Bot Detection in Online Games through Applied Machine Learning and Statistical Analysis of Mouse Movements

Gavin McCracken
School of Computer Science, McGill University
December 2018

Abstract

The video game industry is in a constant arms race against cheat developers that profit off writing code to give players an unfair advantage in online gameplay against other players. As soon as a company releases an update for their cheat detection system, cheat developers swiftly reverse engineer it and find ways to update their cheating software to avoid detection. In recent years there has been some research into detecting cheaters by using machine learning techniques on game-metrics, for example, head shot accuracy, bullets missed, etc. This work looks instead at mouse inputs to the game, as it's easy to imagine that constructing an algorithm that generates human mouse movements is not a trivial task. A large dataset of mouse events was recorded by humans playing an online game called Runescape and the mouse data was analyzed statistically in the hopes of finding metrics to detect cheaters. These metrics were then used with the Kolmogorov-Smirnov test to check whether the human data had a similar distribution to the bot data. Simple feed forward neural networks with only one hidden layer were able to detect all bots that they were tested against in this paper. Even the more advanced cubic mouse movement bot that had its movement modeled based on the analysis of human mouse movements was detected.

Keywords

Bot Detection, Online Games, Machine Learning

1 Introduction

The video game industry has had its market share sky-rocket since its explosion in popularity from the introduction of online gameplay [1]. In fact, companies like Blizzard Entertainment who only make online games, are placed every year in the top 10 gaming companies by revenue [2]. A central aspect to the enjoyment of online game play is however, fairness. Naturally, if people attempt to play a game and other players have methods of circumnavigating the rules to gain an unfair advantage, genuine players will feel discouraged to continue playing. In addition to affecting the ability of legitimate players to have fun, the surge in gaming competitions that carry huge monetary lump sums as prizes, have required that tournament organizers can ensure competing players aren't cheating. It has thus been essential for games to be released with anti-cheat software that is capable of detecting if a player is using cheating software.

Examples of cheating software can range from things like automated bots that can "farm" resources in an MMO by gathering them for hours on end, hacks that allow a player to see other players through walls, or aim assistance such as aiding a cheating player with things like shooting, scoring on a goal, etc. In the case of MMOs, bot programs can drive in-game resource values down, or even devalue the currency used to trade with other players in the game.

An example of anti-cheat software is the Valve Anti-Cheat (VAC) system, which is an anti-cheat service provided to most games on Valve's Steam gaming platform. It functions to remove players who cheat online in games by "banning" them from the game forever. It is however, an anti-cheat designed to absolutely minimize false positives (banning humans by mistake). Thus it only bans cheaters that have malicious code running on their computer that VAC matches to it's database of cheats. This means that the VAC developers have to take time to figure out how cheats appear in the RAM, and then update their database with key function signatures contained in the cheats. In other words, VAC is looking for an exact piece of code. If it finds one, then it knows that person is cheating and bans them accordingly. It is widely speculated that Blizzard Entertainment use

some sort of anti-cheat that analyzes game play metrics, which are things like: head-shot %, % bullets missed, etc, and apply machine learning techniques to the metrics.

We have looked into the feasibility of identifying cheating software by looking at mouse movements. Since most malicious software has to interact with the game in order to actually do things and provide a benefit to the cheater, it has to be able to move the mouse - or at least click on objects. This is done by generating a path and moving the mouse cursor along it. We suspect that since human movements are hard to accurately generate, it's possible to distinguish artificial paths from human paths. In fact, many modern bots use cubic splines to generate a path upon which to move the mouse. Additionally, humans can randomly move the mouse to points that are inefficient and serve no purpose. This can happen for many reasons, such as when they are distracted, talking to someone, stretching or many more. These movements are hard to include in a bot, and also, any movements generated by cubic splines will also be smooth, whereas human movements will be shaky.

Mouse movements were recorded from humans playing Runescape and analyzed in depth. Data specific to certain activities, as well as data from general gameplay was studied to devise possible metrics for mouse movements that can differentiate between bots and humans.

