Teaching computers to revel in glorious combat

A Capstone Project
by
Rei Armenia and Matthew James Harrison

Submitted to the Faculty of the
Department of Software Technology
in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE
IN
COMPUTER SCIENCE And INNOVATION

Champlain College
2018

Abstract

This paper explores the use of a recurrent neural network for creating an AI opponent that can play Rivals of Aether, an indie 2D fighting game. The first section defines the problem and context for this project. From there, we conduct a literature review in which we exploring the work of other researchers who are also working with machine learning as it applies to videogames. Finally, the project methodology is discussed.

Acknowledgements

This project would not be possible without the support and generosity of the *Rivals of Aether* community. We would like to offer our heartfelt gratitude to Etalus, girlritchie, and ICleanWindows for their direct contributions to our dataset. We are also grateful to everyone who took the time to engage with our thread, even if only to offer a comment of support or vote us up for visibility.

Lastly, we wish to thank Dan Fornace, flashygoodness, and the *Rivals of Aether* development team.

Table of Contents

List of Figures

Chapter 1: Introduction

1.1 Problem Statement

Artificial intelligence (hereafter referred to as AI) in video games has often been used for either academic research or customer analytics. We are a team of two undergraduate researchers who wish to see cutting-edge machine learning techniques, such as recurrent neural networks, applied in a way with which the general public can interact and, ultimately, have fun.

The concrete goal of this project is to train an artificial neural network to play Rivals of Aether using replays. This presents three significant obstacles: 1) converting the replay data into a labeled dataset, 2) designing and training the artificial neural network, and 3) writing a game agent that can send data to and receive instructions from the neural network during gameplay.

1.2 Purpose Statement

In order to employ machine learning techniques as a means for entertainment, this project seeks to develop a videogame AI opponent that is fun to play against. The AI shall only be able to make use of the visual buffer as its input. There are two reasons for this constraint. Firstly and chiefly, the visual buffer is readily available from outside of the game's runtime, and can therefore be obtained without the use of hooks or APIs provided by said runtime. This decouples the techniques used to inform and train the agent from any specific game implementation, thereby rendering those techniques reusable. Secondly, if the AI can effectively play with only access to the visual buffer, then it will essentially be playing with the same information with which a human would play. Additionally, this constraint further differentiates the AI from conventional game agents, which are allowed to make decisions based on explicit and concisely packaged game state information such as coordinates, predefined navigational paths, and health points.

The AI produced in this project will play Rivals of Aether, which is a two-dimensional or 2D fighting game created by independent game developer Dan Fornace [4].

1.3 Context

Interest in the field of artificial intelligence has been present since nearly the beginning of computing. However, over the past few years interest in AI has been growing considerably. With the recent popularity of machine learning techniques like neural networks, many researchers have made huge advancements in the field. Companies such as Google are frequently appearing in mainstream media for their groundbreaking projects. Google's

AlphaGo is an amazing example of the work that can be done; AlphaGo is an AI was able to beat the world's best Go player, and it continues to improve its game [1]. Other groups are contributing to the advancement of AI, although they are not as easily recognizable. A research platform called ViZDoom is one such example that closely relates to our work. ViZDoom is an Open Source platform that allows users to create agents in the videogame Doom, these game agents are doing what any human player would: trying to maximize their score. However, unlike more conventional game AI, they are playing using only the visual buffer. This approach more closely resembles how humans play the game, and was also a large inspiration to this project. Yet, ViZDoom is still primarily a research platform [15].

Furthermore, game-related AI research is limited by the actual games that are viable for machine learning. Regardless of one's approach, one will require either labeled training data, access to the game's internal state, or both; the former can be time-consuming to generate and the latter may not even be achievable. As a result, there are two games in particular that tend to be the focus of AI research: Doom and Quake. What sets them apart from other games is that they support the recording demos, which are files that keep a record of what controller inputs were detected from which player during each moment of gameplay. Thus, whereas a conventional video recording associates each frame with its pixel data, a replay does so with each timestamp or frame and its input data. This allows the game to simulate a past match in real-time with perfect accuracy by simply running a new match in which each player's control input is read from the replay file rather than from the controller. An AI can use a collection of these demos to essentially reverse-engineer how to play the game by simultaneously watching the demo and studying the corresponding input data. Doom and Quake are made further popular for AI research because there are immense repositories of demo files available on the world wide web, from websites such as Speed Demo Archives and Quake Terminus [20].

Finding cutting-edge AI techniques in the game industry is quite hard. Many developers find these techniques to be both impractical and excessive. These implementations use up sought after resources. Why should they create more work for themselves when many of the problems that we are looking to solve can be reduced to smaller problems with nice well defined algorithms? With this information, one may think that almost no one uses advanced A.I. in games, and that is primarily the case. However, some companies are implementing these advanced techniques. Instead of using AI to create a more unique and enjoyable experience, however, they use it to push sales for specific features. This is the case with Activision's recently acquired patent, which defines a method for manipulating players into engaging with micro-transactions [18].

