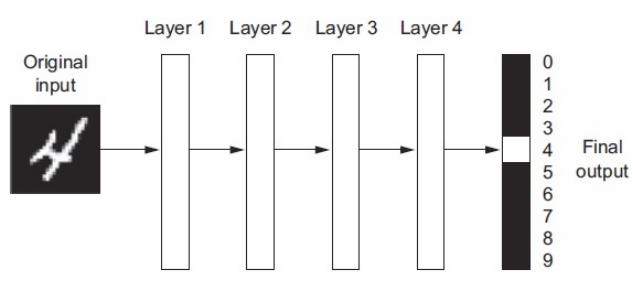

Figure 4.1: Abstract network construction for predicting handwritten numbers [7]

network implies some sort of watching to insure it does not misbehave or produce unwarranted results, "supervising" simply means whether the network has access to the labels/targets for each input (i.e. what category the specified input falls into). For example, if a given input image depicts a handwritten 4, then that image falls into the target class 4. Therefore, to formally define the aforementioned types of learning: unsupervised learning denotes the lack of class labels for each data point/input, semi-supervised learning denotes that a portion of the input dataset that is greater than zero but less than its entirety have associated target labels, and, finally, supervised learning implies that all data points have an associated target label.

It may initially seem that not giving the network the respective target class may prevent the network from being able to correctly map inputs to output, but in the cases where the network is not provided any labels, then the network attempts to identify its own classes and thus subsequently places the inputs into the network's own defined classes. Each of these networks have their own niche, for example, the internet boasts a large quantity of image sharing sites that contain images without proper labels. This means that one can create an unsupervised machine learning algorithm to grab a large quantity of images and attempt to group them and organize them in some way. This is meant to demonstrate the fact that there exist large datasets that do not have their labels readily available, which are the

perfect partners for an unsupervised machine learning algorithm. Additionally, there are cases where datasets are improperly labeled, which results in the inability for networks to learn. This is yet another case where unsupervised learning is beneficial, since we can instead consider the dataset without its labels and determine whether this allows the network to properly identify and classify the data points.

This project uses supervised learning due to the fact that labeling the data points for this project allows for the results to be meaningful. For example, if the dataset is comprised of thousands of randomly selected unlabeled matches of *Counter-Strike*, some with cheaters and some without, then we would be hard pressed to determine how the network classifies all the different data points. We can abstractly think of a unsupervised neural network as looking at data and clumping them together based off of learned characteristics. However, it may be hard to determine what these characteristics are, and these characteristics may even be irrelevant with regards to cheating within *Counter-Strike*. It could be entirely possible that the network groups players by which ones perform the best (which presupposes that cheaters are inherently more successful than those who do not), or even inanely groups the players by those who look most often at the sky or sun. Additionally, in order to determine how the network groups these players, we are required to watch the matches and attempt to synthesize out the meaning, which might take longer than simply watching the match and determining whether the player is cheating. Therefore, in order to circumvent all of these aforementioned problems, we utilize a supervised learning approach, since this allows results to be easily understandable and meaningful and coincides with the fact that matches must be watched at some point in order to have a useful dataset.

The workflow for a neural network is as follows: First, the network receives some sort of input X, which can be an image, a number, or practically anything that is condensable into a readable format for the network. This input is mapped

to an initial layer which is comprised of a specific number of nodes, where the number of nodes is equal to the number of input variables (e.g. each pixel intensity within the handwritten image in Figure 4.1). Each of these nodes within the layer has a specific weight, which relates to how important a value is in relation to the formation of a prediction by the network, and a bias which is a threshold value for determining whether a specific input and weight are significant enough. For example, if there exists a single pixel that immediately indicates that the image being processed is a four then that pixel's related node is weighted highly since it is significant when considering inputs. After weights are applied to the input value, new values are produced and passed along the network (see Figure 4.2). If there exist more layers, then this process is repeated until all layers have applied their weights and produced a new value.

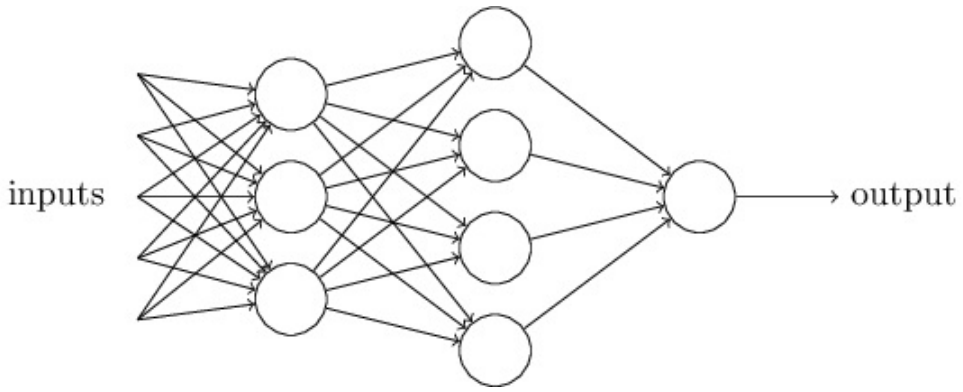

Figure 4.2: A basic depiction of a neural network [18]

One important concept to note is the fact that neural networks base themselves on an abstract understanding of the biology and functioning of neurons within the brain. Electrical signals are passed between neurons in the brain, where neurons are interconnected cells. Each specific neuron has criteria it uses in order to determine whether a signal should continue to be passed through the brain (i.e. to the next neuron). Therefore, we can use the terms nodes and neurons interchangeably in

regards to machine learning, since nodes are comprised of a weight and bias which are responsible for altering how a signal (i.e. an input) that passes through network gets mapped to a specific output.

Therefore, we can formally define the algorithm for generating an output for a specific neuron through:

$$\sum_{i=1}^{n} w_i x_i + b$$

where w_i is the weight of the connection between some node and the current node, x_i is the activation value from node (i.e. an input value to the current node), and b is the associated bias for our current node [18] (see Figure 4.3). Biases are analogous and identical to a threshold value, meaning they determine the ease with which a neuron activates [18].

$$\text{output} = \begin{cases} 0 & \text{if } \sum_{i=1}^{n} w_i x_i + b \leq 0 \\ 1 & \text{if } \sum_{i=1}^{n} w_i x_i + b > 0 \end{cases} \quad [18] \tag{4.1}$$

Since this value is added to the summation of the $w_i x_i$ terms the more negative the bias is, the less likely the weighted sum exceeds 0. Likewise, the more positive the bias is, the more likely the weighted sum exceeds 0. Additionally, the network uses this as a learnable parameter, meaning that it attempts to find the optimal values, which mitigates high values of loss.

Layers within neural networks are simply groupings of nodes which are not connected to each other, but instead connected to every other node from the immediately previous and following layers. Neural networks typically include a multitude of different types of layers, which is dictated by the needs of the machine learning task.

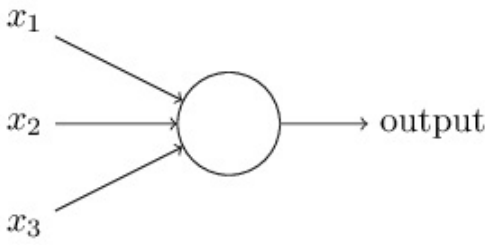

Figure 4.3: A representation of a single neuron within a neural network, where each arrow represents the associated weight between nodes [18]

Next, when considering these connections between nodes, it is necessary to discuss activation values and activation functions. "Activation functions transform the combination of inputs, weights, and biases. Products of these transforms are [used as] input for the next node layer" [20]. These transformations typically involve mapping these outputs to a specific bound. The typically bounds for the output values are $-1 \leq output \leq 1$, or $0 \leq output \leq 1$. However, many other bounds exist for different purposes.

After the network maps an input to a specific output (i.e. makes a prediction), the loss function evaluates its accuracy, which is based upon the difference between the actual answer and the prediction. These typically take the form of a vector with various probabilities for each class label. For example, a possible output vector from the network based off the image of the four could be $\langle 0.1, 0.0, .03, 0.5, 0.0, ..., 0.0 \rangle$, where each entry's index correlates to the prediction for that specific number (i.e. a 0.1 prediction of it being a zero and a 0.5 prediction for the image being a four) [23]. This is meant to punish the network for producing guesses that span multiple different class labels or targets. Finally, the network adjusts the weights of each layer to improve its predictive accuracy, which is achieved through gradient descent. Gradient descent is a method that adjusts the weights to a state with the lowest possible loss associated with predictions (see Figure 4.4).

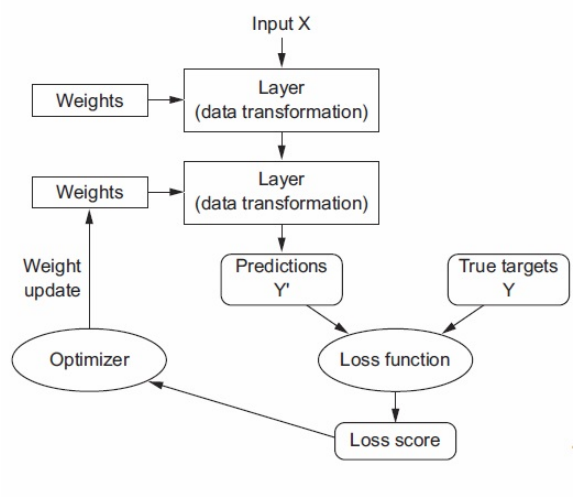

Figure 4.4: Anatomy of prototypical neural networks [7]

4.1 Recurrent Neural Networks

Prototypical neural networks are feed-forward networks, which are simply networks whereby "the output from one layer is used as input to the next layer" [18]. This necessarily means that the information that is passed through the network only flows forward, meaning once a layer produces an output, that output is then used as input for the next layer, and that original layer no longer has any impact with how that output is manipulated and used (i.e. an output from a single layer cannot be reused as a input for that same layer). For many machine learning tasks, this is not an issue and does not limit the network in anyway. This stems from the fact that we can imagine each subsequent layer in these types of networks as producing a "more meaningful representation" of the input (e.g. when processing an image of a handwritten number, each layer synthesizes only the most important information from the image in order to make a prediction and passes it forward). While it may seem detrimental to not allow information to flow freely back and forth, feed-forward networks tend to be more practical for a larger array of machine learning tasks [18].

Obviously, one of the main components that is missing from the typical neural network (i.e. feed-forward) is a form of memory that allows it to use previous inputs or outputs to alter and affect the current input. This lends itself to each input being considered entirely on its own, which, depending on the task, can be beneficial or detrimental. This lends itself towards recurrent neural networks, which are networks that "process sequences by iterating through the sequence elements and maintaining a state containing information relative to what it has seen so far" [7]. Recurrent neural networks emulate the way humans typically learn moreso than the typical feed-forward networks, since humans tend to process information incrementally before conceptualizing the information in its entirety. For example, whilst reading sentences, we consider each word individually as the sentence progresses rather than all the words at once. This also hinges on the fact that the order of the words affects the meaning of the sentence [18].

For the task of cheat-detection, a memory cache is intuitive and beneficial for our goals. This stems from the fact that cheating is a multidimensional gradient, whereby players can cheat in select circumstances with a broad range of different cheating methods. Therefore, rather than considering a demo file in its entirety, we can instead break down the demo into its own time steps (i.e. ticks) and allow the network to consider these ticks individually but also concurrently. This means that the network can view the match as its developing rather than in one entire dump. This is intuitive because, while cheating presents itself in many different forms, there are typically easily identifiable reasons why cheating occurs in matches. For example, two of the most apparent cases within this project's dataset are instances of cheating where the cheater thinks that a player on the opposing team is cheating and thus decides to enable their own cheats to combat that opponent, or, secondly, there are instances where the cheater decides to enable their cheats because they are about to lose the match and would rather win by cheating than to lose by playing

fairly. While these are two small examples, this lends itself towards a belief and outlook that supposes that the majority of cheating can be learned by analysis of the game-state as it develops.

This is also representative of and similar to the manner with which Overwatch cases are reviewed by agents, which is an in-game system whereby players (i.e. Overwatch agents) are given `demo` files of suspicious players in order to review and determine whether a specific player within the `demo` file is cheating. Within Overwatch cases, agents are given a portion of a `demo` file to watch to submit their judgement. However, this obviously includes visual components whereby agents are watching the game-play and events of the match rather than analyzing the specifics of the game-state. This is mitigated by giving the network the same information that an agent would use in their determination of guilt or innocence.

It is important to note that while the network maintains a memory cache throughout learning, the network still utilizes the idea of individual considerations that are seen within the basic feedforward networks. This simply means that the cache is reset with each new consideration of a distinct demo file, since considerations about the previous match do not affect the likelihood of cheating within the subsequent match.

4.2 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is the prototypical recurrent neural network that is present within the Keras library. This is the network of choice for this project due to the fact that this network is specifically designed to avoid the *vanishing gradient problem* [7], which is a phenomenon within deep learning whereby as the number of layers increases, the more untrainable the problem becomes. This is especially apparent when considering recurrent neural networks, because they are mitigating

multiple successive layers of nodes with repeated cached inputs into the same nodes. While, they are quite complex in their construction (see Figure 4.5), they can simply be thought of as a network that has a parallel conveyor belt that layers and nodes can output values to in order to access at a later point during the learning process [7].