After this analysis, we used Tensorflow to train feed forward neural networks to classify whether data was coming from bots or humans. These neural networks had only one hidden layer and performed quite well. The best performing neural network however, was able to even distinguish our best attempt at making bot mouse movements appear human, with quite high accuracy.

1.1 Related Work

Performing a Google Scholar search for relevant work reveals that machine learning algorithms are currently an underutilized tool for detecting cheaters in online games. In fact, only a handful of works appear to have been published, and they mostly cover a wide range of non-mouse-movement metrics such as player avatar movement or internal game metrics, e.g. aim, accuracy, etc [3–5]. Very few papers have actually taken mouse movements into account in the context of cheating in games, such as the work by Pao et al. focusing on general user input as a verification technique [6] and research by Kaminsky et al. on using movements from Starcraft and Solitaire to uniquely identify users [7].

2 Approach

2.1 Building the Dataset

Mouse movements were recorded by writing both a Windows hook and Linux hook that listened and recorded all mouse events that the operating system processed. These mouse movements were recorded while humans were playing an MMO called Runescape and performing multiple in-game activities. There are five events that occur during Runescape game-play: a movement - which is anytime the cursor moves to a new pixel, a LEFT_CLICK_DOWN, a LEFT_CLICK_UP, a RIGHT_CLICK_DOWN, or a RIGHT_CLICK_UP. Every time the operating system registered one of these five events, the hook recorded the event type, as well as the current x and y coordinates of the mouse cursor on the screen, and the time that the event occurred. The raw data was then parsed into three types of movements as follows:

- Movement: the mouse was moved and at least 10 events were observed, then a pause of at least 0.3 seconds occurred.
- Movement-and-click: the mouse was moved and either a left or right click-down event occurred, followed by the corresponding left or right click-up event.
- Drag-and-drop: either a left or right click-down event was observed, and then at least 0.3 seconds passed before a click-up event was observed.

To compare with the human mouse-movements, four programs were written to artificially generate mouse-movements and create data. Three of these programs were naive, while one was more advanced. The advanced one had been designed using the previously discussed metrics and analysis in such a way that it was indistinguishable from human data when analyzed by those metrics. This means that it would have the same distribution for it's metrics as human data. The bot programs are defined below in a list:

- The first would teleport the mouse from its current location to the location it wished to hover over, click-on or drag the mouse to.
- The second would move the mouse in a perfect line to the target.
- The third was a naive implementation of cubic splines; two cubic splines were generated and put together for each mouse movement of the botted activity. The four polynomial coefficients, (a, b, c, d) in the cubic equation ax^3+bx^2+cx+d, were calculated using the start velocity, start position, end velocity and end position sampled from a human mouse movement. The simulated curves

were then evaluated as:
$(x, y) = (spline_x(time), spline_y(time))$
where the spline was determined depending on what mouse movement the simulation was currently in.

- The fourth was generating non-naive cubic curves. These curves were fitted to have their velocities, angles, and maximum deviation come from the same distribution as human data. Given the start and end point of a mouse movement, points 20% into the movement and 80% into the movement were approximated, and the four polynomial coefficients were fitted to run through these points.

2.2 Metrics and Features

Generally, raw data can not just be fed into a neural network. In order to figure out what kind of features to feed into the neural net, some statistical analysis was done, examining various metrics in the data to determine the difference between bot and human data. The metrics a neural network was trained on are:

- Velocity: pixels per second (px/s)
- Acceleration: pixels per second per second (px/s^2)
- Maximum-Deviation: the farthest distance the mouse goes from the line defined by the start point and end point of the mouse-movement
- Angle: the angle, θ, of the mouse movement from the start point to the finish point.
- Velocity-Angle: the velocity and the angle relative to the x axis that the velocity occurred on.

