Detecting Neural Network Driven Bots in Super Mario Bros

Caleb Cavilla[1], Jonathan Hudson[1]

[1]Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, T2N 1N4

Abstract

Gaming bots, or programs that allow players to automate their game-play, are used by people for various reasons. This includes monetary gain, notoriety, and even malicious intent, ruining the integrity of the game and the experience of other players. Despite all the advancements made in the field over the last couple of decades, there has been limited research in the detection of newer, neural network-driven bots which have only recently become popular due to rising accessibility for the general public. Our research proposes a novel addition to the existing literature with a Temporal Convolutional Network (TCN) as a classification model for detecting such bots. We have shown that our model can accurately predict whether or not an unlabeled input sequence from the first level of Super Mario Bros. was performed by a human or bot with a 94% accuracy rate, showing that the accurate detection of modern game-bots is possible.

Keywords

Super Mario Bros, Temporal Convolutional Network, Bot Detection

1. Introduction

Gaming bots are programs that automate player actions. Depending on the game, these bots can be programmed to perform a wide variety of tasks, such as farming resources, attacking players, or even scamming others. In a 2025 survey of 818 US adults, 70% indicated they regularly encounter bots while playing, and more than 7 in 10 US gamers agreed bots both ruin multiplayer competition (71%) and make it less fun to play certain games (74%)[1]. This shows that bots are not only common in gaming, but they ruin the immersion, competitive integrity, and joy that gaming has to offer.

This paper focuses on a particular sub-culture of gaming known as speed-running. Speed-running video games, or trying to complete a game as fast as possible under a particular rule set, is a game-play style where players spend immense amounts of time honing their skills and striving to achieve coveted world records in their game of choice. However, cheating by trying to pass off highly realistic and high performance bot-game-play as human could create uncertainty regarding the legitimacy of records going forward.

Historically, cheating using bots (sometimes referred to as 'botting') to complete speed-runs has been done through the use of scripts created by community tools, these are colloquially known as Tool-Assisted Speedruns (TAS) [2]. While convincing at a glance, speed runs that use scripts can be quickly identified as botting since the in-game player will consistently have 'inhuman' reactions, inputs, and timings. However, more recently the emergence of ever more popular, powerful, and accessible neural network driven bots poses an imminent threat to the speed-running community. Such bots can produce incredibly realistic human-like gameplay leaving no obvious traces or indications that a human would be able to use to identify the player as a bot. Given that speed-running communities are entirely self-moderated by passionate, but imperfect volunteers, this challenge highlights the need for an objective tool for detecting bots driven by neural networks. Our motivation stems from these challenges, seeking to tackle this technology gap by focusing on automatic and reliable detection of neural network-based bots.

Joint AIIDE Workshop on Experimental Artificial Intelligence in Games and Intelligent Narrative Technologies, November 10-11, 2025, Edmonton, Canada.

caleb.cavilla@ucalgary.ca (C. Cavilla); jwhudson@ucalgary.ca (J. Hudson)

1.1. Problem Definition

Formally, we developed a server-sided, input-focused approach using a neural network model that is capable of detecting fraudulent speed-running submissions that are completed by a bot, which itself was also trained and operated by a neural network. We used *Super Mario Bros.* as a case study, which is a popular game for researchers and hobbyists [3]. Our research question was: Is a Temporal Convolutional Network (TCN) model capable of detecting bots in Super Mario Bros. with a high accuracy rate on an unlabeled input set of human/bot submissions?

1.2. Contributions

While answering the main problem, our research makes the following contributions:

1. We researched and implemented our own game-bot using existing neural network technologies and algorithms capable of playing Super Mario Bros. at a high level.
2. Collected significant and modifiable data for both human and bot submissions of Super Mario Bros. to use in training the detection network.
3. Highlighted how existing detection methods, which focus on detecting *rule-based bots* struggle to detect the more natural and adaptive game-play of *neural-network-based bots*, and what improvements (or entirely new methods) can be made to counter these newer bots.
4. Showed it was possible to detect neural network-based bots using an entirely input-focused approach by detecting bots on nothing more than their input sequences (Left, Right, Up, A button, B button, etc...), avoiding reliance on in-game factors such as player trajectory/movement, coins collected, attacks, enemies defeated, etc.

By solving our research problem, we developed a system that allows for automatic, objective, and efficient detection of neural network-driven bots. In the context of the speed-running community, it keeps confidence in the future integrity of the community alive and would allow a bit of relief for often overworked admins if deployed as a tool. In the context of the wider world in general, we hope our contribution plays a role in detecting such bots in potentially more serious environments (such as social media, financial markets, etc).

2. Related Work

Research on the topic of bot detection can be broken down into where it falls in the Detection layer (Client side, Network side, Server side). Each method carries its own strengths, limitations, and application contexts. By examining each of these methods we can gain insights into all the strategies and tools available for detecting bots in video games.

