Construction of an AI player for historic arcade games using object recognition and reinforcement learning

A dissertation submitted in partial fulfilment of the requirements for the MSc in Advanced Computing Technologies

by Henrik Bilar

Department of Computer Science and Information Systems

Birkbeck, University of London

September 2019

This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the TURNITIN Plagiarism Detection Service. I have read and understood the sections on plagiarism in the Programme Handbook and the College website.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

Table of Contents

[List of Figures 6](#_Toc19483565)

[List of Tables 6](#_Toc19483566)

[Acknowledgements 7](#_Toc19483567)

[1. Introduction 8](#_Toc19483568)

[2. Analysis and Design 8](#_Toc19483569)

[2.1 High-level requirements for the system 8](#_Toc19483570)

[2.2 Logical data flows between high level components 9](#_Toc19483571)

[2.3 Components of the Python player process 9](#_Toc19483572)

[2.4 Speed of the system 12](#_Toc19483573)

[2.5 Image training set 12](#_Toc19483574)

[3. Implementation 12](#_Toc19483575)

[3.1 Modified emulator with API 12](#_Toc19483576)

[3.1.1 Emulator GUI removal and command line parameter handling 12](#_Toc19483577)

[3.1.2 Emulator TCP port connections and communication 13](#_Toc19483578)

[3.1.3 Emulator with API: Sending screen contents 13](#_Toc19483579)

[3.1.4 Emulator with API: Handling player input 14](#_Toc19483580)

[3.2 Python game play process – brief overview 14](#_Toc19483581)

[3.2.1 Python game play process: Augmented human player mode 15](#_Toc19483582)

[3.2.2 Python game play process: Full AI-mode 15](#_Toc19483583)

[3.2.3 Full AI-mode NES game start-up process 16](#_Toc19483584)

[3.2.4 Detecting the end of a game in full AI-mode 16](#_Toc19483585)

[3.3 Object detection using TensorFlow 16](#_Toc19483586)

[3.4 Object detection using pixel comparison 17](#_Toc19483587)

[3.5 Object detection of numbers and words 19](#_Toc19483588)

[3.6 Creating a dataset for object detection 20](#_Toc19483589)

[3.7 Training the TensorFlow model to recognize objects 20](#_Toc19483590)

[3.8 Detecting how far Mario has run to the right using a reward function 22](#_Toc19483591)

[3.9 The gameplay neural network 23](#_Toc19483592)

[3.9.1 GPNN: Neurons into layers appropriate for the AI-player 23](#_Toc19483593)

[3.9.2 GPNN: Generating an action in the game 24](#_Toc19483594)

[3.10 Training the gameplay neural network 24](#_Toc19483595)

[3.10.1 Negnetvitsky’s Genetic Algorithm for neural network training 25](#_Toc19483596)

[3.10.2 Simplified Genetic Algorithm 25](#_Toc19483597)

[3.10.3 Generating new networks from a parent 26](#_Toc19483598)

[3.10.4 Automating GPNN training 26](#_Toc19483599)

[3.10.5 GPNN: Layout evaluation and playing performance 27](#_Toc19483600)

[3.10.6 The effect of timing jitter on training and GPNN performance 28](#_Toc19483601)

[4. Evaluation of project 29](#_Toc19483602)

[4.1 Object recognition 29](#_Toc19483603)

[4.2 Gameplay neural network performance 29](#_Toc19483604)

[4.3 Emulator with API capabilities 30](#_Toc19483605)

[5. Potential improvements 30](#_Toc19483606)

[5.1 Headless mode 30](#_Toc19483607)

[5.2 Frame synchronization between emulator and AI-player 31](#_Toc19483608)

[5.3 GPNN training – learning from a human 31](#_Toc19483609)

[5.4 GPNN performance – reducing jitter 31](#_Toc19483610)

[6. Issues encountered 31](#_Toc19483611)

[6.1 Numpy matrix cell addressing 31](#_Toc19483612)

[6.2 TensorFlow object detection training speed 32](#_Toc19483613)