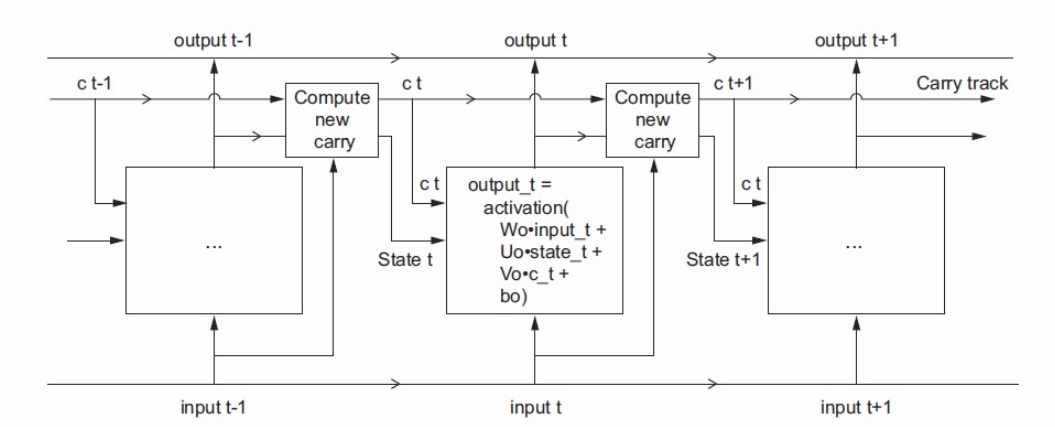

Figure 4.5: Anatomy of an LSTM cell [7]

4.3 Network Construction

The network is constructed quite arbitrarily, since most of the values or construction decisions that are used in the network are derived through trial and error. This means that the values, activation functions, etc., that end up producing the best results are the ones that are typically chosen. Therefore, the input size of our first LSTM layer is obviously dependent on three things: the number of demos that the network is going to be tested upon, the number of ticks within each demo file, and the number of different values within each tick. However, the number of neurons within an LSTM layer is one of the most important architectural design decisions that should be highly optimized. This stems from the fact that the number of neurons within the LSTM layer directly affects the network's memory capabilities. When

the memory capabilities prove to be too insufficient, the network fails to handle long sequences due to its internal memory constraints. Figure 4.6 demonstrates the relationship between the amount of memory and the network's subsequent ability to handle a sequence. When the network memory is insufficient, the network forms its predicted probability of cheating within the first 100 timesteps and does not alter it based upon the latter information from the sequence. Whereas in ideal situations, the network is constantly adjusting its predicted probability of cheating based upon what it is seeing within the file and what it previously saw. Therefore, in order to allow the network to properly handle sequences, a sufficient quantity of neurons needs to be allocated within the recurrent layers.

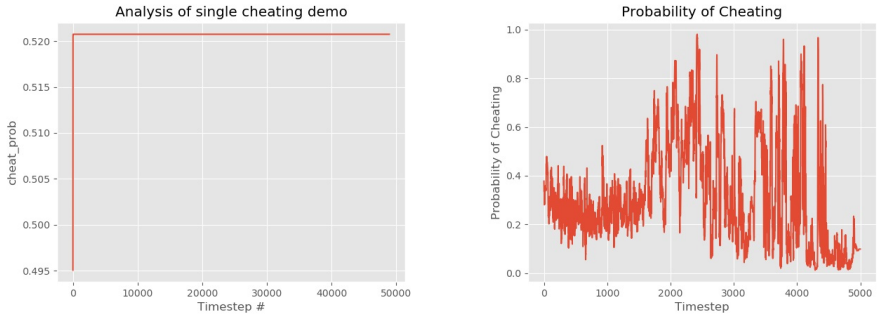

Figure 4.6: Comparison of a network's memory capabilities depending on the number of neurons.

The next layer is a simple dense layer, which takes in the previous values that are returned from the LSTM layer and produces a single output, which is the predicted probability of whether or not the player is cheating. Dense layers implement the prototypical neural network feed forward: `output = activation(dot(input, kernel) + bias)`, where `output` is the value produced by a neuron, `input` is the activation value to be used from the previous neurons, `kernel` is all the associated weights, and this value is then given to the activation function. The activation function for this layer is the sigmoid function since we want the network to output

a probability of that player cheating in order to see the probability of cheating associated with various players.

4.3.1 Gradient Descent

Gradient descent is the means with which the algorithm is optimized in order to improve its predictive accuracy and lower its loss (hence the term "optimization function"). The most prevalent analogy for understanding gradient descent is imagining some sort of object (preferably a ball or something that can roll) and a landscape with slopes (i.e. hilly terrain, or even a simple cereal bowl) [18]. Gradient descent is tasked with getting the ball to the bottom of the landscape (i.e. some sort of valley, or the bottom of the cereal bowl). This is analogous to the network reaching a local minimum in regards to the amount of associated loss (see Figure 4.7).

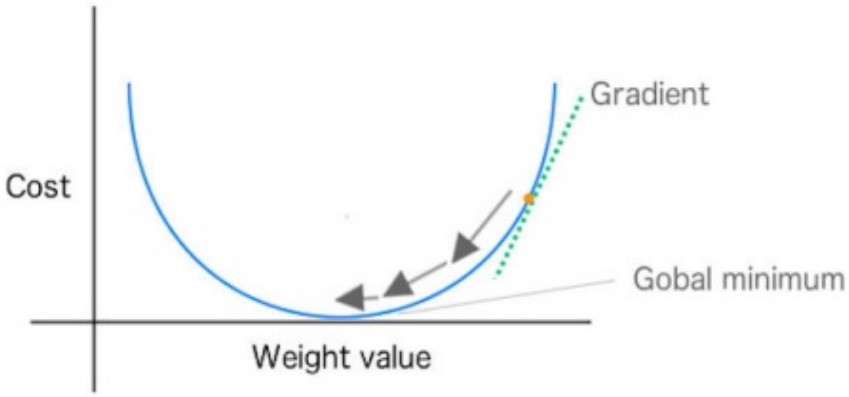

Figure 4.7: Visual representation of gradient descent [20]

The gradient of a function is the direction of steepest ascent, which is simply the direction with the largest slope. The gradient is used in order to find this aforementioned area of low loss (i.e. the local minimum). However, rather than looking at the direction of steepest ascent, we instead consider the negative of the gradient, which is the direction of steepest descent. Therefore, when considering

individual vectors to adjust in order to travel in the direction of steepest descent, we use the following formula [12]:

$$\overrightarrow{x'} = \overrightarrow{x} - \epsilon \nabla_{\overrightarrow{x}} f(\overrightarrow{x})$$

Here, $\nabla_{\overrightarrow{x}} f(x)$ is "the gradient of f [which] is the vector containing all the partial derivatives" for some associated vector $\overrightarrow{x}$ [12]. Additionally, ϵ is the learning rate, which is a misnomer, since it describes the size of the step that is employed in order for determining the new associated weights and biases. Going back to the hilly landscape analogy, if we allow the network to make large adjustments in the direction of steepest descent, we may accidentally end up in regions and positions with higher associated loss. This stems from the fact that rather than our hypothesis space being a simple three-dimensional bowl or valley, it is instead on the dimensional magnitude of tens, hundreds, thousands, or even potentially more. This means that it is quite hard to imagine the appearance of our loss landscape, since it is not as simple as rolling down a bowl-like object to the local minimum. Therefore, the typical philosophy that surrounds learning rate is using very small values in order to more efficiently approach areas of low loss [12].

4.3.1.1 Adam

Adam is "an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments," [15]. Adam essentially uses baseline stochastic gradient descent (SGD) combined with all the best features of all the other contemporary and common-place deep learning optimizers (those being AdaGrad and RMSProp). One of the main problems that is encountered in deep learning projects is optimizing the learning rate of the optimizer. "The learning rate of SGD, as a hyperparameter, is often difficult to tune, since the magnitudes of different parameters vary widely, and adjustment is

required throughout the training process" [30]. Therefore, numerous optimizers exist in order to tackle this very problem, with the earliest being AdaGrad and RMSProp (and eventually Adam). "These algorithms aim to adapt the learning rate to different parameters automatically, based on the statistics of gradient" [30].

Overall, [15] demonstrates that for a large number of deep learning problems, Adam outperforms standard SGD in the rate of convergence. Additionally, due to these aforementioned benefits, Adam is currently one of the fastest growing optimizers within the deep learning community. Therefore, this project employs Adam due to its inherent benefits and strengths.

4.3.2 Activation Functions

Activation functions are used to "propagate the output of one layer's nodes forward to the next layer" [20]. Additionally, activation functions are a means of introducing "nonlinearity into the network's modeling capabilities" [20]. Activation functions are a cornerstone to deep learning because without them, neural networks are simply linear regression models, since we are simply modeling a scalar response. Therefore, careful considerations must be taken in the development and employment of activation functions within deep learning projects due to their massive importance. Additionally, there are many different flavors or types of activation functions which each have a niche within various types of neural networks.

Sigmoid is the main activation function that is present within this project's neural network (see Figure 4.8). It is specifically seen within our dense output layer, since sigmoid outputs values as a probability between 0 and 1. As the input approaches negative infinity, the output approaches zero, while, conversely, as the input approaches positive infinity, the output approaches one. This is an intuitive function for our project, due to the fact that we interested in developing a neural network which can correctly produce confident (i.e. high probability) predictions

on demo files, which are labeled on the bounds of the range of the sigmoid function, i.e. 0 and 1. These two schemes (i.e. our labeling scheme and the sigmoid output scheme) intuitively mirror each other.

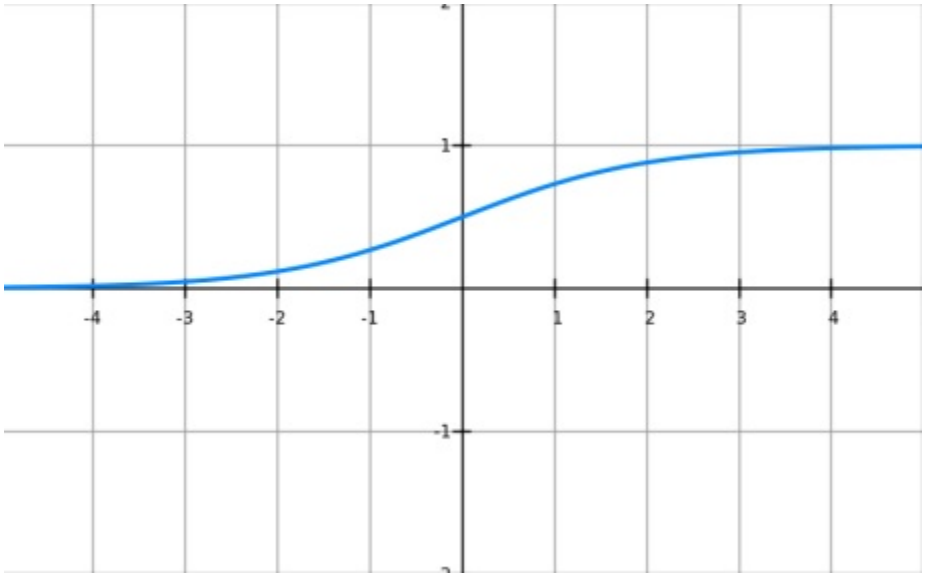

Figure 4.8: The sigmoid activation function [20]

4.3.3 Dropout

Dropout is a method that is employed in order to combat overfitting within machine learning tasks. Overfitting is a phenomenon in which the network attains a high predictive accuracy and/or a low associated loss, yet it performs poorly when given data that it has not seen before. The most desirable trait for neural networks is generalizability, which is the ability for the network to learn generalizable traits and characteristics from the initial training dataset, that also present themselves in data points that fall outside of the training dataset. For example, imagine a scenario where all the data points for this project boast cheaters that always perform very well in the game. This lends itself to the potential that our neural network generalizes the trait that cheating requires players to be performing very well. This significantly

hinders the networks ability to generalize what cheating is on a broader scope, since it has mistakenly correlated cheating with in-game performance. Therefore, if we were to pass the network data points where the cheater is doing poorly, it would most likely surmise that the cheater is clean. While this example highlights a deficiency within the dataset during training, this type of phenomenon can occur even in situations where the dataset is properly representative of the problem as a whole. For example, this same correlation between in-game performance and the likelihood of cheating is still entirely possible when given a diverse dataset, since there is the possibility that the local minimum that is found by gradient descent relies heavily on neurons that represent the various metrics for a players in-game performance. Therefore, this raises the question: How do we circumvent the potential pitfall of the network relying on a select number of learned patterns that may not be entirely representative of the problem as a whole?

Dropout is the quintessential method that is employed in order to combat overfitting within machine learning tasks. Dropout is a method whereby nodes and their related connections are dropped during the training process [25]. The decision for which nodes to drop is typically random, which embeds a bit of entropy during the learning process. This allows the nodes within the neural network to avoid "co-adapting" [25], which is the aforementioned idea of the network having a heavy reliance on a single select set of nodes. Overall, there are relatively few, if any, consequences with using dropout for machine learning tasks, since they typically improve the network's accuracy and reduce the number of required calculations (since certain nodes are no longer being considered). The only potential consequence could be the introduction of randomness and entropy into the training, however, this typically does not result in lower predictive accuracies or higher loss [25].

4.3.4 Loss Function

Loss functions are a means of evaluating the magnitude of the difference between the network's predicted target for a given input and the actual target. The hegemonic and standard loss function within contemporary machine learning, which is typically used as an initial baseline, is *cross-entropy*.

The choice for a loss function is quite obvious for this project, since we are only concerned with the binary choice of whether a single player is cheating or not. Therefore, this project uses *binary cross-entropy*, since we only have two labels, which is the exact scenario where binary cross-entropy is typically employed.

Binary cross-entropy with two class labels is defined as a log-loss function:

$$-y(\log(p) + (1 - y)\log(1 - p))$$

Here, y is a "binary indicator," meaning whether the prediction by the network is correct (where 0 indicates being incorrect, and 1 indicates being correct), and p is the predicted probability from our network that the input falls into some target class [10, 12].

4.4 Padding and Masking

The input size for a neural network must remain constant. Therefore, it is essential to define an input vector that is composed of numerous features that subsequently is passed to the network in order for it to learn. However, one thing that is immediately apparent when interacting with and viewing demo files, or video game matches in general, is the fact that they have variable lengths. Some matches of *Counter-Strike* are very short and can last only 20 minutes, while other matches can go into multiple overtimes and take hours. Therefore, this presents an interesting dilemma when

considering how we should construct and organize the data to give to the network. The first baseline approach is to normalize all the demos to the same tick-length, either capping all demo files to only be as long as the shortest file or extending all the demos to fit the length of the largest file. This first approach might seem more logical and feasible, but it ignores the main stipulation of cheating that this project founds itself upon, this being that cheating is a gradient. Therefore, should we artificially cap all the demo files to fit the length of the shortest file, we may clip out the rounds or instances where a player begins or initiates their cheating. Therefore, our only choice is to artificially extend all the demos to insure that we are passing a constant sized input vector to the neural network. We accomplish this through the inclusion of a special value, which is simply a number. The special value can be arbitrarily defined, but should typically be a value that is not present within the input vector (i.e. any of the features). For example, since view angles within *Counter-Strike* are capped between 180 and -180 for side to side angles, and -89 and 89 for up and down angles, then any value that exceeds those bounds would be an ideal special value.

In terms of implementation, we utilize padding, which means we insert input vectors (i.e. additional artificial timesteps) that have the special value for every feature. For example, if one demo is only 500 ticks, while our longest is 10000, then we pad the demo file that has 500 ticks with 9500 timesteps with the special value as every entry. This allows the network to learn that timesteps that are composed entirely of special values are useless data points and that they should not be used when making predictions. This is a strategy that Chollet emphasizes in his construction of Keras neural networks, since many datasets that are grounded in the real world have missing data points [7].