1.4 Significance of Project

This project is, at least in part, a response to what we perceive to be a rising trend in the games industry for machine learning to be used for purposes which can be best summed up as customer manipulation and greed. At the time of writing, AI is one of the quickest growing STEM fields. We believe that it is important to do our part in establishing a

precedent, showing developers, publishers, and consumers what advanced AI can, and ultimately, ought to be used for.

This project is significant within the niche of game-related AI research because it introduces a hitherto unexplored game, Rivals of Aether, as a viable environment for supervised learning. Rivals of Aether supports the recording of replays, which are essentially the same kind of file as a demo. Unlike Quake and Doom, however, there are no well-established public repositories for replays. Therefore, in order to build our dataset, we solicited from the *Rivals of Aether* community. Contributors provided us with a total of 1,020 replay files from a variety of settings, including tournaments, exhibitions, ranked matches, and private games. Adding our own replays that we recorded ourselves brings the total up to 1,032.

Chapter 2: Literature

2.1 Literature Overview

In *Towards Integrated Imitation of Strategic Planning and Motion Modeling in Interactive Computer Games*, Dublin City University researchers Bernard Gorman and Mark Humphrys use imitation learning techniques to train an artificial agent to collect items in Quake II [13]. Specifically, their agent must traverse three-dimensional environments in order to intelligently seek out desirable items, such as power-ups, weapons, and health pickups. Furthermore, the researchers set an additional requirement: the agent must "employ human-like motion," which is to say that it must traverse the map in such a way that its movements appear fluid and natural, as if a human were controlling it rather than a machine. The researchers refer to this "imitation of the player's movement" as motion modeling. Gorman and Humphrys create their training dataset by retrieving state information from demo files. Specifically, they use topology learning techniques to define a Markov Decision Process or MDP based on all of the player and item positions noted in the demo. They then employ value iteration using information about the player's status, such as current health and inventory contents; this information is also derived from the demo. Value iteration assigns a utility to each state in the MDP and, ultimately, allows the agent to make decisions about where to travel next. Gorman and Humphrys also discuss their approach for motion modeling. First, they define a set of it action primitives by reading all of the input data from the demo file and then disassociating that data from its context. This allows the agent to, after deciding where to navigate, lay out a series of action primitives which will allow it to reach its goal [13].

Gorman and Humphrys cite *Learning human-like Movement Behavior for Computer Games* by Christian Thurau and Christian Bauckhage of Bielefeld University. The paper describes an experimental study which provides a useful behavioral model while also challenging the conventional approach for developing videogame AI [20]. Thurau and Bauckhage define a hierarchy of behavior types, consisting of reactive, tactical, and strategic behaviors. Strategic behaviors are essentially long-term plans for winning the game, such as which objectives to target and in what order. Contrast this with reactive behaviors, which are atomic, immediate decisions, such as moving in a particular direction, turning around, or firing a weapon. Tactical behaviors sit in between reactive and strategic, representing small-scale decisions such as circle-strafing an opponent, or choosing to run away from a battle in order to look for a health pickup [13]. To implement an agent that demonstrates such behaviors, Thurau and Bauckhage employ topology learning techniques to train a type of neural network called a neural gas, using player position data extracted from demo files. This methodology should sound familiar; Gorman and Humphrys use some of the techniques which Thurau and Bauckhage discuss [20].

The research described heretofore addresses considerably more complex problems than the one this project seeks to solve, simply by virtue of their game environment being three rather than two-dimensional. Nonetheless, the problems are still very similar, and as such

the solutions may also bear some resemblance as well. For one: Rivals of Aether replays and Quake demos provide the same kind of information. Furthermore, these projects, like our own, are seeking to create AI opponents that behave like human players. One could therefore reasonably wonder whether or not the techniques described above could translate into a two-dimensional space.

In *Learning to Act by Predicting the Future*, Cornell University researchers Alexey Dosovitskiy and Vladlen Koltun use supervised learning techniques to train an agent to play competitive deathmatches in Doom [8]. Dosovitskiy and Koltun do so by using two distinctive data streams: one is high-dimensional and consists of "raw visual, auditory, and tactile input," while the other is low-dimensional and provides basic game state information such as "health, ammunition levels, and the number of adversaries overcome." These data streams are referred to as the sensory and measurement streams, and together they serve as inputs for a supervised learning model [12]. The resulting AI, named IntelAct, performed remarkably well in the full deathmatch track of 2016 Visual Doom AI Competition [8].

However, on the same track in the 2017 competition, IntelAct was beaten by two other AI, with YanShi and Arnold4 coming in second and first place, respectively [9]. In *Arnold: An Autonomous Agent to play FPS Games*, Devendra Singh Chaplot and Guillaume Lample from Carnegie Mellon University describe their implementation, which consists of a pair of Deep Q-Networks working together in tandem. One of the networks is concerned with learning policies related to combat, while the other does the same for environmental navigation. Champlot and Lample implement this model with UltraDeep and Theano. The result of their project is an AI that "outperforms average humans as well as in-built game bots," which earned "the highest kill-to-death ratio in both tracks of the Visual Doom AI Competition" in 2017. Overall, this paper provides a convenient overview of the project [11]. However, Chaplot and Lample go into far greater detail in the paper titled *Playing FPS Games with Deep Reinforcement Learning* [16].