2.1. Client Side

Client-side methods involve the detection of bots directly on the player's home system. This is typically done in two main ways. The first is done through a CAPTCHA system, where players are prompted in the game to solve a 'challenge' that is generally considered only solvable by a human, resembling a Turing-Test-like approach to bot detection [4]. The second client-side method involves requiring the player to install detection software directly onto their systems. Popular modern examples of this are Blizzard's 'Warden' or Riot Games' 'Vanguard' systems, which run in the background while playing their games, closely resembling an anti-virus-like program that searches for suspicious software running on the player's computer [5]. These methods are intrusive and disruptive to gameplay, making them undesirable for most purposes. Additionally, in the context of speed-running, evaluation of runs is done entirely post-hoc (submissions are evaluated after they are completed), meaning client-side methods, which try to detect bots live, are not even applicable in our case.

2.2. Network Side

Network-side approaches utilize an analysis of the network traffic exchanged *between* the client and the server to make their decisions [6]. These methods often create significant burdens on network performance, causing high response times (commonly known as "lag") and also suffer from the previously discussed post-hoc issue in speed-running, making them again, inapplicable in our case.

2.3. Server Side

To overcome the limitations posed by client and network-sided methods, much of the literature chooses to focus on server-side approaches. These detection methods typically involve the use of 'activity logs' collected by the game server which contain information on the in-game actions and movements of the players. These logs are then processed to determine if the player's behavior is that of a human or a bot. Since this method involves the collection and analysis of data after the fact, the server-side approach is the most desirable in the context of our speed-running challenge. There exist three main approaches to the server-side method.

1. **Actions/Inputs**: This detection method involves an analysis of the physical inputs of the player to the game (keyboard strokes, mouse movements, etc.) or the resulting in-game actions from those inputs (damage dealt, item usage, attacks dodged, etc.). Bernardi et al. [7] use physical input frequencies to detect bots in a WOW, an MMORPG. Chung et al. [8] study an MMORPG called Yulgang Online, and it focuses more on in-game metrics. For example, they track Hunting, Attacks, Hits, Defense, Avoidance, Drops, etc. As an overview of the action/input method, raw inputs and actions are converted into useful parameters that can be fed into a machine learning model, which then returns a decision on the player's humanity based on how the frequency, timings, and types inputs compare to *human* inputs.
2. **Trajectory**: This approach to detection involves tracking the player's in-game movements. In many cases bots tend to go from point A to point B directly, taking no time to deviate off their path or perform inefficient movements. Humans tend to be more unpredictable and inefficient, taking the time to explore the environment on the way to their destination. We can use these assumed differences to identify human vs. bot movement patterns. Qi et al. [9] convert players' trajectories in MMORPGs into a directed graph structure and then use a unified Graph Neural Network (GNN), and Gated Recurrent Unit (GRU), system to detect bots. This method tends to struggle with neural-network-driven bots as they have the chance to make inefficient (human)-like movements, as opposed to the highly efficient robotic-like movements of rule-based bots.
3. **Social Activity**: In multiplayer games, we can use the interactions between players, such as messages or trade attempts, as a way to identify bots. Kwon et al. [10] explore a rule-based method for detecting Gold-Farming-Groups (CFGs) in an MMORPG called 'Aion' by creating rules based on the number of trades, participation in certain 'clans', total gold spent, amount of free gold/items given, etc. Above certain thresholds these players were deemed to be bots, and deemed humans otherwise.

All of the above methods employed in current research on this topic, particularly those in the server-sided approach, operate under the fundamental assumption that ***bots act differently than humans***. This assumption arises from the fact that historically bots have been developed algorithmically. The novelty of this research lies in the fact that newer, neural network-driven bots question this assumption by introducing bots that ***do act*** similarly to humans, which left us with the challenge of developing a method to detect these more advanced bots, under the new assumption that bots can act extremely similar, if not exactly like human players.

In the case of this paper, we explore a learning-based, server-sided input/action approach to detecting neural network-based bots in the context of speed-running video games.

3. Solution

In this section, we introduce a slightly modified ***Temporal Convolutional Network (TCN)*** originally proposed by Bai et al. [11] as our chosen model to implement an action/input-focused approach to detecting bots in video games. TCNs are a special type of convolutional neural network (CNN) specifically designed to handle sequential, time series data where the *order* and *timing* of a sample matters. They are built to capture temporal (time-related) dependencies using 1-D convolutional layers. These features are then internalized and often used for classification and forecasting tasks. The input to the model (discussed further in the experimental setup) is a list of actions made by the player on each frame. You might notice that this means our data is sequential (the ordering of actions matters) and time-based (the frame in which the action was captured), making TCNs the perfect model for our problem. The model's output is a binary classification of the input sequence being that of either a bot or a human. Here, we will dive into detail on the architecture, strategy, and potential benefits and drawbacks of TCNs for our task.

3.1. TCN Architecture

TCNs rely on four core concepts as the foundation for their architecture and sequential modeling. Causal Convolutions, Dilated Convolutions, Residual Blocks, and Fully-Connected Convolutional Layers.