However, one of the main problems that can arise with the use of padding is the addition of noise within the padded data points, which results from the network

treating the padded values as real values. This is usually avoidable, should the amount of padding be low in comparison to the average length of data points (i.e. when the difference between the average length of a data point and the longest data point is small). For example, padding a demo with 500 ticks with 9500 additional timesteps would hinder the network's ability to efficiently use that specific data point, due to all the added noise from padding.

All of the problems that arise with the use of padding can be avoided with the use of masking, which is simply a means of indicating to the network that timesteps that are comprised entirely of features equal to the special value can be ignored. The main downside with this approach is the fact that this is inherently inefficient when it comes to space considerations, since we are forced to arbitrarily bloat our data points in order to produce meaningful results. However, within the scope of this project, this provides one of the only means of effectively using variable length sequences within a recurrent neural network.

4.5 Batch Sizes

Batches are small subsets of the training dataset that are incrementally fed into the network during training. There are many different philosophies and approaches surrounding the quantity of data that the network should see before calculating the gradient for gradient descent. Stochastic gradient descent's intended batch size is one, but the typical real-world strategy is to train on batch sizes larger than one to introduce a bit of entropy, produce averaged gradients, and decrease the number of calculations required since weights and biases are not being updated after every data point [20]. Following this strategy, there are also mini-batch optimizers (e.g. RMSProp), which are intended for training on batch sizes that range from two data points to the entire dataset. These are highly specialized optimizers that retain all the

aforementioned benefits but are also designed specifically for mini-batches. Finally, there is full-batch training, whereby the network sees the entire training dataset and predicts upon it before making any adjustments to weights and biases. This is typically employed when the dataset is very repetitive, there are few target classes, or there are not many features to optimize. All of these different philosophies and strategies have their place within the deep learning field and their own benefits and consequences.

The strategy that yields the lowest consistent loss and highest predictive accuracy across different training sessions within the scope of this project is online training. Online training is the use of batch sizes that are equal to one, i.e. weights and biases are adjusted after every prediction. This is intuitive, due to the fact that masking is currently unsupported within Keras when training on a graphics card. As the batch size increases, more padding needs to be added to data points in order to normalize their lengths. Following this, as the amount of padding increases, the network has an increasingly difficult time learning from the random noise that is added as a result from padding. Therefore, while in an ideal scenario, we would utilize batch sizes that are greater than one in order to reap the aforementioned benefits with larger batch sizes, the consequences associated with large batches when training on a graphics card are too detrimental to warrant their usage.

CHAPTER 5

Implementation Specifics

There is a large quantity of dependencies that this project uses in order to properly function. First, this project is based almost entirely within Python, with the only exclusion being the cheats we developed in C++ for use in our dataset. Additionally, our dependencies can be broken down into two basic categories, frontend and backend. Frontend entails libraries and functions that are tailored for simple user interaction and alteration. This includes constructing our neural network model, plotting the various performance metrics of our network, and data preprocessing methods. Backend entails the low-level libraries and functions that our frontend methods are built on top of, e.g. TensorFlow, which handles all the inner-network calculations and all the other nitty-gritty details associated with our neural network. This frontend-backend distinction is one of the main reasons why Keras is one of the most popular deep learning libraries today. This stems from the fact that practitioners can build streamlined neural network models with Keras's simplistic frontend and not have to worry about implementing their own gradient descent optimizer or vice versa. This allows networks to be built quickly and easily and is the main reason why we choose Keras.

Our entire list of dependencies is as follows:

- Python - 3.7
- TensorFlow - 2.10

- scitkit-learn - 0.22.1
- matplotlib - 3.1.3
- CUDA - 10.1
- NVIDIA GPU Drivers - 441.87
- cuDNN - 7.6.5

5.1 Training on Graphics Cards (GPUs)

The current trend for training neural networks on large datasets is to utilize a graphics card (GPU) rather than a computer's central processing unit (CPU), since GPUs are very efficient at parallelizing the calculations that are required for gradient descent. This means that GPUs easily outperform standard CPUs in regards to training time. However, many of the large-scale libraries and other supporting mechanisms that allow GPUs to be used within deep learning projects are still in their infancy and practically on the bleeding edge of deep learning, meaning that there are various features whose support has not yet been ported over to their GPU counterparts. Additionally, they are constantly being updated and altered, which can stymie the ease with which deep learning practitioners can get their hands on this technology.

5.1.1 Batch Loading

While demo files within our project undergo a conversion pipeline that removes all the unnecessary information that they contain, they still remain relatively large (on the scale of hundreds of megabytes). Therefore, it is impractical to attempt to load the entire dataset into memory before training. This leads us towards the use

of batch loading, which is a strategy that utilizes generators in order to load the demo files from disk only when they are needed for training by the network. This changes up the training architecture, since rather than having our data points be the multi-dimensional input arrays for each demo, we instead create a list of all the filenames for each data point, which is then used for loading when that demo is required for training.

The `Demo_Generator` class inherits from Keras's `Sequence` class, which is simply the "base object for fitting to a sequence of data, such as a dataset" [8]. The majority of our class is based off the basic skeleton `Sequence` class from [8]. In order for our generator class to function properly it must have three functions, `__init__()`, `__len__()`, and `__get_item__()`.

The constructor `__init__()` is quite self explanatory, but it declares and initializes all the data members for the class, which are `demo_filenames`, `labels`, `batch_size`, and `special_value` (see Listing 5.1). The member `demo_filenames` contains the names of all the filenames which we wish to use during the training process of our neural network, with labels being their associated targets (i.e. 0 for non-cheating and 1 for cheating). The member `batch_size` is self-explanatory but is required in order for us to properly pad batches, since all data points within the same batch must have the same sequence length. Finally, `special_value` is the value that we use for padding the shorter data points (i.e. some value that typically falls outside the bounds of our features).

```
1 def __init__(self, demo_filenames, labels, batch_size,
    special_value):
2     self.demo_filenames = demo_filenames
3     self.labels = labels
4     self.batch_size = batch_size
5     self.special_value = special_value
```

Listing 5.1: The `__init__()` function

The function `__len__()` is responsible for returning the number of batches that the network will train on for each epoch (see Listing 5.2). We intuitively take the ceiling of the number of demo files divided by the batch size, since Keras expects the single non-uniform sized batch to be placed at the end of the training session. For example, if we had a dataset of 10 points and a batch size of 3, then, since we want to train on all 10 data points, we would need 4 different batches with the last batch only containing the last data point.

```
1 def __len__(self):
2     return (np.ceil(len(self.demo_filenames) / float(self.
          batch_size))).astype(np.int)
```

Listing 5.2: The `__len__()` function

The function `__get_item__()` is foundational within our class, since it is responsible for preparing the data points for the network (see Listing 5.3). The expected return for this function matches the same exact format that is used if all the data points were loaded into memory at once, except for the fact that this contains only a subset of all the data points. For example, the format of a single demo file is as follows:

$$\begin{bmatrix} tick_1\_feature_1 & tick_1\_feature_2 & \dots & tick_1\_feature_p \\ tick_2\_feature_1 & tick_2\_feature_2 & \dots & tick_2\_feature_p \\ \vdots & \vdots & \ddots & \vdots \\ tick_n\_feature_1 & tick_n\_feature_2 & \dots & tick_n\_feature_p \end{bmatrix}$$

This means each tick is an array of length `num_features`, and we append each tick into an array of ticks. Then, we subsequently append every demo file into a single list to arrive at our tensor, or multidimensional input array, for our neural network. Therefore, our means of organizing a demo file can be boiled down into a slightly complex lists of lists, with the smallest list being an individual time step.

```
def __getitem__(self, idx):
    batch_x = self.demo_filenames[idx * self.batch_size: (
        idx + 1) * self.batch_size]
    batch_y = self.labels[idx * self.batch_size: (idx + 1)
        * self.batch_size]
    max = 0
    loaded_files = [np.load(file_name) for file_name in
        batch_x]
    for file in loaded_files:
        if file.shape[0] > max:
            max = file.shape[0]
    Xpad = np.full((self.batch_size, max, num_features),
        fill_value=special_value)
    for s, x in enumerate(loaded_files):
        seq_length = x.shape[0]
        Xpad[s, 0:seq_length, :] = x
    return_value = Xpad, batch_y
    return return_value
```

Listing 5.3: The `__get_item__()` function

The process for preparing a batch using the `Demo_Generator` class is as follows: first, we are given an `idx` which corresponds to a position within the dataset (i.e. an index into our `demo_filenames`). We splice `demo_filenames` based off the `batch_size` and `idx` values. We mirror this same process to grab the corresponding labels for the data points within the batch that we are preparing. Next, we can do any of the necessary preprocessing methods in order to prepare the data for the network. This obviously requires us to initially load the demo files from disk. The only preprocessing method we employ is padding, which only alters our data points should the `batch_size` be greater than one. In order to pad our files, we first need to identify the longest sequence within our batch; therefore, we iterate through the data points and compare their sizes (i.e. `file.shape[0]`). After we identify the maximum number of timesteps, we create `Xpad`, which is our list of padded demo files. This utilizes NumPy's `full()` function, which creates a list of multidimensional arrays, which all have the previously found maximum

length with each individual timestep intuitively having a length of `num_features`. Additionally, all values within these full multidimensional arrays are the special value. Following this, we iterate through our batch, and we find the length of each sequence (i.e. `seq_length`) and splice all the values from that sequence into the corresponding padded multidimensional array in `Xpad`. This is an important structure, since it follows the prototypical padding convention of having padded values occur exclusively on the right-most side of sequences. This concludes all the preprocessing methods; therefore, we must return the batch and its corresponding labels as a tuple (when using generators, the `__getitem__()` function is expected to return tuples, and returning it in any other format is prohibited).

Preprocessing should typically be avoided within generator classes due to the fact that it inherently slows down training. This stems from the fact that generators alter the training pipeline to account for the fact that we are using a GPU for training. The CPU, within the GPU generator training pipeline, is responsible for preparing batches to subsequently hand off to the GPU for predictions and training. Therefore, having complex preprocessing methods within the generator class can bottleneck training, since the GPU hangs while waiting for a new batch. Therefore, for optimal performance it is essential to develop a `__getitem__()` function that allows for the GPU to be under constant use.

This class can be modified (and potentially enhanced) with the inclusion of a `on_epoch_end()` function, which is designed to manipulate the data in between epochs. An epoch is one training cycle where the neural network trains on the full training dataset. Therefore, epochs are simply the total number of times the network sees the entire dataset during the course of training. The most common use for this function is to shuffle the data in between epochs. However, more complex methods can be employed to alter the training process between each epoch.

5.1.2 cuDNN Library

"NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in (Deep Neural Network) applications" [19]. cuDNN proudly boasts that it is able to handle deep learning tasks more efficiently (in terms of training time and overall performance) while simultaneously consuming less memory than all alternative deep neural network approaches. However, the main problem with using the cuDNN Library in its current form is the lack of documentation and associated information for both configuring the cuDNN Library for use and how to optimize network performance.

The setup process for the cuDNN Library can impede and prevent one's ability to easily use this powerful library. This stems from the fact that all of the different software libraries that interact with each other have specific versions that only properly function with specific versions of the other software libraries. For example, at the time of writing, TensorFlow's documentation page for GPU support lists the software library dependencies as follows:

- "NVIDIA GPU Drivers -CUDA 10.1 requires 418.x or higher
- CUDA Toolkit -TensorFlow supports CUDA 10.1 (TensorFlow >= 2.1.0)
- CUPTI ships with the CUDA Toolkit
- cuDNN SDK (>= 7.6)
- (Optional) TensorRT 6.0 to improve latency and throughput for inference on some models" [28].

This software library structure daisy chains all the requirements together and can make it very difficult to get all the components to properly work, since if one requirement is installed incorrectly the entire library ceases to function properly.

Additionally, in order to properly benefit from this library, we must use the cuDNN specific layers within our front-end deep learning library (i.e. Keras). This means that rather than using the typical `LSTM()` layer within Keras, we must instead switch to the `cuDNNLSTM()` layer. While the original `LSTM()` layer still appears to use the GPU for training with the cuDNN Library installed, it only uses the memory on the GPU rather than using it to perform calculations. Therefore, it is essential to use all the cuDNN supported front-end layers and features in order to get the most out of the cuDNN library.

5.1.3 Lack of supported features

When Keras is running on a CPU, it has all of the aforementioned features and strategies. This includes dropout, recurrent dropout, masking, and padding. However, all of these features barring only padding are currently unsupported when training on a GPU. There is currently progress being made on this front, with TensorFlow (which is this project's backend for Keras) recently adding full fledged support for variable length sequences, meaning that padding and masking are no longer necessary [27]. However, these features have yet to be ported over to Keras, meaning that there is an inherent trade-off between training on a CPU and GPU.

Nevertheless, the massive speed gains that can be acquired through the use of GPUs far outweighs the relative learning gains that can be had using the more complex features that are bound to CPU usage. This stems from the fact that the network can be trained and altered and retrained on a GPU multiple times before the first training session completes upon the CPU. For example, the first training epoch with this project's entire dataset on a CPU is estimated at 44 hours, while training for 15 epochs on the GPU takes 30 to 45 minutes. Therefore, it is inherently only worthwhile to use the CPU to train the network in the case where the GPU training proves to be too ineffective (i.e. it cannot learn). However, it is important

to reiterate that there is good potential that training upon the CPU yields results that surpass those that can be attained through training on a GPU.

CHAPTER 6

Data and Implementation

Data and the means with which we implement methods on top of it is the main crux of this project. A neural network is only ever as good as the dataset that it is given. Therefore, properly sourcing the data, extracting the relevant information from it, and eventually giving it to the network form the heart of the task of cheat detection within *Counter-Strike*

6.1 Sourcing Data

The data within this project is from a multitude of different places, each having their own benefits, consequences and caveats. First, it is essential to identify the basic idea that players who are actively cheating do not inherently want to "give themselves up," i.e. donate evidence of themselves actively cheating. This is obviously due to the fact that it presents the possibility that their personal user-data is publicly known and thus their account has the possibility of being banned for cheating. Therefore, since we do not have much ability to retrieve data directly from the source (meaning from cheaters themselves), we instead rely on *Counter-Strike*'s Overwatch system.