The results of Arnold and IntelAct deserve considerable attention, because their creators not only implemented creative solutions, but they also outperformed all of the other AIs during the 2017 and 2016 competitions. One notable common thread between the two projects is that the researchers found ways to break the problem into smaller, more manageable pieces. Chaplot and Lample do this at the learning stage by implementing two deep reinforcement learning networks [11], while Dosovitskiy and Koltun work from the input stage with their dual-stream approach [12].

In *Novel Moving Target Search Algorithms*, Peter K. K. Loh from Nanyang Technological University and Edmond C. Prakash from Manchester Metropolitan University explain the complexity behind creating an AI that can quickly and effectively respond to a human player's constantly changing position [17]. They state that this criteria creates conflicting requirements. They then introduce two Moving Target Search or MTS algorithms, called Fuzzy MTS and Abstraction MTS. These algorithms were compared against other MTS algorithms with promising results even within large problem spaces. [17]. Although the name "moving target search" might, in some contexts, sound like it is related to aiming a weapon and pointing it towards a so-called "moving target," that does not accurately describe what these algorithms are for. Rather, the the "moving target" in this case is

meant in a more general sense. For example, in the IBM blog post titled *Solving AI's moving-target search problem at IJCAI 2017*, an example of an MTS problem is given in the form of a cab trying to find a customer [7]. Thus, MTS algorithms may be a viable tool for our AI to employ.

In *ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement*, Cornell University researchers Michal Kempa, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski discuss the benefits of their project,ViZDoom, as a research platform [15]. They first discuss their earlier experiments with AI on the Atari 2600 that eventually led to the creation of ViZDoom. They also talk about the jump from two to three-dimensional environments. While this transition may seem problematic and difficult, in actuality a 3D space can simply be reinterpreted as an aggregation of many related 2D spaces. Kempa et all proceed to describe ViZDoom as a lightweight, fast and highly customizable framework for developing AI that plays Doom. As a proof-of-concept, the researchers implement and then train two bots using "Q-Learning and convolutional deep neural networks." The result is two human-like AI that demonstrate not only how ViZDoom can serve as a useful tool for AI research, but also prove the viability of "visual reinforcement learning in 3D realistic environments" [15]. This paper will serve as a valuable reference despite the difference in complexity between their problem and ours, because they are demonstrating the capabilities of the same machine learning algorithms which we are exploring.

In *Autoencoder-augmented Neuroevolution for Visual Doom Playing*, Cornell University researchers Samuel Alvernaz and Julian Togelius explore the use of an autoencoder alongside a neuroevolution technique called "Covariance Matrix Adaptation Evolution Strategy" or CMA-ES [10]. An autoencoder is essentially a technique for simplifying and reconstructing images. The researchers implement their autoencoder using Keras, a high-level machine-learning library. They then use the autoencoder to generate a simplified version of the frame buffer to use as the input for their neural network, which they then train to complete health pack gathering exercises. The autoencoder's purpose is to convert a high-dimensional state space, that being the full-resolution frame buffer, into a comparatively low-dimensional substitute [10]. The use of an autoencoder by Alvernaz et al can be directly compared with SethBling's use of simple downscaling and grayscaling. Both of these techniques result in a simplified state space; however, the key difference between them is that an autoencoder has the potential to learn a much more efficient way to achieve this goal. Otherwise, the one other key technique noted in this paper is CMA-ES. If our dataset, for whatever reason, proves insufficient or otherwise unusable, then neuroevolution algorithms such as this may be our best fall-back option.

2.2 Software

TensorFlow is self-described as "an open source software library for numerical computation using data flow graphs." It is designed explicitly for use in machine learning research, and is able to achieve exceptional performance by employing either GPU computation or SIMD CPU instructions [6]. This project uses Tensorflow-GPU alongside NVIDIA

CUDA/cuDNNfor for its underlying machine learning architecture.

Keras is a high-level machine-learning library designed to serve as an abstraction layer over low-level libraries including TensorFlow, CNTK, and Theano. The low-level library still performs all of the heavy-lifting; Keras simply removes some of the barriers to entry by providing a more accessible API [3]. This project uses Keras to define and compile a neural network.

SerpentAI is an open source Python framework that allows researchers to write game agent scripts that have the ability to easily retrieve the frame buffer from any computer game and then send keyboard or controller input back to said game. The framework is noteworthy for running natively, not relying on game API calls or other hooks, and not making any presumptions about what machine learning techniques the user wishes to implement. SerpentAI claims that it can support anything from simple reflex and random agents to the most advanced and bleeding-edge neural networks and neuroevolution algorithms [5]. This project uses a SerpentAI game agent plugin as an interface for getting the game's frame buffer and sending keyboard input to its window.