3.1.1. Causal Convolutions

One of the features that make TCNs unique and particularly useful for sequential data is the use of causal convolutional layers. Suppose we have a sequence of inputs $x_0, ..., x_n$ and we wish to predict some outcome y_t at a particular time step t. Logically speaking, we want the prediction y_t to only use inputs $x_0, ..., x_t$, and not $x_{t+1}, ..., x_n$. Informally, we want the current prediction at time step t to only rely on the inputs that come at and *before* it, and nothing after. This prevents "future leaking" where information that has yet to be processed interferes with the current prediction, ensuring that inputs are processed sequentially. To this end, TCNs employ 1D causal convolutional layers, where output at time t is convolved only with elements at time t and earlier.

3.1.2. Dilated Convolutions

In order for causal convolutions to make a prediction y_t, they must be able to 'see' the entire input $x_0, ..., x_t$. However, regular convolutions are only able to look back at a history with a size linear to the number of layers in the network. For example, a network with 4 causal convolutional layers with a filter size $k = 3$ making a prediction y_t would only be able to see inputs $\{x_{t-4}, x_{t-3}, x_{t-2}, x_{t-1}, x_t\}$. For tasks where long-term dependencies may be crucial in making predictions, this is simply not enough. For regular CNNs, the solution would be to either increase filter size (the "window" or how much of a sample the model sees at once) or have a massively deep network of thousands upon thousands of layers, both of which would lead to extremely high computational cost. To overcome these limitations, TCNs introduce dialated convolutions, in which the filter can skip over multiple time steps allowing the network to look further back into the input sequence. As depth increases, dilation factor d also increases exponentially: $d = 2^i$ at level i of the network. This ensures that at least one filter hits every single input without increasing computational costs.

3.1.3. Residual Blocks

Often when dealing with extremely large input sizes that require many layers, nearly all neural network models struggle with the ***Vanishing Gradient Problem*** [12]. When training a model, the gradients of the loss function are propagated backward from the output layer to earlier layers through the process known as backpropagation. At each layer of the network, the gradient is multiplied by the weights of the neurons and passed backward; if the weights at each layer are small ($w < 1$), this can lead to the

gradient decreasing exponentially as it is passed through layer by layer. Extremely small gradients mean that earlier layers will barely be updated, leading to long and unstable training. To combat this problem, TCNs introduce 'Residual Blocks' which are groupings of causal convolutional layers. Consider the residual block to be a function F. Let x be the input to the block function and $F(x)$ be the output after passing through all the layers inside of the block. The final output of the residual block y will actually be $y = F(x) + x$. By simply adding the input to the output of the block, we allow the gradient to 'skip' the block and affect the input directly, preventing it from being diluted as it passes through the inner layers. This allows the gradient to propagate backward through the network without 'disappearing' as fast as it otherwise would have.

3.1.4. Fully Connected Layer

This is were we slightly alerted the original TCN design which does not contain a fully connected layer. At this point, we can consider the TCN as a series of connected Residual Blocks, each of which contains a series of dilated causal convolutional layers. The output of this part of the TCN would be a series of high-level features that the network was able to extract from the input sequence. For classification tasks, we tack on a final fully connected (dense) layer that aggregates these features and maps them to one of the output classes (bot or human). This final layer is the decision maker that finally determines what class the input likely belongs to.

3.2. TCN Benefits/Drawbacks

There were many options that we could have chosen as our detection model. In particular, RNN models such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) are specifically designed for sequential data and have been widely used in the past. Variational Encoders and even Generative models have also been proposed in similar situations (although less frequently), so why TCNs?

3.2.1. Benefits

1. **Sequential Data Processing:** Convolutional Networks in general are *not* designed to handle sequential time series data, however through its use of causal convolutions and dilation, TCNs are capable of handling such data while still maintaining the robustness and efficiency of typical CNNs
2. **Parallelization:** Unlike RNNs which handle one time step at a time, TCNs are a type of convolutional neural network that have filters that act as a large "window" sliding over a multitude of time steps at once. This allows for parallel computation over the input decreasing computational resources and time.
3. **Longer Memory:** The dilated convolutions allow TCNs to have a massive receptive field without increasing costs. This allows TCNs to learn long-term dependencies effectively.
4. **Stability:** Through the use of Residual Blocks allowing for skip connections, TCNs boast stable and accurate gradient flow and training.

3.2.2. Drawbacks

1. **Sensitivity:** TCNs are highly sensitive to the initial starting conditions and hyperparameters such as filter size, dilation, and depth. High amounts of tuning and experimentation are required to utilize them at their full potential.
2. **Overfitting:** Large TCNs are prone to overfitting given small and noisy data sets. Given we manually collected our data, time constraints meant our data was relatively small, this is in addition to the fact that our input sequences were thousands of steps long, so this posed a problem we had to deal with.