Overwatch is a system that flags various players who exhibit potentially suspicious behavior or have been reported by multiple other players. The demo file of their game is sent to trusted Overwatch members, who are tasked with watching a portion of the demo file and subsequently submitting a feedback survey stating

whether the user thinks the player is cheating. These files are anonymized and given to users in a unreadable format that cannot be converted using the aforementioned demo-to-plain text conversion application. However, *GitHub* user `takeshixx` created a repository that sources the demo files in their typical demo format and downloads them to extract player information from the Overwatch case. This is an immensely powerful and helpful tool that allows Overwatch users to identify the account information associated with the suspected cheater. This means users can verify whether the player has already been banned for cheating, which helps provide *Valve* with more accurate survey data for Overwatch.

Some liberties have been taken with `takeshixx`'s codebase, and some modifications have been made in order to ameliorate it so it is more suited for this project's goals. The workflow for this method is as follows: First, download an Overwatch case while `takeshixx`'s `overwatcher` application is running in order to retrieve the user-information within the Overwatch case. The demo file is added into the dataset directory in case it is needed for future use. One convention that the demo files use is anonymous labeling of the players within the game. There is a pool of potential names that are randomly given to each player within the demo. However, `takeshixx`'s application circumvents this since it finds the original demo files online and extracts relevant information from that. Additionally, the player that is suspected of cheating is labeled as "The Suspect" within these demo files. Therefore, one of the first crucial steps in labeling the demo file is identifying the real username of "The Suspect." This all stems from the fact that the Overwatch cases are given to agents in an unreadable format that cannot be converted to plain text; therefore, correctly correlating all the real usernames with the random anonymized ones is quintessential to properly labeling the data, since we need to be able to access the plain text format of the demo file in order to extract relevant information about the game to subsequently be given to the neural network. Next, we view the

demo file and label it, which boils down to either labelling the player as a cheater or legitimate player.

This project uses 400 data points, with 200 of those points including cheaters and the other 200 not containing any cheaters. We differentiate between demo files and data points because demo files are from the point of view of the server, while data points are our generated tensors that are through the point of view of the player in question. During the data collection phase, more than 400 demos were encountered, but many are unfortunately unusable due to a multitude of issues, the most common being the incorrect game mode.

Additionally this project employs a labeling scheme for the various types of cheating and non-cheating files within the dataset. The labels are as follows: -1 symbolizes that there is no cheating within the demo file, 0 means that there is no conclusive evidence in regards to whether there is cheating or not, 1 is a catch-all cheating label and implies that there is cheating within the demo file beyond a reasonable doubt, and, finally, 2 symbolizes that the demo file seems to have cheating within it, but it is not beyond a reasonable doubt.

6.2 Data Conversion Pipeline

The data within this project takes a lot of different forms before it eventually passes through the network in order to be predicted upon. Thus, this project uses a data conversion pipeline that synthesizes and filters the demo files down into a more palatable form that can be used by our neural network.

6.2.1 Converting Demo Files to Text Files

Valve's `demoinfogo.exe` application is the tool this project uses to convert demo files to text files. However, these text files in their native/vanilla form lack a crucial

detail that allows the maximum amount of data to be scraped from these files, this being a means of identifying the correlation between a player's username and an `entityID`. Almost every object within a match is an entity with an associated identification number (i.e. `entityID`), e.g. weapons, players, props, etc. These numbers are immensely important because one of the main points of data within the demo files are *Entity Delta updates*, which are packages of information distributed between clients and the server that contain a change in various entity values (see Listing 6.1). Entity values are things such as the x, y, z position, pitch, yaw, velocity in the x, y, and z directions, and other similar values. The second caveat is that all of the game events that are contained within the ticks between the clients and the server are things within the game that produce a sound or alter the game-state in a major way (e.g. `weapon_fire` produces a sound, while `round_start` commences a round within the match). This means that if we only rely on the game events to source information from the demo file, we miss all the events that players perform that do not produce a sound. For example, players can move soundlessly in the game if they hold the shift key, and additionally, players do not produce any sounds when they adjust where they are looking. This entire schema means that relying purely on game events significantly hinders the amount of information that we can extract from the demo files. Therefore, the code library that generates and compiles the `demoinfogo.exe` application has been modified to output every player's entity number and username at various points within the game.

```
1 Entity Delta update: username: Anonymous, id:3, class:40,
     serial:839
2 {
3 Table: DT_CSPlayer
4 Field: 1, m_nTickBase = 4557
5 Field: 2, m_vecOrigin = -1566.161011, 600.366577
6 Field: 4, m_vecVelocity[0] = -3.104770
7 Field: 5, m_vecVelocity[1] = 55.471664
8 Field: 20, m_angEyeAngles[0] = 353.430176
```

```
Field: 21, m_angEyeAngles[1] = 92.186279
}
```

Listing 6.1: Entity Delta Update

Additionally, all the game events within the game are modified to also include all the necessary metadata associated with the player that performed that game event. For example, since Entity Delta Updates include the velocity in the x, y, and z directions for a player/entity, those values are also added to the game event.

Finally, there are a lot of organizational changes that are made to the `demoinfogo.exe` application in order to bundle information more efficiently. For example, game events take on a form that is similar to a JSON format/Python dictionary (see Listing 6.2), therefore alterations are made to allow the Entity Delta Updates to take a similar form.

```
player_footstep
{
 userid: Anonymous (id:10)
  position: 2971.985107, -122.186562, 1613.031250
  facing: pitch:0.670166, yaw:213.189697
  entity: 9
  team: T
}
```

Listing 6.2: player_footstep game event

6.2.2 Synthesizing Important Information

Synthesizing important information from demo files is a crucial step in the pipeline due to two reasons, first, `demoinfogo.exe` dumps all the information from the demo file into a plain text document. The majority of the information from the demo file is entirely useless for the purposes of this project, and these plain text files can reach up to sizes of five or more gigabytes (when considering professional matches).

Therefore, when we are considering a dataset that is of the magnitude of hundreds of demo files, having files that are needlessly large is unwarranted and unwanted. Therefore, we develop a intermediary parsing method to extract only the essential information from each file in the dataset.

Therefore, building off the previous section, we want to extract all of the Entity Delta Updates and game events since those are the two main hubs where information is stored for a single match of *Counter-Strike*. However, this is easily the most difficult step in the entire pipeline due to the basic structuring of demo files. This stems from the fact that each event differs in the information it exchanges between clients and the server, `entityID` values change constantly depending on which players are currently connected to the server. If there are non-player entities (i.e. bots) who join the server to replace players who disconnect unexpectedly, these bots are in turn treated differently when compared to a normal player, and many more reasons. Therefore, constructing methods that synthesize the important information and that are also generalized enough to be able to handle all the different potential cases that arise during matches of *Counter-Strike* is quite challenging. This project accomplishes this feat through the `Scrape_Demo` class.

6.2.3 Organizing Scraped Information into Time Steps

The construction of our sequences is the single most important factor that enhances or degrades the ability for our neural network to learn and identify the patterns within our dataset. Our sequence is our aforementioned tensor; therefore, we use these terms interchangeably. Each individual time step needs to be composed of accurate metrics (i.e. features) that properly correlate with whether a player is cheating. For example, a time step that only includes the names of all of the players within the game is not something that allows the network (or even a human being) to determine whether someone is cheating. Therefore, each time step is constructed

and segmented into different portions that correlate with information that helps to determine whether a specific player is using a single specific cheat.

6.2.3.1 Game-Assistance Based Cheats

We can break down game-assistance-based cheats into two basic categories that are representative of the two main mechanics that cheaters employ assistance applications with, those being movement based cheats and aim assistance cheats. While there are other ways of interacting with the game environment outside of moving and shooting, these directly influence the cheater's ability to win games, thus making them ideal candidates for the development of cheats.

6.2.3.2 Movement Based Cheats

In order to identify whether players are using movement based cheats, we have to include values within each time step that model how the player is moving through time. Therefore, the first values that are obvious to include are all of the players' positions within each time step. These values allow us to derive the more important metrics such as the distances between the cheater and all other players within the game. This is important because one of the most prevalent cheats within *Counter-Strike* is denoted as "spinbotting," which is a cheat whereby a player employs aim-assistance, vision-assistance, and "B-Hop"-assistance, whilst simultaneously spinning around in circles to make it hard for other players to accurately hit them. This type of cheater is always moving towards enemy players in order to eliminate them. Additionally, this allows us to create a metric that stores the change in distance between all the other players and the cheater between each time step.

The next obvious metric to include to combat movement-based cheats is the velocity of the cheater throughout each time step. Players who are cheating and

employing methods such as "B-Hop"-assistance have higher velocities than those who do not employ those cheating methods. Much like the locations and distances we also include the change in velocity between each time step, since players who are using "B-Hop"-assistance are moving at high speeds, which means that the delta in velocities between time steps is generally larger than that of non-cheating players.

6.2.3.3 Aim-Assistance Cheats

Aim-assistance cheats are some of the most researched cheats within the scholarly world, therefore there is an abundance of possible metrics that we can employ in order to aid our network in determining whether someone is cheating. The first and baseline values that we pass to our network is where the cheater in question is currently looking. This establishes a baseline and should help the network in identifying "spinbotters," since one of their main characteristics is that they are looking at the ground and spinning around. Therefore, should the network see that a player is constantly looking at the floor then they have a higher chance at being a "spinbotter."

Next, much like all the other metrics, we also use the delta or change in where the player is looking at each time step. Cheaters who employ aim-correction applications have larger deltas between time steps since aim correction applications rewrite the cheaters current pitch and yaw with the requisite values to be looking/aiming at an enemy player. Additionally, we can conceive of a velocity metric, which is an analysis of the change in the pitch and yaw values over a certain number of previous time steps. This helps in the aforementioned analysis of aim correction, since this value will typically reach inhuman numbers/levels when players employ aim-assistance.

The final and most important values are similar to the delta, but they are instead the values for pitch and yaw that an aim-correction application would use based on

the current positions of all the other players at that current time step. Much like the velocity, we use this value over multiple time steps (typically adjacent ones), since if the aim was corrected during the previous time step to that exact value, then we know without a doubt that a cheat was used between the two time steps, because we calculate these necessary pitch and yaw values in the same exact manner and methods that cheaters employ when using aim-assistance.

6.2.3.4 Knowledge-Based Cheats

Knowledge-based cheats are the hardest to create metrics for determining whether a player is cheating or not. This stems from the fact that it is typically the hardest type of cheat to determine, since it is not something that is always visually blatant. For example, if a player has a knowledge hack that renders the enemy players' positions on the in-game mini-map, the cheater has a general understanding of where the opponents are, but not a pin-point location. However, the most visually blatant knowledge-based cheat is wallhacks when the cheater is not actively attempting to hide the fact that they are cheating. This is seen through the player constantly looking and aiming at enemy players through walls. Therefore, the only metric that we can use to determine whether a player is using knowledge-based cheats is use of the same aforementioned delta analysis between where the cheater is currently looking and the angles that are required for them to be looking at an enemy player.

6.3 Scrape_Demo Class

`Scrape_Demo` is the class that is responsible for storing/tracking various meta-values for a demo file whilst iterating through it. These meta-values include the filename for the demo, the desired output filename to write the scraped information, the current tick value that the iteration/loop is at, a list of all the events within the demo

file, a dictionary of all the players in the game, and a dictionary to convert names to `xuid` values (which are special unique identifiers for accounts within the game library, *Steam*, that houses *Counter-Strike*).

6.3.1 Scrape_Demo function

The `Scrape_Demo` function does the initial heavy lifting in order to properly convert a bloated demo file into a smaller more palatable scraped demo file. The main means with which it achieves this end is through the use of regular expressions. The fundamental regular expression is:

```
((?:tick: ([0-9]+))|(?:Entity Delta update: name:([^\n]*?),
id:[0-9]+, class:[0-9]+, serial:[0-9]+ ({[^}]*\}))|
(?:([a-zA-Z]+[_ ][a-zA-Z]+)\r?\n\{\r?\n(?:.*?\}\r\n)))
```

The regular expression (REGEX) has three main match cases, the first being `(?:tick: ([0-9]+))`, which simply matches the current tick value in a capture group (see Listing 6.3).

```
1 ---- CNETMsg_Tick (12 bytes) -----------------
2 tick: 121782
3 host_computationtime: 11016
4 host_computationtime_std_deviation: 1317
5 host_framestarttime_std_deviation: 86
```

Listing 6.3: CNET message with associated tick

The next two cases that the REGEX matches are Entity Delta updates and game events, which use:

1. `(?:Entity Delta update: name:([^\n]*?), id:[0-9]+, class:[0-9]+, serial:[0-9]+ ({[^}]*\}))`

2. `(?:([a-zA-Z]+[_ ][a-zA-Z]+)\r?\n\{\r?\n(?:.*?\}\r\n)))`

These two match cases mirror each other in their form, where we employ a non-capture group to get the name of the event (i.e either an Entity Delta update or some type of game event), and a capture group to grab all the important information within the respective event. The main means with which they differ is through the use of a capture group for the name of the player within the Entity Delta update, which stems from the inherent structural constraints for Entity Delta updates (i.e. we cannot easily mirror the fact that game events have the username field within the event itself within Entity Delta updates).

Therefore, using Python's `re` library, we compile the regular expression, open the file which we wish to scrape, and use the `findall()` function in order to find all the matches within the file (i.e. all the ticks, all the Entity Delta updates, and all the game events). All of these matches are stored within a list which we parse through to further remove unnecessary matches. The majority of Entity Delta updates and game events differ in the amount of relevant information that they contain. For example, some may include the new position that the player is currently in while some may not, since there is potential that the player has not altered their position between the previous time step and the current one. Therefore, one of the essential tasks of this iteration through all the game events is to only retain the events that contain at least one important metric (i.e. position, velocity, or pitch and yaw).

Since we have combined all the different cases we want to match into a single regular expression, when we iterate through the list we have to check which match groups actually contain matched information. The regular expression contains six different match groups. The first is Group 0, which is the entire matched string (i.e. the entire tick message, the entire Entity Delta update, or the entire game event). Next, Group 1 contains the current tick value if matched, while Group 2 stores the username of the current player in an Entity Delta update and 3 stores the respective

content of that Entity Delta event. Finally, Group 4 contains the name of the game event, while Group 5 contains the respective content for that game event. Therefore, in a single iteration during our loop through all the scraped information in the demo file, we simply check which match group is non empty in our event list that is returned from the `findall()` function.

The next step is to extract the important information from the event, which begins by declaring and initializing all of the desired values which we wish to extract from an event (i.e. the position, velocities, and pitch and yaw). Then we utilize event-specific regular expressions to match the aforementioned desired information. For example, while game events present positional information as `position: x, y, z`, Entity Delta updates store them as `m_vecOrigin = x, y` and `m_vecOrigin[2] = z`. This means that we need specialized regular expressions in order to properly extract the important information from the demo event (i.e. game events, Entity Delta updates, or ticks).