The Anaconda Distribution is software suite for Python 3 which provides package management, environments, development tools, and an immense collection of scientific and mathematical modules [2]. The SerpentAI documentation cites Anaconda as a requirement for its Windows users because without Anaconda it can be difficult to acquire many of SerpentAI's dependencies [5].

2.3 Other Sources

In *MariFlow - Self-Driving Mario Kart w/Recurrent Neural Network*, YouTuber SethBling trains a recurrent neural network to play Mario Kart using only the visual buffer [19]. The goal of SethBling's project is to train the neural network to mimic the way he plays the game by having it learn from fifteen hours of his own recorded gameplay. The network, created in TensorFlow, receives "a low-resolution grayscale version of the screen" for input and employs multiple layers, sigmoid neurons, backpropagation, and "long short term memory" or LSTM cells in order to predict the appropriate controller action. Furthermore, SethBling runs "interactive sessions" in which he monitors the neural network while it plays the game so that he can identify situations in which it gets stuck; he then takes over control of the game so that he can show the neural network what to do in such situations. SethBling released several learning resources with his video, including his source code, utilities, and detailed instructions [19]. Overall, not only does SethBling provide an excellent entry-level primer for the fundamentals of neural networks, he also gives an encouraging demonstration on how to solve a problem that is very close to our own.

In the post titled *The Unreasonable Effectiveness of Recurrent Neural Networks*, weblog author Andrej Karpathy introduces the concept of a recurrent neural network or RNN [14]. Karpathy begins by explaining what an RNN is and showing examples for the kinds of problems they can solve. He then proceeds to walk readers through the construction

of an RNN from scratch, with Lua code samples and detailed explanations at every step. Finally, he runs a series of tests demonstrating the effectiveness of his multi-layer RNN by using it to solve a variety of problems. He also provides the source code for his project, which is written in Lua and uses Torch as its back-end [14]. The post is exceptionally useful, because Karpathy goes into a very high level of technical detail without making it difficult to read.

Chapter 3: Methods

3.1 Design

The project consists of several core components: a SerpentAI game agent plugin, a dataset management program, a frame-action synchronizer, a replay parser, and a machine learning program.

The first phase of the project is data preprocessing. This consists of converting the plaintext replay files into a labeled dataset. The dataset management program is used to move all of the replay files into folders corresponding with their game versions. This results in the following directory structure:

```
replays
├─ 00_15_07
├─ 00_15_08
├─ 00_15_09
├─ 00_15_10
├─ 00_15_11
├─ 01_00_02
├─ 01_00_03
├─ 01_00_05
├─ 01_01_02
├─ 01_02_01
└─ 01_02_02
```

In the example above, *00_15_07* refers to game version 0.15.7, *01_02_01* to 1.2.1, and so on. The game version matters because replays are not compatible across different game versions. If a replay has a stage or character that does not exist in the running version of the game, then the replays menu will actually crash when loaded. Thus, with all of the replay files sorted in the manner shown above, frame collection is made possible.

Frame collection is one of the tasks fulfilled by the SerpentAI game agent. This consists of playing back each replay for the currently installed version of the game, one at a time, while dumping the frame buffer to the file system via serialization. The game agent reliably controls playback using simple timed input sequences. The duration of a replay is not explicitly given by a replay file, but can be estimated with sufficient accuracy by identifying the highest frame index and then converting that number to seconds. For

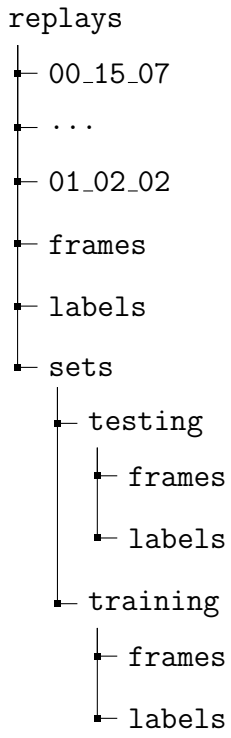
example, the final action for player 1 may happen on frame 8,961, while the final action for player 2 is on 8,954. Given a frame rate of 60 frames per second, this means that player 1 stopped playing after 149.35 seconds and player 2 after 149.23. Thus, the match duration was almost certainly 150 seconds. Knowing this, the game agent can count to 150 while collecting frames. After the match has ended, the agent can stop collecting frames and then use timed inputs to initiate playback of the next replay. Over time, a file structure resembling this will develop:

```
replays
├─ 00_15_07
├─ ...
├─ 01_02_02
├─ frames
│  ├─ 2017-11-07-234316128959
│  ├─ ...
│  └─ 2017-11-07-234241093349
│     ├─ 0.npy
│     ├─ ...
│     └─ 4475.npy
└─ labels
   ├─ 2017-11-07-234316128959
   ├─ ...
   └─ 2017-11-07-234241093349
      ├─ roa_1.npy
      └─ roa_2.npy
```

Alongside the version folders there are now two new folders called *frames* and *labels*. Each frame collected by the agent gets dumped into its own *x.npy* file located in replays/frames/replay-title. The x in the file name is replaced by a number representing its offset from the first frame. An offset of 1, for example, describes the second frame, while an offset of 99 describes the 100th frame. Meanwhile, each player's actions are converted to a matrix format and then dumped into *replays/labels/replay-title* as *roa_x.npy*, where x identifies the player (e.g. player 1 or player 2).