[6.3 The reward function 32](#_Toc19483614)

[7. Final conclusions 33](#_Toc19483615)

[References 34](#_Toc19483616)

[Appendix A: Instructions for using the software 35](#_Toc19483617)

[Appendix A.1: Software Pre-requirements 35](#_Toc19483618)

[Appendix A.2: Installation of AI PLAYER software 35](#_Toc19483619)

[Appendix A.3: Installation of the NES Emulator 35](#_Toc19483620)

[Appendix A.4: Running the NES Emulator 36](#_Toc19483621)

[Appendix A.5: Running the player 36](#_Toc19483622)

[Appendix B: LaiNES source code modifications 37](#_Toc19483623)

[Appendix B.1 LaiNES main.cpp 37](#_Toc19483624)

[Appendix B.1.1 general changes to main.cpp 38](#_Toc19483625)

[Appendix B.1.2 Set up TCP listening socket 38](#_Toc19483626)

[Appendix B.1.3 Command line parameter handling 39](#_Toc19483627)

[Appendix B.2 LaiNES config.cpp 40](#_Toc19483628)

[Appendix B.3 LaiNES gui.cpp 40](#_Toc19483629)

[Appendix B.3.1 general changes to gui.cpp 41](#_Toc19483630)

[Appendix B.3.2 Functions to send data to the player 41](#_Toc19483631)

[Apppendix B.3.3 Handling of remote commands 43](#_Toc19483632)

[Appendix B.3.4 Get state of the remote joypad 44](#_Toc19483633)

[Appendix C: The AI-player: player.py 45](#_Toc19483634)

[Appendix C.1: player.py : globals, imports, and enums. 45](#_Toc19483635)

[Appendix C.2: player.py : drawing to screen 50](#_Toc19483636)

[Appendix C.3: player.py : Connect to emulator and send TCP data 50](#_Toc19483637)

[Appendix C.4: player.py : Get emulator screen data 51](#_Toc19483638)

[Appendix C.5: player.py : Send key press, power off, and reset messages 52](#_Toc19483639)

[Appendix C.6: player.py : Taking screenshots 53](#_Toc19483640)

[Appendix C.7: player.py : TensorFlow object detetion 53](#_Toc19483641)

[Appendix C.8: player.py : Pixel based word and number object detetion 55](#_Toc19483642)

[Appendix C.9: player.py : Detection of holes 56](#_Toc19483643)

[Appendix C.10: player.py : Check screen scroll 57](#_Toc19483644)

[Appendix C.11: player.py : Check for blocks in front of Mario 58](#_Toc19483645)

[Appendix C.12: player.py : Find horizontal objects in front of Mario 58](#_Toc19483646)

[Appendix C.13: player.py : Filter false TensorFlow matches 59](#_Toc19483647)

[Appendix C.14: player.py : Draw bounding boxes 60](#_Toc19483648)

[Appendix C.15: player.py : Start sequence 61](#_Toc19483649)

[Appendix C.16: player.py : Build key state dictionary 61](#_Toc19483650)

[Appendix C.17: player.py : Normalise object detection boxes 62](#_Toc19483651)

[Appendix C.18: player.py : Main player loop 62](#_Toc19483652)

[Appendix C.19: player.py : Initialization of the player process 67](#_Toc19483653)

[Appendix D: gameplay.py : Neural network function for game-play 68](#_Toc19483654)

[Appendix D.1: gameplay.py : Globals and imports 68](#_Toc19483655)

[Appendix D.2: gameplay.py : Sigmoid function 68](#_Toc19483656)

[Appendix D.3: gameplay.py : Neuron class 68](#_Toc19483657)

[Appendix D.4: gameplay.py : Example neural net 69](#_Toc19483658)

[Appendix D.5: gameplay.py : Save and load networks to file 70](#_Toc19483659)

[Appendix D.6: gameplay.py : Processing inputs and generating an output using neural net 71](#_Toc19483660)

[Appendix D.7: gameplay.py : Convert a game state to an action 71](#_Toc19483661)