Within the code block that is responsible for the extraction of information from game events, we have specific and special game events that we treat differently from all the other types of game events, these being: `player info`, `player_team`, `player_death`, and `player_hurt`.

6.3.1.1 PLAYER INFO

`player info` events are the initial events that we iterate towards during our parse, since these are typically the first events that human players incur within the game. These allow us to determine which players are within the match, which we subsequently use to append to the `Scrape_Demo` class's `player_list` private member. Additionally, this allows us to retrieve a player's unique `xuid`, which is a unique value given to every account within the game library that houses *Counter-Strike* (see Listing 6.4). These are always preferable to use rather than a player's username,

since it is thoroughly defined as a number that begins with 76561198 and never changes. This is contrasted by the fact that usernames can contain practically any writable character, including special escape characters, and they can constantly change throughout the match. For example, `\r` and `\n` are valid characters that properly function within names, which allows for players to create usernames that interfere with the way in which fields and entries are organized within demo files. Fields and entries within Entity Delta updates and game events are delimited with `\r\n`, meaning they take the form: `position: x, y, z \r\n facing: pitch: p, yaw: y \r\n...`. Therefore, names that include those characters hinder our ability to extract information.

```
player info
{
 adding:true
 xuid:76561198...
 name:Anonymous
 userID:3
 guid:STEAM_1:1:...
 friendsID:...
 friendsName:
 fakeplayer:0
 ishltv:0
 filesDownloaded:0
}
```

Listing 6.4: `player info` game event

6.3.1.2 `PLAYER_TEAM`

`player_team` game events are the next game events that require special treatment, due to the fact that they are responsible for players joining the match, leaving the match, and switching teams. They have two main uses: first, determining which players within the match are actual players and which ones are spectators or

observers, and second, adding new bot names to `Scrape_Demo`'s `player_list` data member when they join the server to replace a human player that disconnected.

Counter-Strike has two basic teams, the attacking team and the defending team, however, there is an additional team specifically devoted to spectators. Spectators can switch between all the players in the match and watch any player's point of view during the game. Additionally, there is the concept of observers, which are analogous to cameramen within sporting events, since they are responsible for dictating which player's point of view is displayed to the audience during tournaments and professional matches. Observers only appear in tournaments and professional matches, meaning that the typical Overwatch case does not have any observers (and rarely spectators either). Nevertheless, these players are treated much like all the other real players within the match, since they can enact or induce certain game events as if they were a real player. Therefore, it is essential to flag and identify these players so that their game events are ignored.

Next, players often have unforeseen technical issues that result in them disconnecting from the match. *Counter-Strike* attempts to remedy these situations by allowing a bot to take the place of the disconnected player. These bots can be taken over by real players, basically acting as a second life for that specific player. Therefore, in order for us to be able to properly handle this, we need to always keep track of which players are currently connected and disconnected, which also subsequently means we need to keep track of the bots that are currently within the game as well.

The important metrics within a `player_team` event are the `disconnect` and `team` fields (see Listing 6.5). `disconnect` can either be 0, meaning that the player is reconnecting to the game (or connecting for the first time, in the case that it is a bot replacing a player), or 1 meaning that the player or bot is disconnecting. The `team` field indicates which team the player is switching to. This can be four

different values: 0 for a player that is no longer in the game, 1 for spectators, 2 for the attacking team, and 3 for the defending team.

```
player_team
{
 userid: Anonymous (id:3)
  tickbase: 450
  facing: pitch:1.422729, yaw:329.276733
  entity: 3
 team: 2
 oldteam: 0
 disconnect: 0
 autoteam: 0
 silent: 0
 isbot: 0
}
```

Listing 6.5: `player_team` game event

6.3.1.3 PLAYER_DEATH

The `player_death` event is one of the most important events within this project and *Counter-Strike* demo files as a whole. This is due to the fact that it involves multiple players, which means we treat them as multiple different events (i.e. get all the necessary metrics for all the players involved in the event). Additionally, these allow us to analyze how the player is performing throughout the match in terms of in-game performance (i.e. kills, deaths, assists, damage done, etc.).

Overall, we can treat a `player_death` event much like any other game event, just with the added stipulation that we need to track which player died, which player was the killer, and which player assisted, if there was one (see Listing 6.6). However, this requires the creation of the `Player` class, in order to efficiently organize all the pertinent information.

```
player_death
{
 userid: Anonymous_2 (id:5)
  position: -625.223145, 711.190857, -33.999344
  vecs: 33.898331, -94.573013, 0.000000
  tickbase: 7545
  facing: pitch:354.830933, yaw:19.605103
  entity: 4
  team: T
 attacker: Anonymous (id:3)
  position: 55.030094, 925.404358, 74.876999
  vecs: 0.000000, 0.000000, 0.000000
  tickbase: 7545
  facing: pitch:9.728394, yaw:197.006836
  entity: 10
  team: CT
 assister: Anonymous_3 (id:9)
  position: -18.733099, 899.216797, 61.064739
  vecs: 0.000000, 0.000000, 0.000000
  tickbase: 7547
  facing: pitch:15.188599, yaw:196.572876
  entity: 8
  team: CT
 assistedflash: 0
 weapon: revolver
 weapon_itemid: 17506698137
 weapon_fauxitemid: 17293822569103491136
 weapon_originalowner_xuid: 76561198...
 headshot: 0
 dominated: 0
 revenge: 0
 wipe: 0
 penetrated: 0
 noreplay: 0
}
```

Listing 6.6: `player_death` game event

6.3.1.4 Player class

The `Player` class is a very simple class that is only concerned with tracking the various player statistics throughout the duration of a demo. Each player object has 11 data members, those being `playername`, `xuid`, `kills`, `deaths`, `assists`, `adpr`,

`headshots`, `total_shots`, `remove_player`, `num_game_events`, and `wallbangs` (see Listing 6.7). The majority of these are self-explanatory, except for `adpr`, `total_shots`, and `wallbangs`. First, `adpr` is a metric that stands for average damage per round. All of the various weapons within *Counter-Strike* that players can use vary in the amount of damage that they inflict should the player accurately hit an enemy player. Additionally, there are portions of the enemy player models that if hit, deal more damage (i.e. receive a damage multiplier). Therefore, since cheaters typically have augmented aim, they tend to receive the damage multiplier more often than non-cheating players.

Next, `total_shots` is a slight misnomer but mimics that way in which *Counter-Strike* calculates player-specific accuracy metrics (i.e. headshot accuracy). When *Counter-Strike* estimates the relative accuracy of a player (i.e. the ratio of times they receive the damage multiplier over the times that they did not), it only considers the times in which the player accurately hits an enemy player. This is contrasted by the true accuracy of a player, which would be the ratio of the number of shots that accurately hit an enemy player over the total shots fired (i.e. those that hit enemies plus those that do not). This mostly stems from the fact that there is a gameplay tactic called spamming, in which a player attempts to blindly fire their weapon at a map location that may or may not contain an enemy. Therefore, this would artificially lower most players true accuracy and result in it not being representative of the how often the player attempted to accurately hit an enemy.

A *wallbang* is a shot that penetrates an object in the game's environment before hitting an enemy player. The name stems from the fact that walls tend to be the most penetrateable objects within maps. The `wallbangs` field tracks these. This is one of the easiest metrics to determine whether a player is cheating or not due to the fact that a lot of cheaters employ methods that allow them to know the locations of enemies through walls. This in turn allows them to fire shots through these

aforementioned walls and get hits/kills that a normal player would never get. A wallbang is not something that is exclusive to cheaters; normal players can perform them too, but they tend to receive far fewer and attempt far fewer than cheaters.

The `num_game_events` and `remove_player` data members are both metrics designed to ease our ability to determine whether an entity within the game is a real player rather than a spectator or observer. If a player joins the spectating team during the demo, then we know that they are not one of the 10 real players within the match, since the ability to join the spectating team is prohibited to those 10 players. Additionally, should that initial approach not remove all the unnecessary players within the demo, we can analyze the total number of game events that each player incurred during the match. Real players have heaps more game events than spectators and observers (and even bots for that matter). Therefore, simply comparing the total number of game events provides a supplementary method for identifying the 10 real players within the match.

```
1  class Player(object):
2      def __init__(self, playername, xuid):
3          self.playername = playername
4          self.xuid = xuid
5          self.kills = 0
6          self.deaths = 0
7          self.assists = 0
8          self.adpr = 0
9          self.headshots = 0
10         self.total_shots = 0
11         self.wallbangs = 0
12         self.remove_player = False
13         self.num_game_events = 0
```

Listing 6.7: The `Player` class

6.3.1.5 PLAYER_HURT

The `player_hurt` game event is quite similar to `player_death` in terms of organization, but differs mostly in the terminology used for various fields (see Listing 6.8). For example, rather than specifying whether the shot that injured/hurt the player was a headshot or not, it instead lists the `hitgroup`, which is the collection of different body parts on the player models within the game. Additionally, the `player_hurt` event provides us with the amount of damage the attacker applied to the hurt player, which we can then use to update the various player performance metrics. Finally, much like `player_death`, we can treat this as two separate events for the two players involved. This is only possible with inbuilt functionality that handles this specific event, since rather than using `re`'s `search` function to find the first occurrences of position, velocities, and pitch and yaw, we must use `re`'s `findall` function in order to properly extract all the relevant information.

```
player_hurt
{
 userid: Anonymous (id:10)
  position: 1180.895386, -597.700012, 132.237579
  vecs: -39.881027, -52.565716, 0.000000
  tickbase: 27146
  facing: pitch:3.092651, yaw:182.279663
  entity: 9
  team: CT
 attacker: Anonymous_2 (id:2)
  position: 274.065155, -636.064331, 96.031250
  vecs: 2.371802, -72.996651, 0.000000
  tickbase: 27145
  facing: pitch:356.099854, yaw:1.691895
  entity: 2
  team: T
 health: 51
 armor: 92
 weapon: ak47
 dmg_health: 49
 dmg_armor: 7
 hitgroup: 1
```

```
23 }
```

Listing 6.8: `player_death` game event

6.3.2 Constructing Input

After a file is scraped, it must undergo the next part of the prepossessing pipeline, which entails the conversion from the list of game events which was produced by the `Scrape_Demo` function into a sequenced format (i.e. a bunch of multi-dimensional arrays). The sequenced format is simply a special list with each entry being a time step of the demo. Therefore, parsing of the game events is as follows. First, identify the 10 players within the demo file and give each a unique identifier (an integer), with the player of interest (i.e. the player being analyzed whether they are cheating or not) getting identifier 0. The unique identifiers range from 0 to 9 and are used to determine the indices of various features for each player. After all the players are added to the list, we can begin crafting time steps, which means grouping all the distinct game events that occur at the same tick into a single time step array. This is a two part process. First, we must append all the game event information to the list (i.e. all the players' positions, pitches and yaws, etc. from the same tick). Then, after all of the baseline information is added, we calculate the various aforementioned cheat detection metrics (i.e. aim correcting angles, deltas between time steps, etc.). After all the events are parsed, we write this to disk in order to use it for batch load training for our neural network.

When analyzing a single event within the game events file, each has 20 different entries: the tick number, the name of the game event, the player's `xuid` or name (should they be a bot), the x, y, and z positions, the velocities in the x, y, and z directions, `pitch`, `yaw`, `kills`, `deaths`, `assists`, `adpr`, `headshots`, `wallbangs`, `total_shots`, `headshot_percentage`, and `wallbang_percentage` (see Listing 6.9).

When constructing the array for a single time step, we only need 17 out those 20 entries, since the tick number, event, and name are all irrelevant when it comes to training our neural network.

```
offset = player_list_[curr_player_xuid]
# event list's format per event [tick, event, name, x, y,
    z,
# vec0, vec1, vec2, pitch, yaw, kills, deaths, assists,
    adpr,
# headshots, wallbangs, total_shots, headshot_percentage,
# wallbang_percentage]
init_val = offset * 17
item_index = 3
while init_val < init_val + 16:
    if(item_index < 11):
        # raw values from the file (i.e. decode these
            bytes)
        input_vector[init_val] = float(item[item_index].
            decode())
    else:
        # artificially calculated values (i.e. no decoding
            )
        input_vector[init_val] = float(item[item_index])
    init_val += 1
    item_index += 1
```

Listing 6.9: Loop responsible for extracting values to be placed into the input array

Therefore, the first 170 entries within our crafted time step are these 17 values that we grab directly from the game events list. Additionally, with the use of an offset value to index into the proper position within the time step vector, all players occupy subsequent length-17 partitions of the vector (i.e. entries 0-16 correspond to player with unique identifier 0, 17-33 to player with unique identifier 1, etc.).

While looping through the list of game events, we keep an internal variable that tracks the current tick. Once we encounter an event with a different tick, we know that we have added all the possible players into the current time step vector (i.e. the baseline 170 values), and we can then calculate all the supplementary cheat

detection metrics (see Listing 6.10). After these values are calculated we can add the crafted time step vector into a list of all the time steps. This is subsequently followed by writing that list to disk in order to be used by our generator during the training of our neural network.

```
1 if curr_tick != item[0] and curr_tick != '0':
2     curr_tick = item[0]
3     write = True
```

Listing 6.10: Verifying that we are still iterating on the same time step

One of the design principles that is present within our construction of time steps for the neural network is continuity of players in the game environment. This stems from the fact that Entity Delta updates only provide us with deltas in player metrics, meaning we only see the values that have changed for a single player between time steps. This means that while an Entity Delta update may only provide us with, for example, a new pitch and yaw value, we have to assume that the player is in the same location as their previous time step. Therefore, when we build time steps, we have to iterate through the entire vector, find any 0 or 0.0 values which indicate that that value is not present within any of the game events or Entity delta updates within that time step and replace those empty values with the non-empty ones from the previous time step (see Listing 6.11).