Actually tracking the replays and moving them into the game's replays folder is handled by the dataset manager. During collection, only 1 replay is allowed in the game's replays folder at a time. Thus, the dataset manager tracks which replays are visited and, when asked, deletes the contents of the replays folder before copying over the next replay. When the manager reports that there are no replays left for the selected game version, the collection agent knows to stop.

When collection is over, the dataset manager can be used to sort all of the data in the *frames* and *labels* folders across the training and testing sets. Invoking this task will result in the following changes to the file structure:

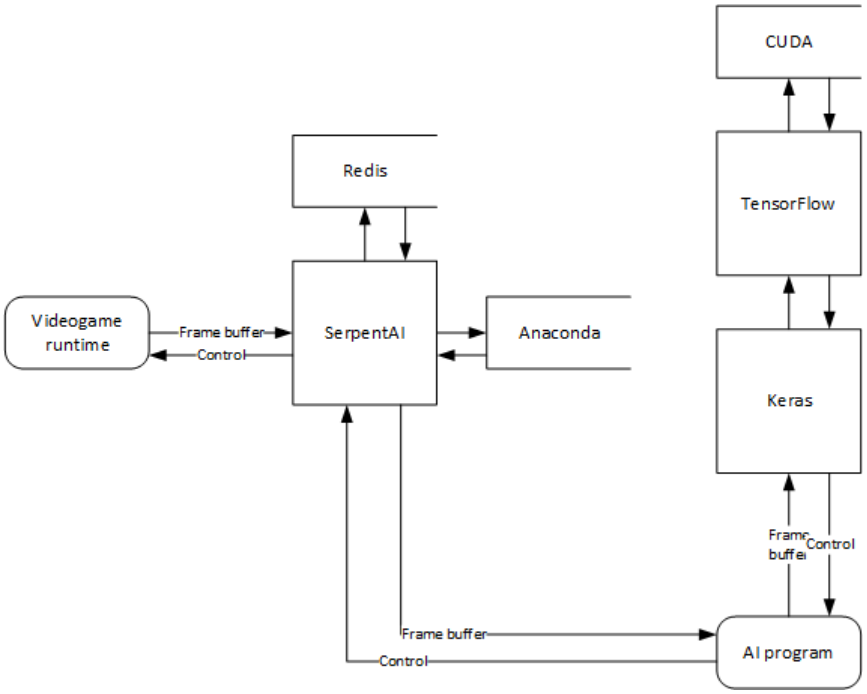


By default, 80 percent of the replays will go into the *training* folder and the remaining 20 percent into *testing*. The machine learning program includes a specialized loader designed to read datasets in the form shown above. In other words, it can load the training set from *replays/sets/training* and the testing set from *replays/sets/testing*. For either set, the loader gathers a list to every file path in the *frames* and *labels* folders. It then starts a worker thread which loads the frame and label data into a queue, which allows the neural network to run its training and testing loops without getting blocked.

There are several expensive operations that the worker thread handles. First, it loads all of the frames from the file system into memory. Then it explodes the label data. Exploding the labels is necessary because there is no guarantee that the frames that were collected are the frames on which actions took place. For example, consider a situation in which frames 6, 17, and 25 were collected by the game agent. According to the replay file, player 1 pressed the jump key in frame 3, released it on frame 15, and then pressed it again on frame 82. The replay file contains no references to the collected frames. One might assume that there is nothing to do; the collected frames are useless. However, it is entirely possible to extrapolate the button states for all of the frames that were collected, even if the replay file contains no references to them. The jump key was pressed on frame 3 and released on frame 15, which means that it was in a pressed state on frame 6. Given that the jump button is not pressed again until 82, it's safe to assume that it is in an unpressed state on frames 17 and 25. If this process is repeated for all 9 buttons across the entire replay, then all of the collected frames can be given accurate labels.

With the replay loader, it is possible to use replays to efficiently train a machine learning

Figure 3.1: Data flow within our software stack



model.

3.2 Frameworks

Our project uses a software stack comprised of the following modules and libraries: SerpentAI, Keras, TensorFlow, and the Anaconda Distribution for Python 3. We chose to use Python 3 as our primary programming language for its flexibility, intuitive syntax, and data processing capabilities. Python is also helpfully compatible with all of the other software listed above. Furthermore, the *Anaconda distribution* provides us with a large assortment of Python modules, some of which *SerpentAI* cites as dependencies [5]. This allows us to more easily work from within a Windows environment, which in turn gives us the most straightforward and stable access to GPU computation via NVIDIA proprietary drivers. TensorFlow serves as the project's backbone by actually creating the neural network in addition to performing the necessary computations [6], with Keras allowing us to perform rapid prototyping in TensorFlow via its abstracted API [3].