Velocities and accelerations were calculated using the central difference with 25 pixels. This is because as the mouse moves, each event has coordinates that are only slightly shifted from the last events coordinates. This is because the CPU polls a modern gaming mouse around 1000 times per second. The central difference is given below:

$$\delta_h[f](x) = f(x + \frac{1}{2}h) - f(x - \frac{1}{2}h)$$

Angles were calculated using atan2. Maximum deviation was calculated by iterating through all events in the mouse movement. Starting at the start event point, the distance of each events x and y coordinates, to the line travelling through the start and end point was computed. With $P_1 = (x_1, y_1)$ being the start point and $P_2 = (x_2, y_2)$ being the end point, and x_i, y_i being the coordinates of the third point, The distance to the line (P_1, P_2) is calculated by:

$$D = \frac{|(y_2 - y_1)x_i - (x_2 - x_1)y_i + x_2y_1 - y_2x_1|}{\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}}$$

and the max deviation is:

$$arg\max_{x_i, y_i} D$$

The data was then analyzed by plotting it into histograms with a Gaussian smoothing operator to remove noise. The amount of smoothing was specified by using a particular value for full width half maximum (FWHM). If the considered function is the density of a normal distribution of the form

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x - x_0)^2}{2\sigma^2}\right]$$

where σ is the standard deviation and x_0 is the expected value, then the relationship between FWHM and the standard deviation is $\text{FWHM} = 2\sqrt{2\ln 2}\,\sigma$.

Initially, the Kolmogorov–Smirnov test was used to come to the conclusion of whether or not it was possible to tell the difference between human and algorithmically generated mouse movements for a given metric. To do this, first the empirical distribution functions F_n for n i.i.d. observations X_i was calculated as follows:

$$F_n(x) = \frac{1}{n}\sum_{i}^{n} I_{X_i \in [\infty, x]}(X_i)$$

where $I_{[-\infty,x]}(X_i)$ is the indicator function, equal to 1 if $X_i \leq xX_i$ and equal to 0 otherwise. Then the Kolmogorov–Smirnov statistic for the calculated empirical cumulative distribution function F(x) is computed by first calculating the maximum distance from the empirical cumulative distribution function (ECDF) to the "analytical" cumulative distribution function (CDF) that we are comparing to,

$$\text{MaxDistance} = \sup_x |F(x) - F_{\text{analytical}}(x)|$$

The confidence level in rejecting our null hypothesis is then calculated via a bisection search such that it is the largest value satisfying the equation

$$\text{MaxDistance} > c(\alpha) * \sqrt{\frac{1}{n}}$$

where n is the number of samples and $c(\alpha)$ is calculated such that it satisfies

$$\Pr(K \leq c(\alpha)) = 1 - \alpha$$

via another application of the bisection method. Finally, $\Pr(K \leq x)$ is the CDF of the Kolmogorov distribution, calculated here as

$$\Pr(K \leq x) = \frac{\sqrt{2\pi}}{x} \sum_{k=1}^{10000} e^{-(2k-1)^2\pi^2/(8x^2)}$$

2.3 Neural Network Creation

Tensorflow was used to create four feed-forward neural networks, one for each type of artificial mouse movement generation, and they were trained on human data and one of the four artificially generated datasets. Sparse categorical cross entropy was used and humans were classified as 0, and bots were classified as 1. The input layer took the 5 metrics as input, and fed them into a dense hidden layer using a rectified linear activation function, which then was fed into a Dense output layer with 2 nodes and a softmax activation function. An extra node was added to both the input and hidden layers to allow for biasing. The number of nodes in the hidden layer was changed throughout testing to observe the effects such changes had on validation accuracy and false positives. Dropout was used to prevent overfitting, and the neural networks were tested on data they had never seen before.

After this, it was seen that the movements generated by the advanced cubic were capable of tricking the neural network that was trained on the metrics we chose. As such, another neural network was trained. This one took mouse-movements that had their start points translated to the origin (x=0, y=0), and all other points translated accordingly. The input layer received the events that a mouse-movement was composed of in the form of (time, x, y), and took 40 events, so 120 total inputs per mouse movement. To ensure that both human and bot mouse movements had the same number of events and thus points, the position of x and y coordinates were linearly interpolated by using the points nearest to them. Now, since the human and advanced cubic movements had the same number of points, they were fed into the input layer. Once again, the architecture was: bias nodes were added, dropout was used, a dense hidden layer with a rectified linear activation function, and a dense softmax output layer, and with humans being classified as 0 and bots as 1. The number of nodes in the hidden layer was changed to observe the effects such changes had on validation accuracy and false positives and the neural networks were tested on data they had never seen before.