3. **High Memory Costs:** Deep TCN models simply require massive amounts of memory to work properly, more so than that of other common models.

For our particular problem, we believed the benefits of TCNs outweighed the drawbacks and have the ability to act as the perfect solution to our problem . This is in conjunction with the fact that TCNs are typically used for forecasting tasks and not classification (common applications in traffic, weather, and sunspot prediction). And, to the best of our knowledge, have not been used in video game bot detection, further adding to the novelty of the project. The details of the TCN as a solution to detecting bots in video games will be explored in the next section as well as its implementation in code.

4. Experimental Setup

In this section, we discuss the datasets, model implementation details, and evaluation metrics that will be used to assess the model's performance in detecting bots.

4.1. Datasets

When developing supervised learning models, a challenge is often not developing the model itself, but rather collecting sufficient and usable data for training. For the task of detecting bots in Super Mario Bros., two data sets were required, a set of labeled bot submissions and a set of labeled human submissions. Given that we were trying to develop an action/input-based detection model, we wanted our data to contain nothing more than a list of inputs completed during the course of the submission. For our final model, we collected an even split of 1000 bot runs and 1000 human runs, for a total of 2000 samples to train the model. Here we will go into more detail on the collection of the bot and human data separately.

4.1.1. Bot Data

One of the most important objectives of this paper (and the main novelty source of this project) was testing detection methods for modern *Neural-Network-Based bots* capable of natural, human-like gameplay. Of course, this means that the bot data that we collected should have been sourced from a neural network bot. To solve this, we created a custom implementation of a recurrent neural network [13] that is capable of completing the first level of Super Mario Bros. We developed the bot using OpenAI's Gym API [14].

As mentioned, the bot is a type of recurrent neural network (RNN) and operates based on a typical Reinforcement learning framework of agents and environments. The bot acts as an agent interacting with the first level of Super Mario Bros. - the environment. Based on the bot's actions within the game, the environment updates and sends rewards back to the bot. In the case of speed-running, we want to incentivize the network to move *forward* as *fast* as possible, hence we program the environment to return a +1 reward for every unit right the agent moves and -1 for every second that passes on the in-game clock. The agent chooses its actions based on its policy function which determines what actions to take based on the current state of the environment to maximize the chance of getting a reward. By optimizing the agent's policy using the Double Deep Q-Network (DDQN) algorithm [15], we can create a bot that chooses the best action in every state of the environment, allowing it to complete the level as fast as possible.

After training the bot for about 100,000 iterations which took nearly 14 days of uninterrupted training, the bot was only able to reach the end of the level roughly 10% of the time. However, when it was able to complete the level, it did so at a respectable pace of about 66 seconds on average (see Fig 1). Given that the world record sits at 53 seconds and our average human pace was 61 seconds, our bot's speed is very comparable to humans. By saving and re-running, were able to program the trained model to log its input sequences and report them to a JSON file after each attempt at the level. Given the 10% success rate, we only saved runs that made it to the end flag pole. In the end, we collected 1038 bot

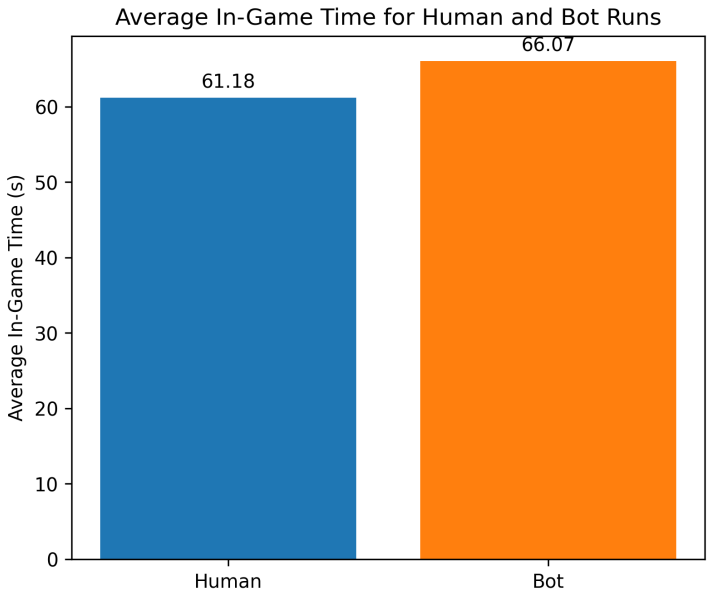

Figure 1: Average Submission Time Human Vs. Bot

input sequences to train our model on, this number was chosen to match the limited number of human samples (discussed next) so that we did not end up with an unbalanced dataset.

4.1.2. Human Data

At first, an attempt was made to find publicly available data sets of Super Mario speed runs with associated input data and timings. Unfortunately, this search turned up nothing, requiring a compromise to be made. Due to time constraints, we had to manually collect human data by completing our own runs of Super Mario Bros. Luckily this is quite trivial to set up, as the same Python Gym API that was used to train our bot model has a human mode, allowing us to play and track inputs in the exact same environment as our bot. However, this compromise has several drawbacks.