All experiments will take place on a pair of 64-bit Windows 10 machines with Ubuntu subsystems. There are several reasons to justify this decision. Firstly, we wanted to ensure that we would have the best possible experience when dealing with GPU computation, and NVIDIA simply has a longer and better track record for supporting Windows than it does for Linux. Secondly, we wanted to avoid the potential pitfall of our platform restricting us to a smaller library of target games. Our two top candidates were Quake and Rivals of Aether because both of these games support replays, or demos; however, of those two games, only Quake natively supports Linux, and ultimately we selected Rivals of Aether.

Lastly, and with regards to using our own computers for both development and testing: we made the decision to abstain from distributed or cloud computing chiefly to save costs, but also as an aesthetic choice based in our desire to push our own hardware to its absolute limits. Despite all of the above justifications for using Windows, it remains to be said that our project cannot live without Linux. SerpentAI requires a Redis database, so in order for that to work we use an Ubuntu subsystem; this is the official recommendation from the SerpentAI developers [5].

3.3 Algorithms

The artificial neural network takes 2 inputs and produces 1 output. These are x, y_1, and y, respectively.

The primary input, x, is of a video clip consisting of 60 gray-scale frames, where each frame is 80 by 45 pixels with 1 color channel. The auxiliary input, y_1, contains the most recent actions taken by the player; these are represented by an array containing 9 integers, where each index of the array is associated with a different action and contains either a 1 if the player took that action or a 0 if not.

The video clips are processed by a stack of four 2D convolutional LSTM layers that are buffered with 2D batch normalization layers. At the same time, the recent actions are fed into a stack of two normal LSTM layers. Next, both stacks are concatenated together. The resulting concatenation is then given to a deep neural network containing RELU activations. Finally, this leads to the output layer, y_1; this layer applies a sigmoid function across 9 nodes, where each node again represents each of the different actions available to the player. In this case, the layer is predicting which actions are about to be taken by the player.

During training and testing, y_1 is always the y from the previous frame. During live gameplay, however, each value from y is converted into either 1 or 0 based on a threshold value. The result of this thresholding is then used as the next y_1.

3.4 Features

In order to play Rivals of Aether, one must use a set of controls containing a total of nine buttons, where five are for actions such as attacking or dodging and four are for movement.

These buttons can be combined for different effects. An attack will undergo significant changes depending on if the attacking character is standing, walking, running, or jumping. Furthermore, tilting the control stick during the attack will effect additional changes. Dodging in the air is different from doing so on the ground, which is different still from dodging while moving the control stick in a direction.

Figure 3.2: High-level overview of the neural network

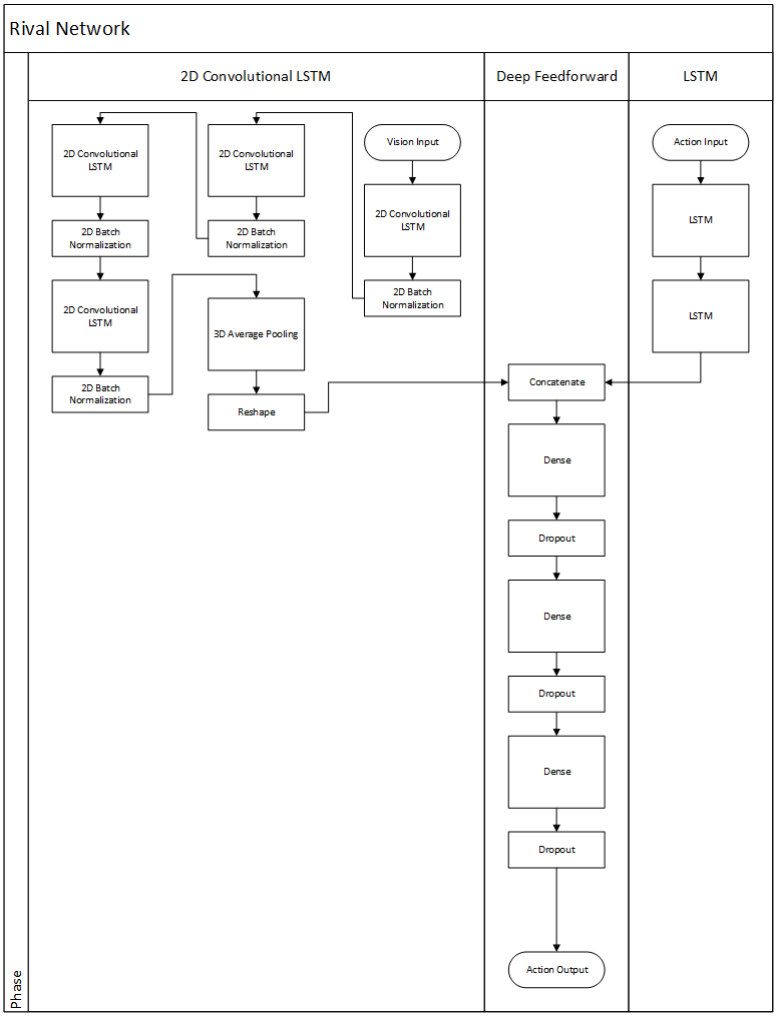


Figure 3.3: In-game menu displaying game controls

Due to the complex nature of the game's control scheme, at any given moment the AI will need to be able to output any combination of two simultaneous button presses, where one of the buttons is for movement and the other is for an action.