3 Analysis & Results

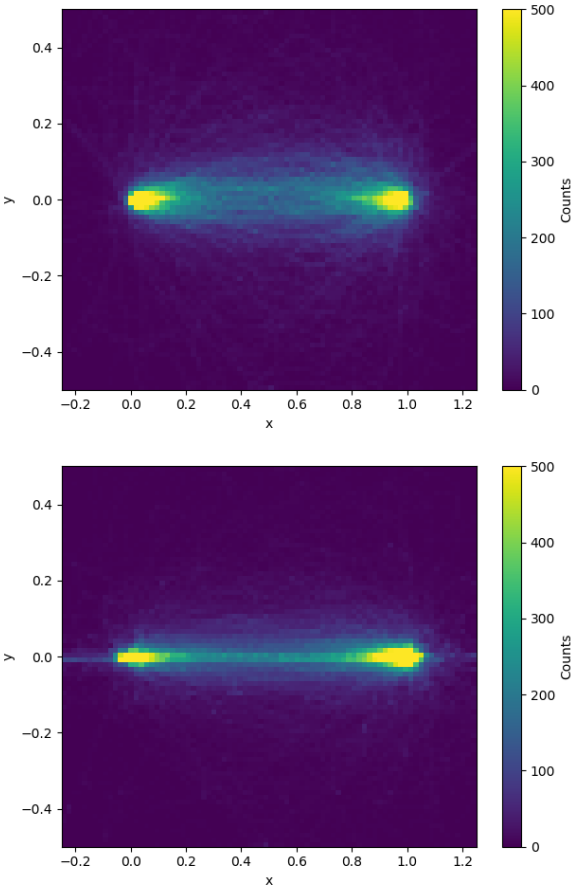

Figure 1: Normalized (rotated and scaled) mouse movement histogram for human data (top) and advanced cubic bot (bottom).

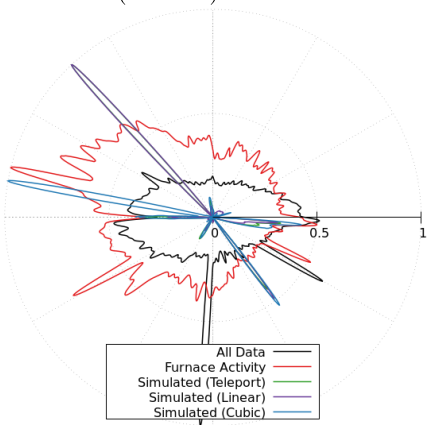

Figure 2: Polar plot of the distribution of mouse velocities. The "All data" and "Furnace Activity" data both come from humans (FWHM = 0.05).