1. We (the authors) are not Super Mario Bot speed runners, meaning the human data collected may not be generalizable to speedrunners.
2. Collecting data manually is extremely time-consuming, taking about 50-100 seconds per run, meaning the size of our data set was naturally very restricted. For machine learning projects, the bare minimum number of samples for non-trivial projects is about 1000 per class, so this was our goal.
3. The fact that we are requiring software to be on the player's PC to track input sequences technically means we are developing a client-sided / server-sided hybrid detection strategy, however, detection is still being done post-hoc after the run is completed, so we deem this to be acceptable.

All of these shortcomings could be overcome by creating *another* separate model to analyze video footage of gameplay and extract input sequences from that. This has the extra benefit of being more accurate to how real speedrunners submit their runs to leaderboards in practice. However, such a task could be an entire research project by itself and is simply not possible with our circumstances. With that said, we now have a way to collect human runs in the same format as the bot runs. After about 15 hours of cumulative gameplay, we were able to collect 1060 runs, giving us the required data to train the detection model.

4.2. Data Pre-Processing

Since we collected all of the data by ourselves, everything was already close to being in the exact format and structure that we needed. However, there were still some issues that had to be dealt with, mainly

pertaining to how TCN's expect inputs to be formatted. Here, we will discuss these problems and how we solved them.

4.2.1. Data Compilation

From the data collection phase, we had roughly 1000 human samples and 1000 bot samples in the form of JSON files containing the input sequences for each run. We saved these JSON files in the following format: (human/bot)_episode_ timestamp_runtimes.json. This gave us access to the player type and runtime of each run organized by date. However, for the sake of producing training and test sets, it's much easier if we compile all 2000 files into a single CSV. This was a relatively straightforward process of iterating over each file, extracting the input sequence list, checking the file name (human/bot), and appending it to the table in the CSV file. This gave us a table with 2 columns (input_sequence, is_bot) with input sequence being a list of inputs and is_bot being a binary label indicating if the run was completed by a bot (0 = human, 1 = bot).

4.2.2. Data Encoding

The input sequences that are present in the JSON files (and now the CSV table) come as categorical tokens of inputs formatted as lists of strings. However, TCNs expect numerical values as inputs so we must encode these lists as numbers. There are 6 possible input types are present in the lists: ['NOOP'],['right'],['right', 'A'],['right', 'B'],['right', 'A', 'B']. We encoded these tokens as a fixed three-digit binary vector, this is known as multi-hot encoding. Each digit in the vector corresponds to one of the three buttons (right, A, B) respectively, with 1 indicating if the button was pressed and 0 if not. 'NOOP' indicates no button was pressed, corresponding to the encoding [0,0,0]. Table 1 below outlines the 6 possible encodings to give a better idea of what was happening.

Table 1
Input Encoding

Un-encoded Input	Encoded (Multi-hot)
`['NOOP']`	[0, 0, 0]
`['right']`	[1, 0, 0]
`['right', 'A']`	[1, 1, 0]
`['right', 'B']`	[1, 0, 1]
`['right', 'A', 'B']`	[1, 1, 1]

After applying this to each sample, we got a list of inputs that looks something like: [[1,0,0], [1,0,0] [1,1,0] ...]. This format is understandable by the model while not abstracting away valuable information about which button's were pressed in each input.

4.2.3. Data Padding/Trimming

Due to variations in how the actual gameplay of each run pans out, the length of the input sequences is not consistent even when they all make it to the end flagpole. In fact, across both the human and bot runs, the shortest sequence was 854 while the longest was 2526. There are problems with this, First off TCN's, and most forms of convolutional networks require fixed-length inputs. Secondly, long inputs can introduce a lot of noisy/irrelevant context. With excessively long sequences the model may end up overfitting to the noise rather than focusing on key generalizable patterns that differentiate bot and human play. Third, the average bot game was longer than the average human game, if we don't trim inputs the network might use this feature to overfit and label longer games as bots. Forth, long sequences are simply computationally expensive, which can lead to long training times and failure to find convergence. To solve these issues, we need to pad/trim our input sequences to a fixed length. After training the TCN model on several different input lengths, we found the highest accuracy with input sequences that are 1000 steps long. At this level the model still has enough data to work with,

preventing underfitting, but not too much data that it causes overfitting and unreasonable training times. To achieve this, for sequence lengths greater than 1000, we simply take the first 1000 inputs. For sequences shorter than 1000, we will add (1000-sequence_length) 'NOOP' inputs to the end. Since the average sequence length was 1773, the vast majority of the samples are being trimmed.