Additionally, to ensure that the network does not learn how to distinguish between 128 tick demos (which are primarily non-cheating files) and 64 tick demos (which are primarily cheating files), we employ multiple tick reduction methods. We achieve this through the use of `reduce` and `mod_counter` variables. The `reduce` variable is the value by which we wish to reduce the amount of Entity Delta updates, meaning if `reduce` is equal to 4, we reduce the amount of ticks within the demo file by a factor of 4 (e.g. if there are 400 Entity delta updates, we reduce it to 100

Entity Delta updates). The mod_counter variable counts the number of Entity Delta updates that have occurred and is subsequently used in conjunction with reduce and the modulus operator to employ synthetic tick reduction (see Line 1 in Listing 6.11).

```
1 if (mod_counter % reduce == 0 and item[1] == b'Entity␣
  Delta␣update') or item[1] != b'Entity␣Delta␣update':
2     if item[1] == b'Entity␣Delta␣update':
3         mod_counter += 1
4     for index, value in enumerate(input_vector):
5         if (value == 0.0 or value == 0) and len(
              list_of_ticks) > 0:
6             input_vector[index] = list_of_ticks[len(
                  list_of_ticks) - 1][index]
```

Listing 6.11: Implementing continuity of time steps and artificial tick reduction

6.3.3 BATCH_LOADER

The batch_loader function is the last main function within the Scrape_Demo class, and it is primarily responsible for crafting a tensor containing all the filenames of all the files that we want our network to train on. It has four parameters, including suspects_json, cheating_labels, noncheating_labels, and num_demos. The suspects_json parameter is the file containing all the information in order to craft a datapoint (i.e. name of a potential cheater, a demo file, etc.). The cheating_labels and noncheating_labels are lists containing the label for the various types of cheats and noncheats that are present within the data set (i.e. WallHack, AimBot, SpinBot, professional non cheater, non professional non cheater etc.). Finally, num_demos is a metric that ensures that the same number of noncheating datapoints and cheating datapoints are used within training (e.g. passing 50 means that 50 cheating demos are used as well as 50 noncheating demos).

First, we open the JSON file containing all the information pertaining to the

labeled dataset (see Listing 6.12). Then, we initiate a loop sequence for each entry within that JSON file. For each entry, we grab the name of the demo file, the name of the player with whom we are going to analyze, that players respective `xuid`, and the label for that data point (i.e. the `cheating` variable in Line 13 in Listing 6.12). Additionally we declare and initialize the `demofile_npy` variable, whose construction is very important. When using a batch loader, all the data points need to have unique names for indexing. Therefore, since we have numerous demo files that are used multiple times within training (e.g. multiple cheaters or non-cheaters within the same game), we have to develop a naming schema that creates unique names for each data point. We achieve this through the use of a player's unique `xuid` and the name of the demo file in conjuction (see Line 12 in Listing 6.12).

```
1  with open(suspects_json, 'r', encoding='utf8') as f:
2      data = f.read()
3      try:
4          data_json = json.loads(data)
5      except Exception as e:
6          print(e)
7          return
8      for s in data_json:
9          demofile = s['demo']
10         name = s['name']
11         xuid = s['xuid']
12         demofile_npy = xuid + '_' + demofile[:-3] + 'npy'
13         cheating = int(s['cheating'])
```

Listing 6.12: Iterating through the JSON file

Next, after initializing and declaring all of the various variables, we analyze the cheating label (see Listing 6.13). We check three basic conditions. First, if the cheating label falls within one of our lists of cheating or non-cheating labels. This is followed by checking whether we have already reached the desired number of data points for that specific target class. If we fail any of these conditions, we continue iterating through our list of data points. However, if we pass these conditional

statements, then we check whether the file exists, and if so, we add that to our input list of file names as well as adding the respective cheating label (i.e 0 or 1) to our list of target labels for each data point.

```
1  if cheating not in cheating_labels and cheating not in
       noncheating_labels:
2      continue
3  elif cheating in cheating_labels and num_cheat_demos >=
       num_demos:
4      continue
5  elif cheating in noncheating_labels and num_clean_demos >=
       num_demos:
6      continue
7  if(isfile(f)):
8      file_list.append(f)
9      if cheating in cheating_labels:
10         label_list.append(1)
11         num_cheat_demos += 1
12     else:
13         label_list.append(0)
14         num_clean_demos += 1
```

Listing 6.13: Checking labels and building the file list for our batch generator

CHAPTER 7

Results

This project employs numerous different testing methods to ensure the optimal network environments and conditions are found. This includes initial baseline testing methods with little to no optimization methods, L1 and L2 Regularizers, subsampling with Truncated Backpropagation Through Time (TBTT), condensed sequences, and *k*-fold cross-validation. The results are generally similar across all the different fronts but most of the improvements are seen with general lack of overfitting.

7.0.1 Standard Training and Testing

The standard approach we take within this project is the validation set approach. "The validation set approach... involves randomly dividing the available set of samples into two parts, a training set and a validation set or hold-out set. The model is fit on the training set, and the fitted model is used to predict the responses for the observations in the validation set" [14]. This method is the most straightforward and requires little to no overhead, as this is the standard way of training and testing networks using Keras.

However, this method is only proficient at providing baseline estimates of network performance. This is due to a multiplicity of issues, first, "the validation estimate of the test error can be highly variable, depending on precisely which observations are included in the training set and which observations are included in

the validation set" [14]. This can be seen within Figures 7.1, 7.2, and 7.3. Additionally, the validation set error "may tend to overestimate the test error rate for the model fit on the entire data set" [14]. Therefore, while this method allows us to generate results in a simple and straight-forward manner with little to no overhead, it is inherently important to utilize multiple different training and testing methods so that we can get a full understanding of how the network performs.

The first step before building and training the network is to build the list of filenames and associated targets (see Listing 7.1). This is straightforward, since `Scrape_Demo`'s `batch_loader()` function returns a list of the training filenames, the validation filenames, the training targets, and the validation targets.

```
1 combined_list = Scrape_Demo('', '').batch_loader('
    master_suspects.json', [1, 3], [-1], 200)
2 X_train_filenames = combined_list[0]
3 X_val_filenames = combined_list[1]
4 y_train = combined_list[2]
5 y_val = combined_list[3]
```

Listing 7.1: Building the NumPy array of filenames

The next step is to build our `Demo_Generator` objects with the previously created NumPy arrays of filenames and targets (see Listing 7.2).

```
1 my_training_batch_generator = Demo_Generator(
    X_train_filenames, y_train, batch_size)
2 my_validation_batch_generator = Demo_Generator(
    X_val_filenames, y_val, batch_size)
```

Listing 7.2: Creating a `Demo_Generator` class object with our training and validation datasets

Finally, we can build our model with all of our desired layers and features and fit our model to the training data (see Listing 7.3). Keras's `fit()` function requires various parameters when using a generator, such as `validation_steps` if a `validation_data`

argument parameter is present (see Keras's documentation for full usage). The most important component of the `fit()` function is the `history` object which it returns. The `history` object retains all the metrics associated with the model during the training process (e.g. loss and accuracy metrics for each epoch). Therefore, this object is essential in order to properly plot the network's performance and analyze it.

```
model = Sequential()
model.add(CuDNNLSTM(250, input_shape=(None, num_features))
    )
model.add(Dense(1, activation='sigmoid'))
adm = Adam(learning_rate=0.001)
model.compile(optimizer=adm, loss='binary_crossentropy',
    metrics=['acc'])
H = model.fit(x=my_training_batch_generator, epochs=
    num_epochs, validation_data=
    my_validation_batch_generator, verbose=2,
    validation_steps=int(num_val_demos // batch_size),
    steps_per_epoch=(num_demos // batch_size),
    max_queue_size=1)
```

Listing 7.3: Building our neural network

Overall, without the inclusion of any additional complex features, i.e regularization, dropout or recurrent dropout, and TBTT, our model is able to obtain approximately .87 training accuracy, .87 validation accuracy, .35 training loss, and .35 validation loss (see Figure 7.1).

7.0.1.1 L1 and L2 Regularizers

"Regularizers allow to apply [sic] penalties on layer parameters or layer activity during optimization. These penalties are incorporated in the loss function that the network optimizes" [8]. L1 regularization is called lasso regression, which stands for least absolute shrinkage and selection operator. The core idea of this method is the fact that it "shrinks some coefficients and sets others to 0, and hence tries to

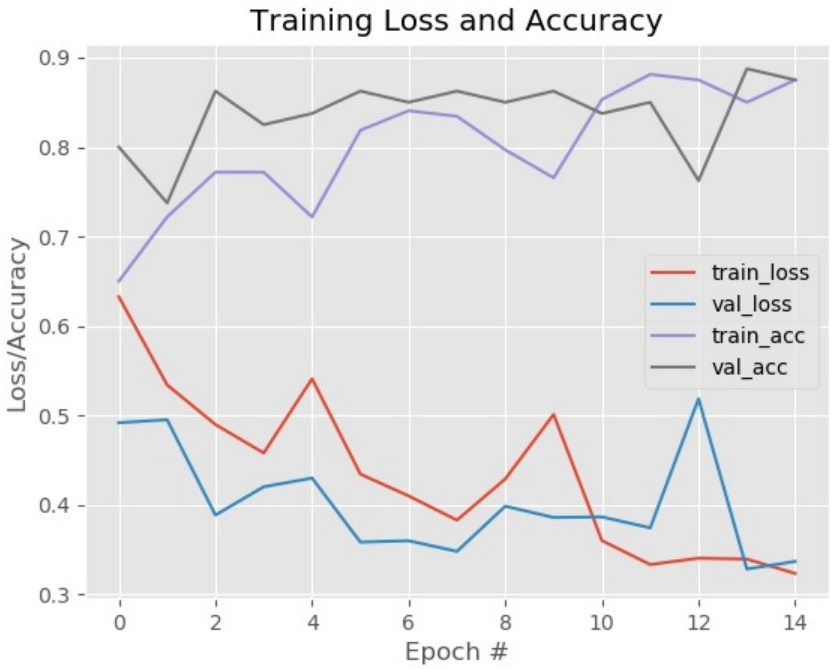

Figure 7.1: Plot 1 of network performance using the validation set approach

retain the good features of both subset selection and ridge regression" [22]. This functionally means that during the training process, penalties are imposed that result in the network dropping unimportant features (i.e. the idea of setting certain feature's coefficients to zero) [17]. This is an intuitive feature to add to our network due to the quantity of features that we have. Not all 228 of our features are most likely required for our network to determine whether someone is cheating or not. Therefore, removing the least significant features can allow our network to form a better and more comprehensive understanding of cheating.

L2 regularization is called ridge regression or better known as Tikhonov regularization. L2 regularization introduces a penalty term to the loss function that punishes the network for utilizing large coefficients on a specific feature (i.e. "encourages the sum of the squares of the parameters to be small" [17]). This forces the network to develop a coefficient and weighting scheme that spans multiple different

features. For example, imagine that there exists a subset of cheating data points within our dataset where a single feature's weight, for example, the z position of the 7^{th} player in the match (i.e. some random contrived feature that should not denote whether the player in question is cheating or not), results in the network correctly predicting that the player in question is cheating. Then, during the training process, the network heavily weighs this inane feature while leaving other features relatively low. In this example, the network heavily overfits since it is not properly developing weights and biases that generalize the problem of cheating. L2 regularization would incur massive penalties (in terms of the loss value) due to the network developing a heavy reliance on a single feature.

Overall, Figure 7.2 displays the network's associated training accuracy, validation accuracy, training loss, and validation loss over 4 training sessions with both L1 and L2 regularization. After 15 epochs, training accuracy approaches .9 while validation accuracy lags slightly behind at .85.

7.0.2 Subsampling and Truncated Backpropagation Through Time

Subsampling and Truncated Backpropagation Through time (TPTT) is a training approach whereby rather than training on individual sequences in their entirety, sequences are broken down into smaller subsequences and trained in small batches that iterate through each sequence. This method currently requires a large amount of overhead within Keras, therefore, it should typically be avoided unless deemed absolutely necessary.

This method entails the use of a rolling window, which is simply a subset of the input tensor that moves during each training batch. For example, with a window size of 20, sequences are broken down into sub-tensors of size 20 (which typically form a partition of the original tensor), and after the network analyzes a subset, it processes the subsequent 20-element window in the next batch.

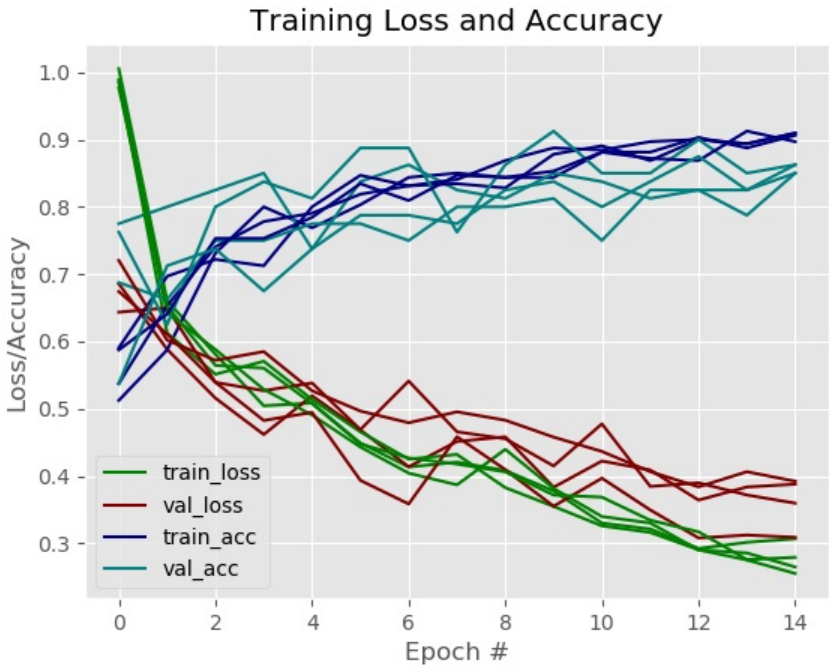

Figure 7.2: Compiled plot of predictive accuracy and loss over 4 training sessions

Additionally, this network architecture requires the use of stateful recurrent neural networks. The parameter `stateful` within Keras's `LSTM` layer is defined as, "Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state [sic] for the sample of index i in the following batch" [8]. This is an intuitive network construction due to the fact that subsampling is breaking down a single demo file sequence into a partition of smaller sequences. This means that there is a sense of continuity between batches, rather than treating each batch as an entirely separate data point.

Overall, while this method was tested, the immense amount of overhead required to implement this method stymied our ability to generate meaningful results. This most likely stems from implementation errors, rather than poor training results. However, this method could prove to be important in situations where sequences are too long for a network to produce meaningful results.