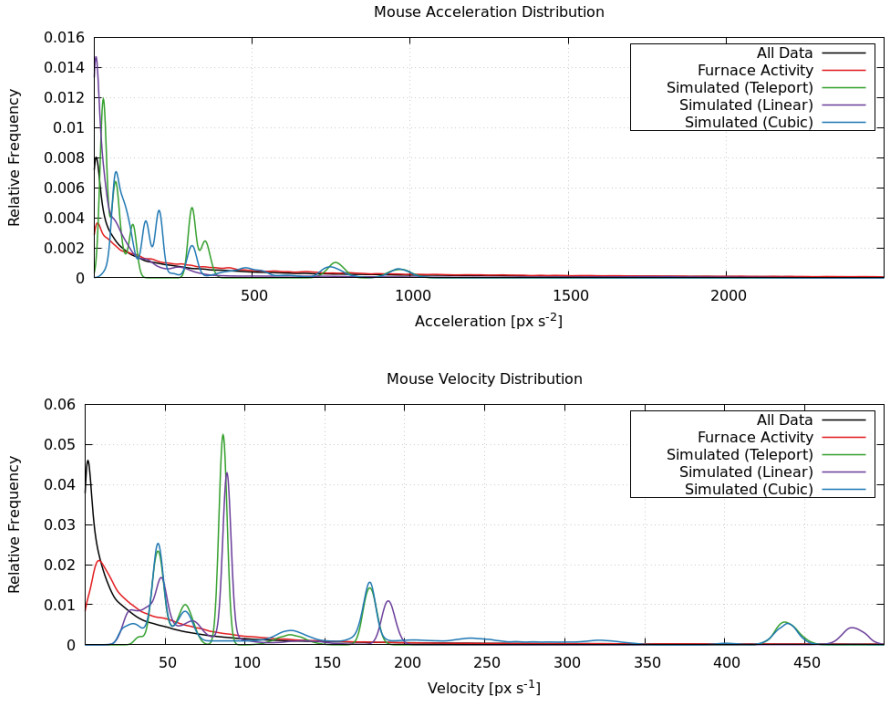

Figure 3: Distribution of mouse accelerations (FWHM = 25), and distribution of mouse velocities. It is observed that none of the three naive generation methods were close to human-like movement (FWHM = 5).

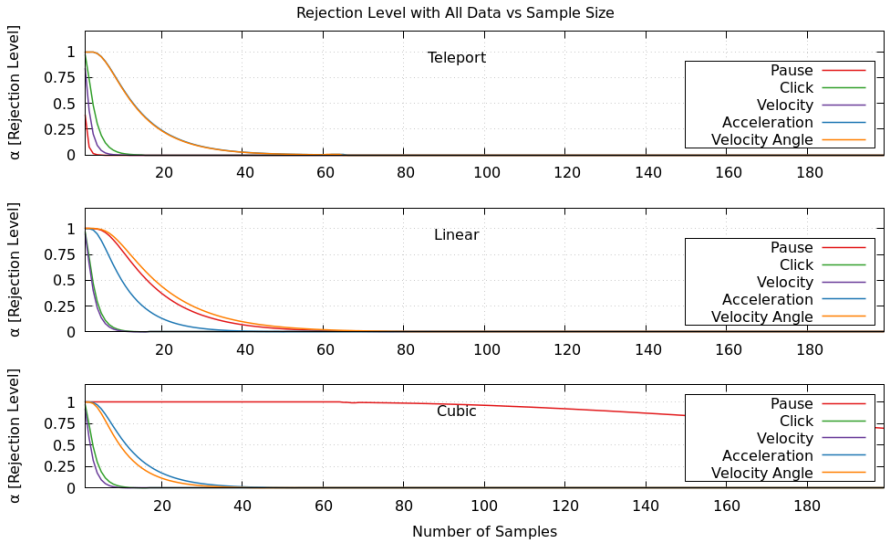

Figure 4: Plot of α for the Kolmogorov-Smirnov test vs sample size. The "analytical distribution" was trained from all data. An alpha value of 1 means that the data matches and is from the same distribution, an alpha value of 0 means that the two are from different distributions. The only metric that isn't statistically different from human mouse-movement is cubic pause-time.