3.5 Test Plan

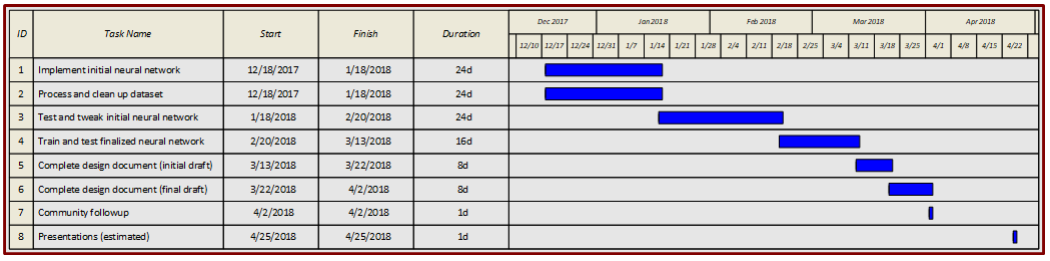
The full dataset contains a total of 1,032 replay files provided by the Rivals of Aether community. Of those, 222 replay files are unusable because they were recorded in pre-release versions of the game that do not appear to normally support replay recording or playback. Given that the replays exist in the first place, it is conceivable that the functionality exists via an undocumented developer menu or console command. No such accesses were found, however. With the unusable 222 replay files removed, the dataset is left with only 810 replays. Leaving out 20 percent of the dataset for testing reduces the training set to a meager 641 replays. This is not as poor as it may seem at first, however. Each replay can have a duration anywhere up to 8 minutes. Thus, with the frame collector running at 10 frames per second, this leads to an upper limit of 4,800 frames sampled per replay. As a result, the training set contains a total of 859,249 frames while the testing set has 221,782.

The amount of data can still be increased, however. Replays contain input data for a minimum of 2 players and a maximum of 4. This one-to-many relation from replays to players can be exploited if each combination of a replay and player is treated as a separate video. In other words, the frame buffer for replay R can be used not only with the input data for player P_1 but also with those of players $P_{2\text{-}4}$. Exploding the dataset in this fashion yields a total of 1271 replays for training and 332 replays for testing.

A training session will run as follows. The SerpentAI agent will traverse the game's menus to initiate playback of the next replay from a shuffled copy of the training set. While the match is simulated, the agent will simultaneously read the visual buffer from the game and control input data from the replay file. For each frame, the agent will pair the input data with its corresponding visual buffer and then send this bundle to a TensorFlow neural network as labels and data, respectively. The neural network will attempt to produce a set of appropriate control inputs in response and adjust its weights whenever its prediction are incorrect. When playback has concluded, the agent will traverse the menus once again to start a new replay, repeatedly, until it has reached its quota. The total number of training iterations will vary between experiments until it produces satisfactory results.

A properly trained neural network shall demonstrate several qualities other than a high accuracy when tested against the held back data. Firstly, it shall be able to defeat low-level in-game bots, thereby demonstrating its competence. This can be tested by placing the neural network in matches against bots of increasing difficulty, thereby enabling the assignment of numerical performance benchmarks. Secondly, it shall play in a believably-human manner. This means that it should, wherever possible, be discouraged from playing *inhumanly* well, getting stuck, or displaying mechanical tics. Examples of these undesirable behaviors would include, in order, executing nonstop frame-perfect moves throughout the course of an entire game, running off of the stage or standing still due

Figure 3.4: Gantt chart



ID	Task Name	Start	Finish	Duration
1	Implement initial neural network	12/18/2017	1/18/2018	24d
2	Process and clean up dataset	12/18/2017	1/18/2018	24d
3	Test and tweak initial neural network	1/18/2018	2/20/2018	24d
4	Train and test finalized neural network	2/20/2018	3/13/2018	16d
5	Complete design document (initial draft)	3/13/2018	3/22/2018	8d
6	Complete design document (final draft)	3/22/2018	4/2/2018	8d
7	Community followup	4/2/2018	4/2/2018	1d
8	Presentations (estimated)	4/25/2018	4/25/2018	1d

to an as-of-yet unseen situation, or constantly changing direction for no reason while otherwise playing competently. Identifying all such behaviors will be difficult prior to live experiments, however.