7.0.3 Condensed Sequences

Condensed sequences is an approach to avoid memory problems when attempting to use the typical scraped input tensor (which still tends to be quite large). This approach is defined by two basic components. First, tensor reduction, and subsequently training on the CPU. The former component entails the addition of a few lines of code within our `Demo_Generator` class (see Listing 7.4). First, we use a combination of NumPy's `round()` and `linspace()` functions to get evenly spaced indices across our sequence. The function `linspace` takes in as parameters `start`, `end`, `num`, and `dtype`. The parameters `start` and `end` specify the starting and ending indices of an array-like object, while `num` defines the desired number of samples to extract from the array-like object. Finally, `dtype` defines the type of the output array. This only yields evenly-spaced indices, therefore, following this, we have to declare and initialize a new truncated array, which we denote as `trunc_file`, with the previously found index values. Finally, we can append the truncated file `trunc_file` to our batch of truncated files `trunc_files`.

```
1 for file in loaded_files:
2     indices = np.round(np.linspace(0, file.shape[0] - 1,
          5000, dtype='int'))
3     trunc_file = file[indices]
4     trunc_files.append(trunc_file)
```

Listing 7.4: Grabbing evenly spaced indexes across the tensor

Additionally, since training is being done upon the CPU, we can employ dropout and recurrent dropout in order to get the best potential performance out of this network. This simply requires replacing our `CuDNNLSTM()` layer with `LSTM()` and adding arguments `dropout=.2` and `recurrent_dropout=.2`. Both of the dropout arguments take in a float from 0.0 to 1.0, which symbolizes the percentage of neurons

to be dropped during the course of training (e.g. .2 means 20% of neurons are dropped during training).

Next, this method boasts similar predictive accuracy and loss values to our L1 and L2 regularizer results with approximately .82 training accuracy, .83 validation accuracy, .33 training loss, and .33 validation loss. However, in general, the corresponding training and validation metrics (i.e. training accuracy and validation accuracy, and training loss and validation loss) tend to remain in close proximity throughout all epochs, meaning the deltas between the these metrics within the same epochs are quite small.

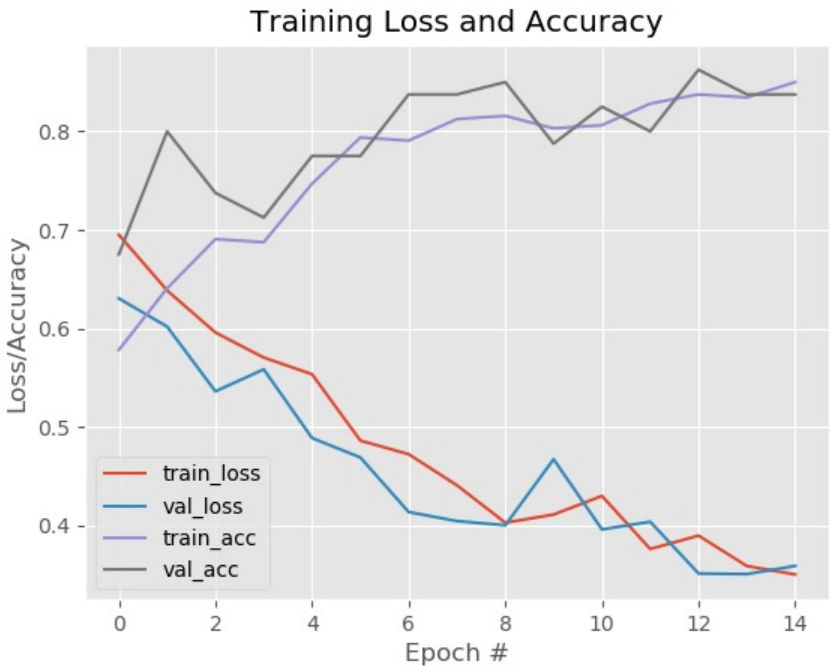

Figure 7.3: A CPU training session with dropout and recurrent dropout

Finally, analysis of individual sequences (i.e. individual matches) is quite simple with this testing method due to the size of the data points. Therefore, Figure 7.4 presents a match where the cheater toggles (i.e. enables) cheats halfway through the match. The cheater's team loses 7 of the first 9 rounds of the game, which leads to

them enabling all of their cheats on round 10. The cheater continues cheating until round 19, with his team winning every round since then. At the start of round 20, the opposing team calls a timeout, which within *Counter-Strike* halts the match for one minute, and subsequently decides to forfeit the match. The network's predicted probability of cheating mirrors the exact aforementioned events of the game. For the initial 1800 time steps, the networks predicted probability of cheating remains well below .4. While in the following 1000 time steps, the network's predicted probability of cheating skyrockets, nearly reaching a predicted probably of 1.0 (i.e. 100%) at around time step 2500. The probability of cheating remains quite volatile for the remainder of the match, but generally remains very high in comparison to the initial 1800 time steps. Finally, within the final 500 time steps, the probability stagnates and remains quite low, which is most likely due to the fact that the game is paused.

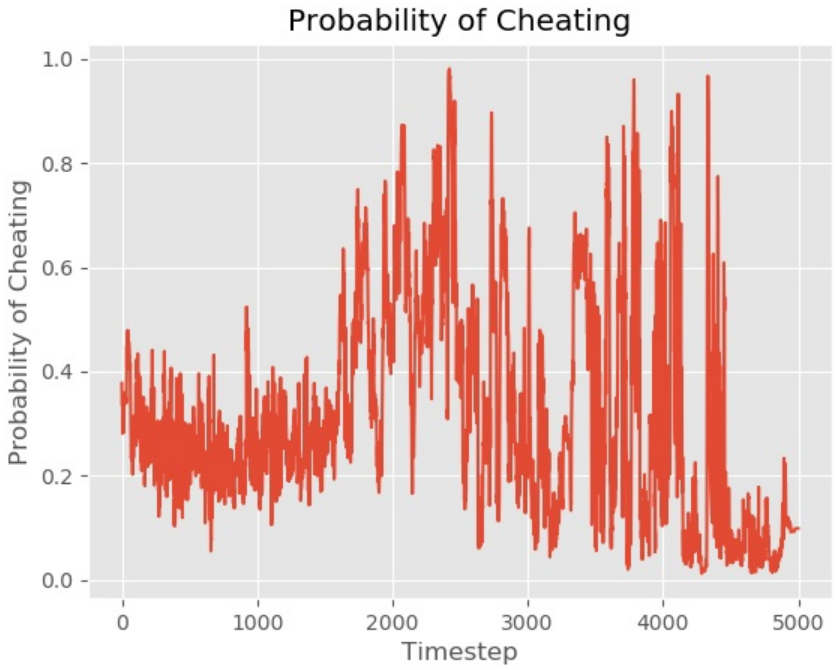

Figure 7.4: Network analysis of a cheater who enabled their cheats midway through the game

7.0.4 *k*-fold Cross Validation

"This approach involves randomly dividing the set of observations into k groups, or folds, of approximately equal size. The first fold is treated as a validation set, and the method is fit on the remaining $k - 1$ folds" [14]. Our implementation of k-fold cross-validation is based upon [3, 9]. This involves altering the basic network architecture in the following ways: First, we have to properly split the data into folds (see Listing 7.5), which is achieved through scikit-learn's `StratifiedKFold` function. Stratified k-fold cross-validation differs from the prototypical k-fold cross-validation method in that it "returns stratified folds. The folds are made by preserving the percentage of samples for each class" [9]. This means that each fold is balanced in terms of class representation. Next, we iterate through each fold that was returned from the `StratifiedKFold` function and declare and initialize the training generator and testing generator for each fold. Following this, we utilize the same network construction methods that were previously outlined. The final two steps that differ in this approach are how we evaluate the model on the testing generator. Keras's `evaluate` function "returns the loss value and metrics values for the model in test mode" [8], which allows us to analyze the network's performance on each subset that was held-out. Overall, this method attempts to avoid all of the previously mentioned weaknesses with the validation set approach as well as being well suited for smaller datasets.

```
1 kfold = StratifiedKFold(n_splits=10, shuffle=True,
     random_state=1)
2 cvscores = []
3 for train, test in kfold.split(X_train_filenames, y_train)
     :
4     my_training_batch_generator = Demo_Generator(
          X_train_filenames[train], y_train[train],
          batch_size, window_length)
```

```
5       my_testing_batch_generator = Demo_Generator(
            X_train_filenames[test], y_train[test], batch_size,
             window_length)
6       model = Sequential()
7       model.add(CuDNNLSTM(num_neurons, input_shape=(None,
            num_features)))
8       model.add(Dense(1, activation='sigmoid'))
9       adm = Adam(learning_rate=0.001)
10      model.compile(optimizer=adm,
11                    loss='binary_crossentropy',
12                    metrics=['acc'])
13      H = model.fit(x=my_training_batch_generator, epochs=
            num_epochs,
14                    verbose=0,
15                    steps_per_epoch=(num_demos - (num_demos
                          / 10)),
16                    max_queue_size=1, shuffle=False)
17      H_list.append(H)
18      scores = model.evaluate(my_testing_batch_generator,
            verbose=0)
19      print("%s: %.2f%%" % (model.metrics_names[1], scores
            [1]*100))
20      cvscores.append(scores[1] * 100)
21  print("%.2f%% (+/- %.2f%%)" % (np.mean(cvscores), np.std(
        cvscores)))
```

Listing 7.5: *k*-fold cross-validation implementation

k-fold cross-validation is a completely different testing and training scheme from the validation set approach, meaning that we can employ any or all of the previously mentioned supplementary testing methods on top of it in order to produce better results. Therefore, we opt to employ a combination of the condensed sequences approach and *k*-fold cross-validation due to the fact that this approach boasts efficient and short training times as well as strong results. One of the main drawbacks with *k*-fold cross-validation is that we are required to build and train multiple different networks in order to evaluate a network through *k*-fold cross-validation. This means that when $k = 10$, we have to build and train 10 networks in order to complete the testing process. This is equivalent to running 10 test runs using the validation set approach. Therefore, condensed sequences is the ideal

candidate for this testing method due to the speed with which we can perform this testing method.

The overall average across each fold equals 87.50% predictive accuracy upon the held-out sets with standard deviation equal to 4.74% for the initial 10-fold cross validation test. For the second 10-fold cross validation test, the average predictive accuracy on the test sets is 89.00% with standard deviation equal to 6.14% (see Tables 7.1 and 7.2).

Fold #	Test Accuracy
1	82.50
2	82.50
3	90.00
4	90.00
5	87.50
6	77.50
7	92.50
8	92.50
9	90.00
10	90.00

Table 7.1: First test of 10-fold cross validation test

Fold #	Test Accuracy
1	95.00
2	95.00
3	85.00
4	92.50
5	82.50
6	87.50
7	85.00
8	97.50
9	92.50
10	77.50

Table 7.2: Second test of 10-fold cross validation

Additionally, the network's training behavior mimics that of the validation testing approach (see Figure 7.5). The network typically attains a predictive accuracy between 90 to 95% on the training sets. The associated loss varies between .1 and .3. It appears that slight overfitting begins to occur around epochs 12 to 13 for some of the folds, which can be seen through the volatility of the plots in epochs after 13. However, this proved to not be detrimental to the testing results, therefore, the "early stop" training method, which is a method in which you halt training before convergence to combat overfitting, is not required.

Figure 7.5: Training results for each fold over 20 epochs, with blue being predictive accuracy and green being loss

CHAPTER 8

Conclusion

This project effectively sought and identifies a method for employing machine learning to tackle the task of cheat detection within *Counter-Strike*. The solution is rooted within two components, a network structure that imitates the natural way with which humans review demo files (i.e. recurrent neural networks) and designing metrics/features that can be used to properly classify cheating. The predictive accuracy and associated loss for the trained network varies depending on the training methods that are employed. Nevertheless, baseline accuracy varies between 80-90% with little to no overfitting. This thus demonstrates an easily scaleable and deployable method of combatting cheating within any video game.

8.1 Errors and Improvements

There are many things that are present and ingrained within this project that complicate or impede our ability to improve various components of this project. These include the original labeling scheme, data collection techniques, and hardware limitations.

8.1.1 Labeling Data

One of the earliest decisions that this project is founded upon is the labeling scheme. This project adopts a very simplistic labeling scheme whereby -1 symbolizes that

there is no cheating within this data point (i.e. this is the quintessential no-cheating label). Next, 0 means that this data point should not be used because there is no conclusive evidence in regards to whether there is cheating or not. 1 is the catch-all cheating label, meaning that this data point has cheating within it beyond a reasonable doubt. Finally, 2 symbolizes that the data point seems to have cheating within it, but it is not beyond a reasonable doubt.

Originally, these labels were sufficient, since these adhere to the basic goal of developing an algorithm that could identify whether there is cheating as a simple binary decision (i.e. yes or no in regards to the existence of cheating within a data point). However, as the number of data points increased, it became more and more apparent that there is a single specific type of cheat that presents itself more often than all the other forms of cheating, which is "spinbotting." Therefore, to circumvent the potential over-saturation of a single type of cheat within the dataset, the new label 3 was added, which is responsible for symbolizing that that data point is "spinbotting."

The main benefit of this scheme is that there is no requirement of knowing what specific type of cheat is present within each individual demo. This is beneficial since at the onset of the project, the entire hypothesis space (i.e. what cheating looks like, and what types are present) was unknown. However, the main consequence of this scheme is our lack of ability to determine which forms of cheating the network is efficient at identifying and which forms the network is inefficient at identifying. For example, the network may be very inefficient at identifying cheaters who only employ wall hacks, since there are fewer metrics designed to target that specific type of cheat. Despite of all this, developing individual labels and testing them individually would only produce meaningful results should the dataset contain an ample number of data-points per label. This is not an easy task, since some types of cheats are very rare or more experimental. Additionally, this presupposes a sort of

omniscience when it comes to the ability to identify which type of cheat is present within the demo file, since it is very difficult to identify the exact type of cheat/cheats being employed by a cheater. Therefore, while a more complex labeling scheme would be ideal given a massive dataset and easier means of identifying cheats, the simplistic labeling scheme that is currently employed is sufficient for the current needs of the project.

8.1.2 Optimizing Data Collection

Throughout the development process of this project, many ways of ameliorating and improving the speed and efficiency of the data collection methods were found. First, the hypothesis space became better understood as the project progressed, meaning that all the various types of cheats became easier to identify as the amount of exposure to those cheats increased. This means that less time is required per Overwatch case, which allows the reviewer to iterate through more cases in a more efficient manner.

Additionally, `takeshixx`'s code base underwent many quality-of-life changes that significantly improved our ability to interact with the demo files from Overwatch cases. This initially entailed improvements to the parser to allay the difficulties associated with correlating the anonymized players in the Overwatch case with their non-anonymized counterparts. However, at this project's climax, its functionality and subsequent workflow for classifying and labeling data transformed to better accommodate speedy data collection. Overall, in terms of the total quantity of data collected within the scope of this project, the majority of files skew towards being collected in the latter half of this project's timeline due to the optimizations that were made at various parts of the project. Therefore, this point mostly speaks upon the lack of documentation and information about the major cheating methods within *Counter-Strike*, since if more research and information had been present at the

onset of this project, these optimizations could have occurred much sooner within this project's timeline.