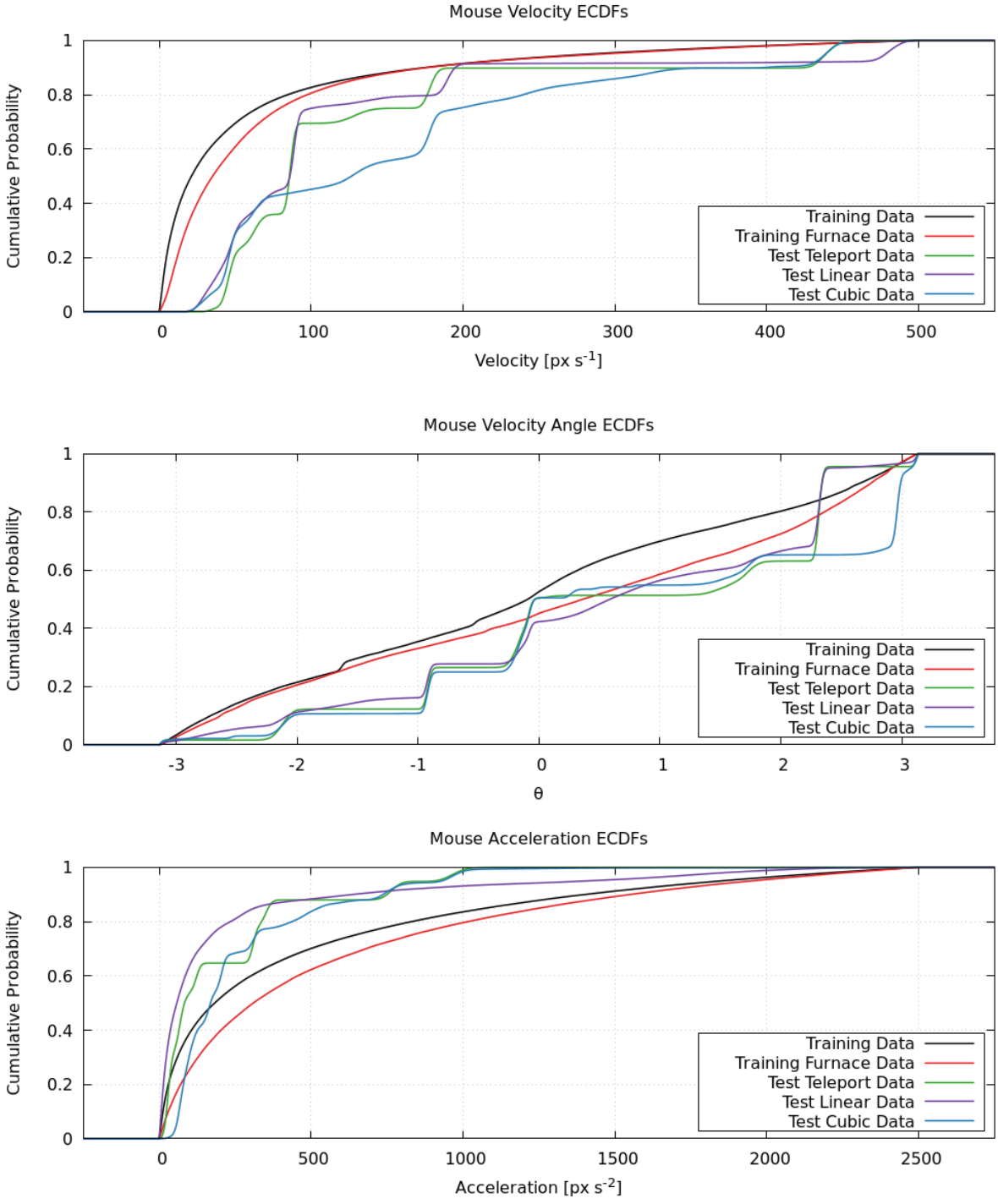

Figure 5: ECDFs for mouse velocity, mouse velocity angle, and mouse acceleration. All three of these plots confirm that the naive simulations differ substantially from human movement.

3.1 Statistics

The plots of the human data provided great insight into what mouse movements look like. In fact, the velangle plot, - the distribution of velocities and their angle relative to the x axis that the velocity occurred on - was very intuitive, Figure 2. It showed for an average playing session on Runescape, horizontal movements are more common than vertical. This makes sense as Runescape plays on an 800x600 window, and a lot of the game-play consists of horizontal movements of the mouse pointer. Secondly, it showed that for a specific activity, in this case the blast furnace activity, the movements are occurring more frequently diagonals. This also makes sense as this specific activity has a lot of diagonal mouse movements. Additionally, both the mouse velocity, Figure 3, and acceleration distributions, Figure 3, were also what we expected. As high velocities (and high accelerations), generally only occur on large "swipe" motions of the mouse (which are performed rarely), it should be expected that the majority of mouse-movements will have a lower velocity, as well as a lower acceleration. This is really demonstrated well by Figure 5, as 80% of mouse-movements occur below 100px/s and 1500px/s^2.