For reader comprehension, we have compiled 5 samples from the actual post-processed dataset into table 2 to give a clear understanding of the data we were working with before moving on to the detection model:

Table 2
Sample Data from CSV

input_sequence	is_bot
[[1, 1, 0], [1, 1, 0], [1, 1, 1], [1, 1, 1], [1, 1, 1], ...]	1
[[1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0], ...]	1
[[1, 0, 1], [1, 0, 1], [1, 0, 1], [1, 0, 1], [1, 0, 1], ...]	0
[[1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0], ...]	0
[[1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 1], [1, 1, 0], ...]	1

4.3. The Detection Model

As mentioned in 3.1, the detection model is a type of convolutional neural network known as a Temporal Convolutional Network (TCN) that is adept at processing and classifying long sequential time series data. The model was implemented using the keras-tcn package, which was developed and still maintained by the authors of the original TCN paper [11]. This package allows us to abstract away all of the technical details such as dilated convolutions and residual blocks into easy-to-adjust hyper-parameters. A high-level implementation of the project is described below:

4.3.1. Preprocess data

The data is preprocessed such that we have a CSV file of encoded input sequences and labels. The sequences are split into train/test sets on an 80:20 ratio, the same is done for labels. This split strikes a balance between having enough data to sufficiently train the model and having enough samples to test the model's performance after training.

4.3.2. Constructing/Optimizing TCN Layer

The keras-TCN package gives us access to a pre-built TCN layer with adjustable hyperparameters, but how do we know what hyperparameter values are the best for our particular model? The answer is the Succesive Halving Algorithm (SHA). SHA lets us define a 'range' of possible hyperparameter-values that we want to try out and efficiently searches for the best combination iteratively. It works by randomly selecting a wide set of hyperparameter combinations and doing a light test (1 epoch). Then, the top 33% of models are passed to the next round and allocated more epochs for testing. For each iteration, the number of epochs increases and the boundary for passing is raised until only 1 model remains with theoretically optimal hyperparameters. We then store these hyperparameters and create the optimal TCN layer to be used in the final model. The parameters tested and their final values are provided below:

1. `nb_filters` : 128
2. `kernel_size` : 3
3. `nb_stacks` : 1
4. `dialations` : (1,2,4,8,16,32)
5. `dropout_rate` : 0.3
6. `use_batch_norm` : False
7. `activation` : tanh

4.3.3. Input

TCN's expect an input of format: (batch_size, sequence_length, features) where:

- `batch_size` is the number of samples to evaluate at once, we always used a batch size of 32 in our testing.
- `sequence_length` is the length of our input sequences, in this case always 1000 (see section 4.2.3).
- `features` is the number of features in each input, in our case it is always 3 via the 3-digit multi-hot encoding (see section 4.2.2)

this means our input to the model is always of shape 32 x 1000 x 3.

4.3.4. Output

The final layer of the model is a fully-connected dense layer containing a single neuron that used activation sigmoid to squish the output into a value between [0,1]. This value represents the models confidence that the input sequence came from a bot, if its value is over 0.5 it will predict a bot label of 1, otherwise a human label of 0.

4.3.5. Compiling

The model is compiled using the Adam optimizer and binary cross-entropy loss function. Adam optimizer allows for adaptive learning rate tuning and gradient optimizations, leading to quicker and optimal convergence. The binary cross-entropy loss function is what is responsible for evaluating how well the model's predicted output matches the true label and calculates the error for each sample in the batch. This will be the gradient that is propagated back through the network and used to adjust weights and biases as discussed in 3.1.3.

4.3.6. Fitting

The model is fit on the train set using a 10% validation split to monitor the models accuracy and loss during training. This provides an early indicator of overfitting if the training accuracy is improving but validation set (representing unseen data) is declining. We fit for a maximum of 20 epochs but allow for early stopping if we detect that the validation loss is no longer improving, suggesting convergence has been reached.

4.3.7. Evaluation

Finally, the experiment is concluded by evaluating the model's performance on the test set to check how well the model can detect bots. The metrics used are proposed and discussed extensively in 4.4 and the results, which determine the success or failure of our experiment, are shown and analyzed in section 5.

4.4. Evaluation Metrics

Since we are dealing with bot detection as a form of binary classification, we need our evaluation metrics to evaluate the accuracy of our predictions and frequency of miss classifications (false positives/false negatives). To this end, we will use accuracy, precision, recall, and its F1-Score to assess the performance of the TCN model in detecting bots in Super Mario Bros. In particular, we are most interested in a model that boasts both a high accuracy and F1-Score. Such results would indicate that our detection model can both accurately and fairly detect neural network-based bots using an entirely input-focused approach.