8.1.3 Automating Data Collection and Training

One key feature that could improve overall network efficiency is increasing the number of demo files. Deep learning tasks typically become easier in some regards as the quantity of data increases. Obviously there is an upper bound to the potential gains from metrics such as prediction accuracy and loss as the the quantity of data points increase, but the main idea is the fact that overfitting becomes substantially harder once the dataset reaches a certain threshold. This stems from the fact that as a dataset gets larger, the individual nuances that can be present within specific/individual data points become less apparent and important. For example, if this project uses cheating files that feature cheaters who only play in a specific manner, our network would most likely overfit to account for that. This would, in turn, demonstrate that the network is not able to generalize what cheating is and instead is only able to identify a specific type of cheat. This lends itself towards the basic deep learning philosophy that the more data a deep learning project has, the more likely it is able to succeed in its goals, since the dataset becomes more representative of the problem as a whole.

Additionally, one of the greatest ways of expediting the data collection process is to remove any human components of it (i.e. complete automation). This approach would require our neural network to attain an above average prediction accuracy on untrained data points (i.e. be decently proficient at generalizing its conception of cheating), since it would take the role of reviewing Overwatch cases. This would mean that the network would be labeling its own data points to use for later training. While this could seem inherently disastrous, since if the network does not have a

sufficiently high prediction accuracy (i.e. sub 80%) it could heavily mislabel data. However, the automation process can be broken down into two phases.

First, automation would take place once the network achieves a baseline predictive accuracy of 60%. This is slightly above the network randomly guessing, but it would still indicate that it achieves some basic understanding of cheating. Thus, we would first employ scripts to automatically grab demo files from new Overwatch cases. Next, we would properly preprocess the demo and give it to the network for a prediction. Specifically, this would entail the use of Keras's `predict()` function, which feeds a data point into a network and returns the predicted label. This means that the internal weights and biases of the network are not altered. We then use the predicted label to determine our verdict for the associated Overwatch case (i.e. whether the Overwatch case presented a cheater or not). This automation system is immensely beneficial and ideal for a multitude of reasons. First, Overwatch favors Overwatch agents who submit large quantities of verdicts as well as those who are the most accurate with their verdicts. "Favor" within this context means that Overwatch tends to send more Overwatch cases to the agents that it favors. Therefore, we can significantly increase the amount of demo files within our dataset by developing an automated system that can always respond to Overwatch cases.

One of Overwatch's main caveats is the fact that there are often large deltas in the amount of time between the reception of Overwatch cases. This delta can be on the scale of minutes or hours. Therefore, data collection tends to be inconsistent, however, this can be remedied through our use of an automated Overwatch system. However, this initial stage of automation is only attempting to bridge the gap (i.e. deltas) between demo files, since our network is unreliable at this stage (i.e. only boasts a 60% or more predictive accuracy). This unreliability means that demo files would still need to be hand labeled and viewed, in order to actually be used within the training dataset.

Stage two of automation would entail having a network with a minimum predictive accuracy of 80% (ideally this would be 90%). At this stage of automation, the network is proficient and reliable enough to completely remove all human aspects of training. This would mean that the network would label the Overwatch cases as it sees fit without any human intervention. The main caveat with this approach is that the network would most likely fail to improve itself and stagnate at its initial accuracy value, just due to the basic design architecture in this approach. However, this multistage automation process would allow this project to attain a massive dataset in a relatively short amount of time while minimizing the overall human cost within the project.

8.1.4 Hardware Limitations

One of the major limitations that this project faces is hardware, specifically memory limitations. Before the various modifications to the library that builds and compiles `demoinfogo.exe`, each data point within the project was quite small, since each only contained game events rather than having both game events and Entity Delta updates. Therefore, large portions of the dataset could be loaded into memory for testing and training at this project's onset. However, after the inclusion of Entity Delta updates within the data points, the size of the dataset exploded, making it impossible to load it into memory. However, even after methods such as batch generators were employed, the network architecture (e.g. number of neurons) was always constrained by the amount of available memory. Therefore, this project would have benefited from larger-scale enterprise-level systems that circumvent all the hardware limitations. Nevertheless, these limitations proved to not completely impede this project's ability to attain tangible results, however, there is potential for performance gains (i.e. in predictive accuracy) if the network architecture is not constrained.

8.1.5 New and Unique Cheats

During the development of this project, three new cheats that fall slightly outside the typical cheating "families" were discovered, these being: "anti-aim," "backtracking," and deep learning based cheats. The former cheat method is a technique that desynchronizes the visible part of a player model and the hittable portion of a player model (called the hit-box). This means that while a player may appear to occupy a certain space within the game environment, they are instead occupying a different space. The difference between the real space and the fake space is typically seen solely through a rotation of the player model (see Figure 8.1).

Figure 8.1: Demonstration of "anti-aim," where green denotes the actual hit-box and red denotes what other players can see [1].

[1]This source is available from the author upon request; it is redacted from this paper due to concerns about raising awareness of illicit or otherwise dishonest content.

Additionally, "backtracking" mimics the behavior resulting from the online multiplayer phenomenon of rubberbanding, which is a phenomenon where players within the game environment appear to move in twitchy, spasmodic, and shaky manners. This is typically accredited to poor internet connection between that specific client and the server. This behavior also results in discrepancies between where that player thinks they are in the game environment and where they actually are. This allows other players to seemingly spot and attack them while they are not physically visible. For example, perhaps a player ducks behind a corner, however, due to rubberbanding, their real player model still appears out in the open and away from the corner, which allows other players to easily spot them and attack them.

"Backtracking" removes the requirement of a poor internet connection and, additionally, allows other players to force the rubberbanding behavior upon their enemies. This means that cheaters who employ this method can seemingly see and attack enemy players in places that they previously occupied (e.g. the place before the player hid behind a corner), thereby switching the basic dynamic of rubberbanding (see Figure 8.2).

Finally, there are numerous deep learning based cheating methods that are growing in popularity due to their inherent characteristic of being hard to detect. These methods are trained using the play styles and aiming strategies employed by real players. These methods are very deceiving and could pose a large threat to *Counter-Strike*. However, these still appear to be in their infancy and in the development phase, which means that they are not as common place within the cheating community as "WallHack" and "AimBot."

Overall, these cheats were rarely ever encountered during data collection (if at all), meaning that developing specific metrics to target these cheating methods would not yield large benefits in terms of predictive accuracy. Additionally, more

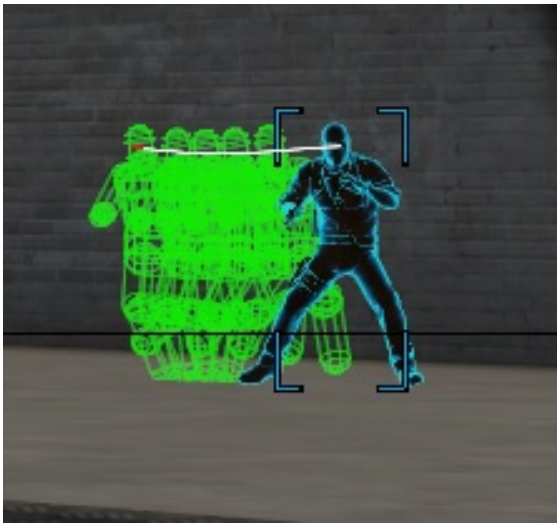

Figure 8.2: Demonstration of "backtracking," where green denotes places where the player previously occupied and blue denotes where the player is currently occupying [2].

research would have been required in order to determine whether these types of cheats present themselves in meaningful ways within demo files.

8.1.6 This Project's Cheats

This project also developed cheats in order to understand the fundamentals of many of the cheats that are present within *Counter-Strike* as well as use them as a potential source for data. However, there was little need to use these cheats for our own dataset, since we were able to achieve meaningful results without them. Additionally, there are some potential caveats associated with the use of our own personal cheats that could have presented themselves within the dataset. First, most of these self-sourced cheating demo files would predominately feature bots versus a select number of real players. This type of scenario is not representative of the typical match within *Counter-Strike*; therefore, using these within our dataset could result in the network learning how to distinguish between bot matches and non-bot

[2]This source is available from the author upon request; it is redacted from this paper due to concerns about raising awareness of illicit or otherwise dishonest content.

matches rather than distinguishing between cheating matches and non-cheating matches. However, the development of these cheats proved to be essential for the crafting of metrics, since they allowed us to get a better understanding of how cheats would present themselves within demo files. Therefore, we would have used our own cheating methods to source data had it been easier to source an ample number of willing participants to replace the bots within these matches.

8.1.7 Normalization

Normalization is one component of this project that could have undergone a more rigorous testing regime. This project adopted the use of the most straightforward normalization approach by normalizing all values within each feature within each demo file to fall between 0 and 1. However, one normalization scheme that could improve network performance is normalizing the same features across all the different players in conjunction (typically the player performance features). For example, one data point within this project boasts a cheater who finishes the match with approximately 70 kills (an inhuman feat), while every other player within the game remains at approximately 0 kills and 0 average damage per round. All of the cheater's performance features swamp all of those of the other players within the match. However, with normalization reducing all of the players performance features to fall between 0 and 1, the magnitude of this discrepancy in performance between players is lost. Therefore, the use of either a normalization scheme that normalizes all of the same features across all the players or the development of new comparative performance features could prove to improve the network performance.

8.2 Future Work Within the Field

Machine learning as a means of combating cheating within games boasts significant promise for the future of anti-cheat methods. Not only does it circumvent all the associated problems with current and contemporary anti-cheats, such as relying on memory surveillance, but it also can be used in parallel with contemporary anti-cheat measures to further strengthen a game's overall security. For the majority of games, this responsibility falls upon the game creators rather than third party services or entities, since data collection and preparation is far more easily attained by the developers who maintain the game. This project is only possible for the most part due to the Overwatch system and the abundance of demo files within the sphere of the *Counter-Strike* community. This means that for other third-party entities who wish to employ similar anti-cheat tactics within their game, data must generally be easy to obtain and process with little overhead. Nevertheless, the methods outlined within this paper are applicable to any anti-cheating project for any game. The main stipulations are that network construction needs to be tailored to properly account for the format of the data, and that individual cheating metrics must be identified and crafted in order to allow the network to properly identify cheating.

Additionally, third-party matchmaking services for *Counter-Strike* have already began implementing anti-cheat methods based upon this approach. As of late 2019, third-party matchmaking service *FaceIt* announced its use of HestiaNet, which is a neural network that follows a very similar architectural scheme as the one presented within this paper. Therefore, it is easy to see that there is a slow trend towards the adoption of this type of anti-cheat system to combat cheating within *Counter-Strike*.

References

1. D. Botvich, L. Chapel, and D. Malone. Probabilisitc Approaches to Cheating Detection in Online Games. *2010 IEEE Conference on Computational Intelligence and Games*, CIG'10:1–7, 2010.

2. P. Brooke, H. Chivers, P. Laurens, and R. Paige. A Novel Approach to the Detection of Cheating in Multiplayer Online Games. *12th IEEE International Conference on Engineering Complex Computer Systems*, ICECCS 2007:1–11, 2007.

3. J. Brownlee. A Gentle Introduction to k-fold Cross-Validation. `https://machinelearningmastery.com/k-fold-cross-validation/`, 2019.

4. L. Cardamone, L. Galli, P. Lanzi, and D. Loiacono. A Cheating Detection Framework for Unreal Tournament III: a Machine Learning Approach. *2011 IEEE Conference on Computational Intelligence and Games*, CIG'11:266–272, 2011.

5. C. Chambers, W.C. Feng, and D. Saha. Mitigating Information Exposure to Cheaters in Real-Time Strategy Games. *Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV'05:7–12, 2005.

6. H.C. Chang, K.T. Chen, and H.K. Pao. Game Bot Identification Based on Manifold Learning. pages 21–26, 2008.

7. F. Chollet. *Deep Learning with Python*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017. ISBN 1617294438, 9781617294433.

8. F. Chollet et al. Keras. `https://keras.io`, 2015.

9. D. Cournapeau et al. scikit-learn's Stratified K-Fold cross-validator. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html`, 2007.

10. B. Fortuner, M. Viana, and B. Kowshik. Loss functions, 2019. `https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html`.

11. F. Frangoudes, A. Hashem, and C. Neuman. Behavioral-Based Cheating Detection in Online First Person Shooters Using Machine Learning Techniques. *2013 IEEE Conference on Computational Intelligence and Games*, CIG'13:1–8, 2013.

12. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

13. injx. Strafing theory. https://web.archive.org/web/20150508100656/http://www.funender.com/quake/articles/strafing_theory.html, 2008. Accessed: 2019-12-18.

14. G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

15. D. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.

16. C.S. Lui, J. Lui, J. Yan, and S.F. Yeung. Dynamic Bayesian Approach for Detecting Cheats in Multi-Player Online Games. *Multimedia Systems*, 14.4: 221–236, 2008.

17. A. Ng. Feature selection, L1 vs. L2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78, 2004.

18. M. Nielsen. *Neural Networks and Deep Learning*. http://neuralnetworksanddeeplearning.com, 2019.

19. NVIDIA. NVIDIA cuDNN. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html, 2014.

20. J. Patterson and A. Gibson. *Deep learning: A practitioner's approach*. " O'Reilly Media, Inc.", 2017.

21. B. Randell and J. Yan. A Systematic Classification of Cheating in Online Games. *Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV'04: 1–9, 2004.

22. T. Robert. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL http://www.jstor.org/stable/2346178.

23. A. Rosebrock. *Deep Learning for Computer Vision with Python*, pages 21–168. PyImageSearch, 06 2018.

24. M. Sicart. Defining game mechanics. *Game Studies*, 8(2):n, 2008.

25. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

26. takeshixx. csgo-overwatcher repository. `https://github.com/takeshixx/csgo-overwatcher`, 2018.

27. TensorFlow. Variable Length Sequence Support for CUDNN LSTM. `https://github.com/tensorflow/tensorflow/issues/23269`, 2018.

28. TensorFlow. TensorFlow GPU support documentation. https://www.tensorflow.org/install/gpu, 2020.

29. Valve Software. demoninfogo repository. `https://github.com/ValveSoftware/csgo-demoinfo/tree/master/demoinfogo`, 2014.

30. Z. Zhang, L. Ma, Z. Li, and C. Wu. Normalized Direction-preserving Adam, 2017.