All naive bot movement methods were substantially different from the human movements, as shown by the Kolmogorov-Smirnov test, Figure 4. An alpha value of 1 means the data is coming from the same distribution, and an alpha value of 0 means it is not. As the analytic cumulative distribution was "trained" on human data, when it was compared to the ECDF's for naive bot data, the alpha value quickly plummeted to 0. The only alpha value that didn't quickly head to 0 was for cubic pause time, and this was because a small amount of effort was invested into making the naive cubic have a similar pause time distribution. A Kolmogorov-Smirnov test was not performed on the advanced cubic bot data because this data had it's metric values approximated based on the observations in the human dataset. Thus it will have a similar distribution. Unfortunately however, the existence of this movement generating algorithm shows that we could trick these metrics, and thus any detection system trained on them. This implies that we need stronger metrics to detect more advanced bots, and further research into better metrics needs to be done.

Looking at the statistical results, we can conclude that mouse movements should be in the toolkit of an anti-cheat developers arsenal, however, without better metrics they won't be able to fully detect bots.

3.2 Feed Forward Neural Networks

We can see that using the metrics to train the neural network, the naive mouse movements coming from bots are mostly all detected 6. This even occurs with 1, 2, and 4 nodes in the hidden layer. Thus, we can do a pretty good job and get a reasonably high accuracy, however, false positives were still occurring on all numbers of nodes in the hidden layers of the neural network trained only with metrics as inputs. In the neural network that was trained on mouse events in a mouse movement, the false positives rate dropped to 0 for both 32 nodes and 16 nodes. This is fantastic as no humans were getting banned. When the advanced cubic movements were used to train the neural network alongside human movements, the neural network could only classify movements with 56% accuracy. This is why it was decided to attempt training a neural network on mouse events instead of the metrics.

To prevent humans from getting falsely classified as bots, it would be wise to aggregate the outputs of any neural net that is classifying mouse movements. You could do this by accumulating mouse movements and counting how many are classified as bots vs humans, and then choosing a threshold ratio to ban people. For example, only ban people if 100% of every 200 tested mouse-movements are classified as bots. This threshold ratio would probably vary depending on the nature of the game, and would need to be determined specifically for the game that wants to use a neural network to detect bots.

3.3 Future Work

Future work could look at more metrics such as the ones below:

- *Time-to-Max-Velocity*: the time it takes to reach max velocity.
- *Time-to-Max-Acceleration*: the time at which the max acceleration occurs.
- *Max-Deviation-time*: the time when the max deviation occurs.
- *X-shake-sum*: the absolute value of the sum of all x movements in a mouse movement.
- *X-shake-count*: the number of times the direction on the x-axis changed.
- *Y-shake-sum*: the absolute value of the sum of all y movements in a mouse movement.
- *Y-shake-count*: the number of times the direction on the y-axis changed.

In addition to mouse movement metrics, mouse interactions with objects can be modeled. For example, future work could investigate the distribution of clicks on an object. For example, a naive click-pattern