[Appendix E: Generate modified networks with generate\_network.py 72](#_Toc19483662)

[Appendix F: Automate training of GPNN with train-player.sh 73](#_Toc19483663)

[Appendix G: Automatic generation of training data with create\_training\_data.py 76](#_Toc19483664)

[Appendix H: Automating the automatic generation of training data images 79](#_Toc19483665)

[Appendix I: Script to split training into training data and test data 80](#_Toc19483666)

[Appendix J: Bash Shell Wrapper for training TensorFlow 81](#_Toc19483667)

[Appendix K: Pipeline config for TensorFlow object detection training 81](#_Toc19483668)

# List of Figures

Figure 2.1: Core components of the system

Figure 2.2: Logical steps of the Python player process

Figure 3.1: Screenshot showing large areas of identical pixels

Figure 3.2: Augmented image of gameplay with object bounding boxes

Figure 3.3: Outputs shown after a game has finished

Figure 3.4: "Koopa-Troopa" enemy and "Question Mark" block similarities

Figure 3.5: Pipe section with colours shown

Figure 3.6: Obstacles in the game are often grouped together

Figure 3.7: Data structure for simple object detection

Figure 3.8: Screenshot of the beginning of game level identifier

Figure 3.9: Mean Average Precision graph for TensorFlow training

Figure 3.10: Test set image with ground truth information and the detected object

Figure 3.11: Detail of the pixels at the bottom of the game

Figure 3.12: Neuron layer data structure

Figure 3.13: Graphical representation of the gameplay neural network

Figure 6.1: Mario gets stuck next to an object due to incorrect reward function

# List of Tables

Table 3.1: Jitter in the performance of the gameplay neural network

# Acknowledgements

Working on this project has been a very rewarding experience, but it would have been impossible to complete without the help from some very important people.

Firstly, I would like to thank my project supervisor Professor Steve Maybank for his excellent guidance on report writing and reviewing my many incomplete drafts swiftly. Without his input, this report would no doubt have looked very different.

I would also like to extend my sincere gratitude to the many, many participants in the Open Source community, most of whom I have never met. Without these companies and individuals - often working without financial gain - software development today would be a much poorer experience. I have leveraged many common libraries and tools, but one component was critical for the success of the project; the LaiNES Nintendo Entertainment System emulator, developed by Andrea Orru.

Lastly, but most importantly, I would like to thank my wife and kids for offering support, listening to my ramblings about neural networks, and putting up with a grump-o-saurus during difficult parts of the project.

# 1. Introduction

In 1983 Nintendo released its “Nintendo Entertainment System” (NES) games console (Wikipedia, 2019). A large variety of games were produced for the NES, and there were over 61 million NES consoles sold worldwide.

The console is connected to a TV to display images. The player controls the game using a handheld controller known as a “joypad”. The joypad has directional left/right/up/down type button (where two adjacent directions buttons can be pressed at once), and also an A and a B button. There is also a “start” and a “select” button, but these are typically not used in game play.

Even though the NES console itself has not been on sale for many years, it is still possible to play NES games today using one of a variety of software emulators. A NES software emulator accurately simulates the original NES console, with the original TV display rendered on a computer screen, allowing a player to play the games.

The goal of this project is to create a stand-alone AI program, capable of acting as a player in the well-known game Super Mario Brothers. The program connects to an emulator, and issues game-play commands in a similar way to a human player. For the AI player to do this, it must understand the world, at least in terms of objects and enemies, so that it can take appropriate action at all times.

Various companies and individuals have already created AI players for various vintage games. For example, the company Deepmind has successfully played many Atari games using reinforcement learning (Mnih, et al., 2013). Most of the implementations of AI players feed the pixel state of several game frames into a deep neural network, without attempting to identify objects per se, however this project has taken a different approach in using TensorFlow (TF) for object recognition in the video feed, and then feeding object locations into a shallow game-play neural network, thus emulating the process of a real human player observing the state of the world and issuing commands based on this.