3.6 Criteria and Constraints

Initial work on the neural network implementation is due to start on December 12, 2017, shortly after the conclusion of the 2017 Fall Semester. This work should be concluded by mid-March. From there, about two weeks are allocated for finalizing the design document. Overall, the schedule is intentionally tight in order to emphasize getting most of the work done in the beginning of the semester, thereby maximizing flex time and minimizing the competition between this project and other finals.

The project has other constraints in addition to its schedule. The dataset was collected from the community with the promise that each replay file we receive will be treated as private data and not, under any circumstances, be redistributed. Thus, care shall be taken to ensure that no replay files from the community are leaked. At the same time, however, these files must not be lost due to accidental deletion, disk failure, or lack of backup. To address these requirements, multiple redundant copies of the dataset are stored on our own local drives as well as via encrypted, private, cloud storage services such as Google Drive. Only the dozen or so replay files that we brought to the dataset ourselves may be stored publicly on the GitHub repository.

Our experiments will be deemed adequately successful if we can produce AI that beat the in-built bots on lowest difficulty levels. This is considered a baseline that we will attempt to surpass through improvements to the neural network and additional training.

Chapter 4: Results

4.1 Analytical Results

The neural network has been trained and tested on the full dataset over 1 epoch. The test results showed a mean accuracy of 30 percent with a standard deviation of 9 percent. In other words, the model was usually able to predict between 2 and 4 actions out of 9.

4.2 Testing Results

This section is intentionally blank.

Chapter 5: Conclusion

5.1 Context

This section is intentionally blank.

5.2 Challenges and Solutions

This section is intentionally blank.

5.3 Limitations and Delimitations

This section is intentionally blank.

5.4 Future Work

This section is intentionally blank.

5.5 Project Importance

This section is intentionally blank.

References

[1] "Alphago." [Online]. Available: https://deepmind.com/research/alphago/

[2] "Anaconda: The most popular python data science platform." [Online]. Available: https://docs.anaconda.com/

[3] "Keras: The python deep learning library)." [Online]. Available: https://keras.io/

[4] "Rivals of aether." [Online]. Available: http://www.rivalsofaether.com/

[5] "Serpent.ai - game agent framework (python)." [Online]. Available: https://github.com/SerpentAI/SerpentAI

[6] "Tensorflow: An open-source software library for machine intelligence)." [Online]. Available: https://www.tensorflow.org/

[7] "Solving ai's moving-target search problem at ijcai 2017," 2017. [Online]. Available: https://www.ibm.com/blogs/research/2017/08/ai-moving-target-search/

[8] "Visual doom ai competition 2016 @ cig," 2017. [Online]. Available: http://vizdoom.cs.put.edu.pl/competition-cig-2016/results

[9] "Visual doom ai competition 2017 @ cig," 2017. [Online]. Available: http://vizdoom.cs.put.edu.pl/competition-cig-2017/results

[10] S. Alvernaz and J. Togelius, "Autoencoder-augmented neuroevolution for visual doom playing," *CoRR*, vol. abs/1707.03902, 2017. [Online]. Available: http://arxiv.org/abs/1707.03902

[11] D. S. Chaplot and G. Lample, "Arnold: An autonomous agent to play fps games." [Online]. Available: http://www.cs.cmu.edu/~dchaplot/papers/arnold_aaai17.pdf

[12] A. Dosovitskiy and V. Koltun, "Learning to act by predicting the future," *CoRR*, vol. abs/1611.01779, 2016. [Online]. Available: http://arxiv.org/abs/1611.01779

[13] B. Gorman and M. Humphrys, "Towards integrated imitation of strategic planning and motion modeling in interactive computer games," *Comput. Entertain.*, vol. 4, no. 4, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1178418.1178432

[14] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks," 2015. [Online]. Available: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[15] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski, "Vizdoom: A doom-based AI research platform for visual reinforcement learning," *CoRR*, vol. abs/1605.02097, 2016. [Online]. Available: http://arxiv.org/abs/1605.02097

[16] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," *CoRR*, vol. abs/1609.05521, 2016. [Online]. Available: http://arxiv.org/abs/1609.05521

[17] P. K. K. Loh and E. C. Prakash, "Novel moving target search algorithms for computer gaming," *Comput. Entertain.*, vol. 7, no. 2, pp. 27:1–27:16, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541895.1541907

[18] M. D. Marr, K. S. Kaplan, and N. T. Lewis, "System and method for driving microtransactions in multiplayer video games," no. 9789406, 2017. [Online]. Available: http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=9,789,406.PN.&OS=PN/9,789,406&RS=PN/9,789,406

[19] SethBling, "Mariflow - self-driving mario kart w/recurrent neural network," 2017. [Online]. Available: https://www.youtube.com/watch?v=Ipi40cb_RsI

[20] C. Thurau and C. Bauckhage, "Learning human-like movement behavior for computer games," in *In Proceedings of the Eighth International Conference on the Simulation of Adaptive Behavior (SAB04*, 2004.

Appendix A: Additional Information

As necessary.