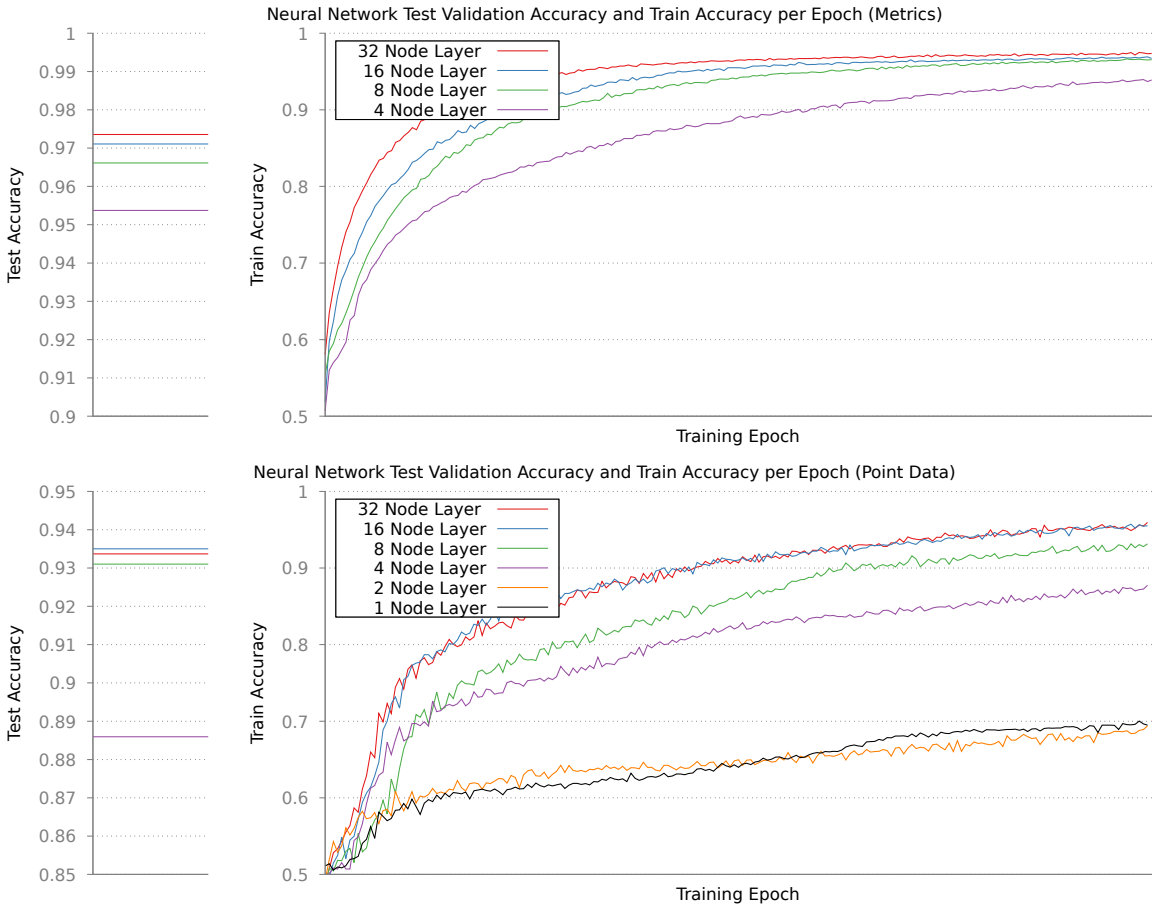

Figure 6: Training epoch vs accuracy for the metric-based neural network with a linear bot model (top) and the point-based neural network with advanced cubic bot model (bottom). On the left is the test accuracy achieved after the neural net has finished training, and is tested on data it has never seen before, with the corresponding number of nodes in it's hidden layer. For the purely-metric based neural network, the accuracy approaches 95%+ even with four nodes, while the point-based network struggled due to the higher quality bot data.

would most likely click on the same pixel on the object repeatedly, whereas a less naive one may click on any pixel that represents the object with equal probability. A well devised click-pattern would probably have a concentration mean on the center of the object, and the probability of clicking at some radius away from the center would decay according to a normal distribution. An analysis of "misclicks", where a human tries to click on something but accidentally clicks on something else, could also be analyzed.

References

[1] `https://www.digi-capital.com/reports.html\#global-games-investment-review/`. Accessed: 2018-12-01.

[2] `https://newzoo.com/insights/rankings/top-25-companies-game-revenues/`. Accessed: 2018-12-01.

[3] Kuan-Ta Chen, Hsing-Kuo Kenneth Pao, and Hong-Chung Chang. In: *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*. ACM. 2008, pp. 21–26.

[4] Hashem Alayed, Fotos Frangoudes, and Clifford Neuman. In: *Computational Intelligence in*

Games (CIG), 2013 IEEE Conference on. Citeseer. 2013, pp. 1–8.

[5] Luca Galli et al. In: *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE. 2011, pp. 266–272.

[6] Hsing-Kuo Pao et al. In: *Knowledge-Based Systems* 34 (2012), pp. 81–90.

[7] Ryan Kaminsky, Miro Enev, and Erik Andersen. In: *University of Washington, Tech. Rep* (2008).