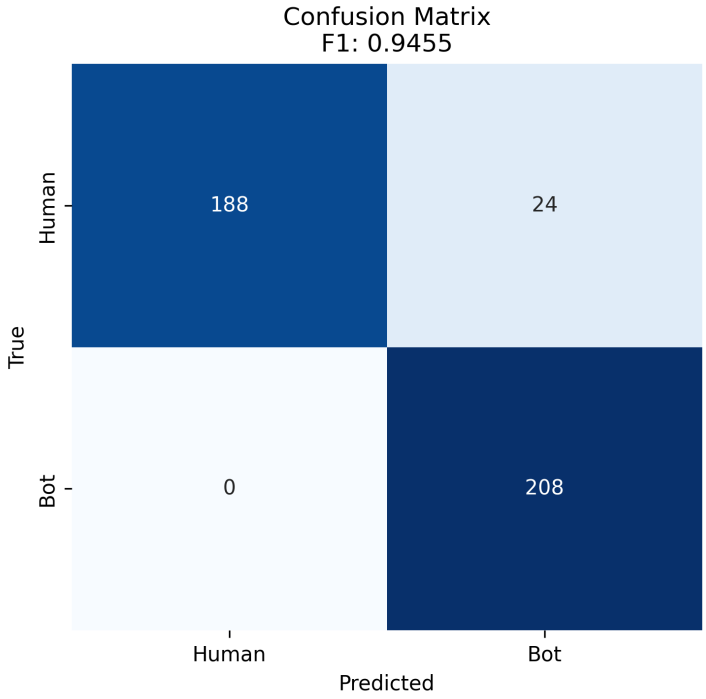

Figure 2: TCN Confusion Matrix

5. Evaluation

Here, we will detail the performance of the model on the test set of 400 samples, explain what the results mean, how they prove our objectives/overall problem definition, and discuss the limitations of the project and how these be improved in future research.

5.1. Results

On the test set of unseen data, the model boasted an impressive performance of 94% accuracy with a macro precision of 0.95, recall of 0.95, and F1-Score of 0.95.

These results indicate several things about our model:

1. The model can make exceptionally accurate predictions. Given a sample, the model can predict whether or not that particular input sequence was produced by a bot or a human 94% of the time. Given that we had nearly perfectly balanced classes, this is an accurate measurement that can be taken at face value. In fact, this directly proves that the model is capable of detecting neural network bots in Super Mario Bros. using an entirely input-focused approach.
2. The precision of the model for the bot class is 0.90, while Recall is 1.00. This indicates that the model is slightly biased towards making 'false positive' predictions, or labeling human runs as bots when they otherwise shouldn't be. However in exchange, the model has a perfect catch rate. If a sample came from a bot it predicted it as so 100% of the time, allowing no bots to go undetected (no false negatives).
3. F1-score is the a harmonic mean between precision and recall. An F1-score of 0.95 indicates that the model can strike an amazing balance between false positives and false negatives, and proves that the model is able to discriminate between bot and human classes with real predictive power.
4. For further comprehension of these results, a 'confusion matrix' is provided in Fig 2 to break down the exact predictions made by the model on the test set. The rows represent the true label and the columns represent the predicted label. From this, we can see the model correctly identifies all 208 bot samples, but it misclassified 24 of the 212 human samples as bots. This also allows

us to look at the metrics from the model's POV. When the model predicts human, it is always right, however when it predicts bot it is sometimes wrong, directly correlating to the 1.00 human precision and 0.90 bot precision.

Overall, we have shown that the model can not only accurately detect neural network bots, but can do so in a fair manner, only accusing humans of being cheaters 90% of the time.

5.2. Problem Definition Revisited

As outlined in section 1.1, the goal of this project was to identify and test a server-sided, input-focused approach using a neural network model that can detect speed-run submissions completed by a bot which itself was operated by a neural network with a high accuracy rate.

In the pursuit of this problem, we have developed a TCN based detection model that is capable of detecting when a Recurrent neural network bot is playing Super Mario Bros. using an entirely input-focused approach with a 94% accuracy rate.

5.3. Limitations and Future Directions

Here we will discuss the shortcomings of the project and provide some future research directions that could yield solutions to the problems we discuss.

5.3.1. Data Size / Source

As first mentioned in 4.1.2, there are limitations to the human data used in this project. Due to not having any publicly available datasets in the format and context that we required, we (the authors) had to complete the roughly 1000 samples of human runs ourselves. This had three main consequences: One, our dataset size is naturally restricted, which gave our model limited data to learn the patterns and features that differentiate bot and human game-play. Two, we are not Super Mario Bros. speed-runners meaning that our detection model may not be generalizable to the speedrunning community. Three, we have implicit playstyles and habits that affect how we play games, meaning the model may not be even generalizable to regular, non-speed-running gamers. These three issues pose the serious question of overfitting, as our detection model may have learned how to specifically differentiate bot vs. researcher runs rather than learning the intended bot vs. human runs.

There are many solutions to this issue that could be explored in future research. For one, instead of only having the researchers complete runs, the task could be assigned to the general public. Not only would this increase the speed at which data can be collected, leading to a larger dataset, but it would also introduce much more variation between human samples. A model trained on this data would be far more robust and generalizable to any run completed by most humans. Another option would be to develop an additional neural network that can accept video files as input and extract input sequences based on the visual game-play. Since there is an abundance of Super Mario Speed run videos online, this would mean no manual collection aside from finding videos. Such a model would be generalizable to speed-runners and would effectively solve the original problem in its entirety.

5.3.2. The Black Box Problem

While we have proven TCNs to be an excellent option for accurately identifying bots in video games, we were unable to extract any meaningful conclusions as to why our model was making the decisions that it did. This is a common problem found in many deep-learning projects. An interesting future research direction would be to add some explainable AI elements to the project that would force the model to show patterns it picked up on or explain features that it used to make its final classification.

5.3.3. Baseline Comparisons

During the course of this research project, we only explored and experimented with TCNs as our chosen detection model. We would continue this work by comparing TCNs with previously established models such as LSTMs and GRUs.

6. Conclusion

In this paper, we explored the problem of bot detection in video games. While this topic has been exhaustively researched in related works dating back to the 90s, we have shown that the advent of bots controlled by *neural networks* is a new problem relatively unexplored in recent literature. Such bots call the legitimacy of previous works into question by challenging the assumption that **bots act fundamentally different than humans**. Additionally, previous literature tends to rely on in-game metrics such as attacks, levels, trajectory between points, and currency that are not translatable between games. This makes the solutions that are proposed often too specific and un-adaptable to have any sustained use in the gaming industry as a whole.

To fill these gaps, we have proposed a TCN detection model for detecting a recurrent neural network bot using nothing more than the inputs of the player. Our proposed solution achieves exceptional results in the task of detecting when both humans and bots are playing the game with not only a high accuracy of 94% but also a high F1-Score, highlighting that our model is both accurate and precise in its predictions. Such a model not only detects neural networks in a way never done before through TCNs but does so only using raw input sequences, allowing the model to be theoretically generalizable to a wide variety of games with similar input schemes.

Acknowledgments

We thank the support provided by Mea Wang, instructor of the research capstone course in the Department of Computer Science at the University of Calgary. We thank Richard Zhao and the anonymous reviewers for their feedback.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] U. Author, 2025. URL: https://world.org/blog/announcements/bots-gamers-how-world-addressing-surging-global-demand-for-fair-play.

[2] M. Groß, D. Zühlke, B. Naujoks, Automating speedrun routing: Overview and vision, in: International Conference on the Applications of Evolutionary Computation (Part of EvoStar), Springer, 2022, pp. 471–486.

[3] J. Togelius, S. Karakovskiy, R. Baumgarten, The 2009 mario ai competition, in: IEEE Congress on Evolutionary Computation, IEEE, 2010, pp. 1–8.

[4] M. Guerar, L. Verderame, M. Migliardi, F. Palmieri, A. Merlo, Gotta captcha'em all: a survey of 20 years of the human-or-computer dilemma, ACM Computing Surveys (CSUR) 54 (2021) 1–33.

[5] B. van de Ven, Cheating and anti-cheat system action impacts on user experience (2023).

[6] P. Laurens, R. F. Paige, P. J. Brooke, H. Chivers, A novel approach to the detection of cheating in multiplayer online games, in: 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), IEEE, 2007, pp. 97–106.

[7] M. L. Bernardi, M. Cimitile, F. Martinelli, F. Mercaldo, A Machine Learning Approach for Game Bot Detection Through Behavioural Features, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, New York, NY, USA, 2009, p. 256–268.

[8] Y. Chung, C.-y. Park, N.-r. Kim, H. Cho, T. Yoon, H. Lee, J.-H. Lee, Game Bot Detection Approach Based on Behavior Analysis and Consideration of Various Play Styles, ETRI Journal 35 (2013) 1058–1067.

[9] X. Qi, J. Pu, S. Zhao, R. Wu, J. Tao, A GNN-Enhanced Game Bot Detection Model for MMORPGs, in: Advances in Knowledge Discovery and Data Mining, Portland, Oregon, 2022, pp. 316–327.

[10] H. Kwon, A. Mohaisen, J. Woo, Y. Kim, E. Lee, H. K. Kim, Crime Scene Reconstruction: Online Gold Farming Network Analysis, IEEE Transactions on Information Forensics and Security 12 (2017) 544–556.

[11] S. Bai, J. Z. Kolter, V. Koltun, An empirical evaluation of generic convolutional and recurrent networks for sequence modeling, arXiv preprint arXiv:1803.01271 (2018). URL: http://arxiv.org/abs/1803.01271.

[12] S. Hochreiter, The vanishing gradient problem during learning recurrent neural nets and problem solutions, International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 06 (1998) 107–116. doi:10.1142/S0218488598000094.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, nature 518 (2015) 529–533.

[14] C. Kauten, Super Mario Bros for OpenAI Gym, GitHub, 2018. URL: https://github.com/Kautenja/gym-super-mario-bros.

[15] H. van Hasselt, A. Guez, D. Silver, Deep Reinforcement Learning with Double Q-Learning, Proceedings of the AAAI Conference on Artificial Intelligence 30 (2016) 1476–4687.