# 2. Analysis and Design

## 2.1 High-level requirements for the system

The following high-level requirements were identified as necessary for fulfilling the project goal:

- Speed of the whole system must be near real time, so that it “looks” as if a real player is playing the game.

- An emulator controllable from within Python

- A Python player process that:

- connects to the emulator.

- allows a player to manually play the game.

- performs object recognition of interesting objects in the game.

- has a neural net that generates actions given detected object state.

- understands when the game being played has finished, so that it can terminate Python player process.

- can calculate how well a neural network plays the game, to facilitate training of neural networks.

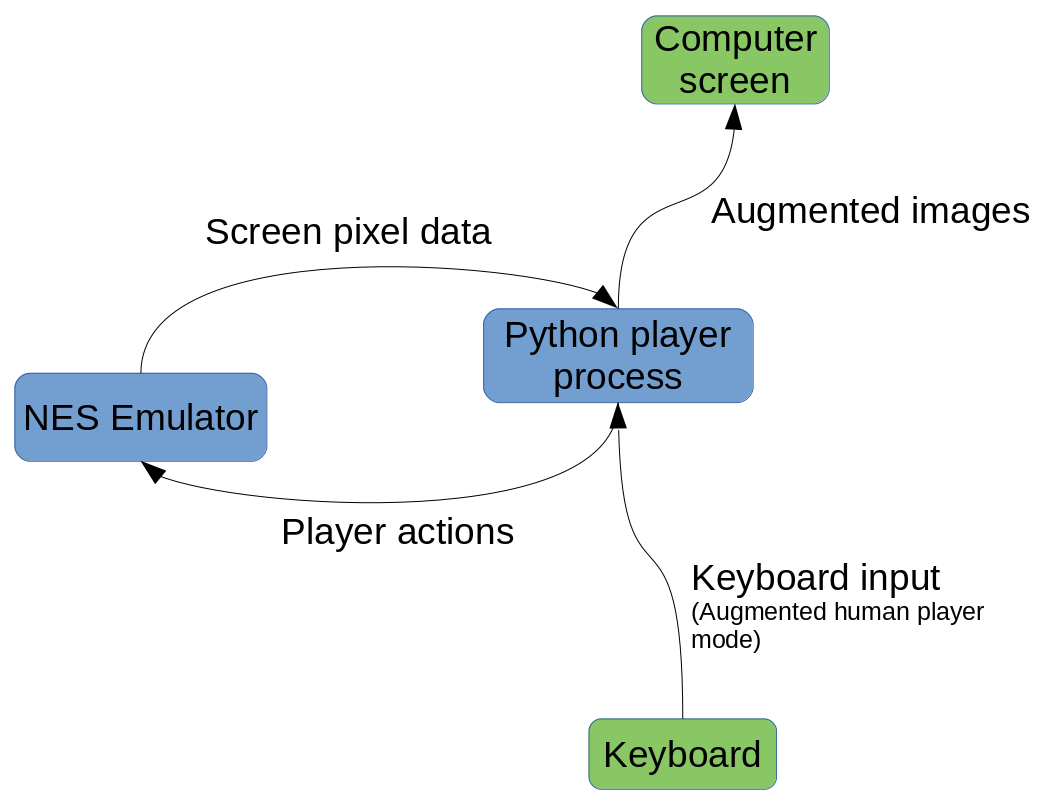
- Creation and training of an image data set.

## 2.2 Logical data flows between high level components

The system consists of two main components, and two auxiliary input and output devices.

Figure 2.1 shows the logical data flows between components in the system. The NES Emulator sends screen data to the Python player process, which augments the images with object bounding boxes, and displays the augmented image to screen.

Figure2.1 - Core components, in blue, with relevant data flows indicated. Keyboard input is only considered in Augmented human player mode.



The Python player process sends player actions to the emulator. In augmented human player mode, the keyboard provides the source for these actions, where-as in AI player mode, a neural net generates actions, which are then passed to the emulator.

## 2.3 Components of the Python player process

Figure 2.2 shows a high-level overview of the steps taken by the Python process to analyse the game scene and produce a game action.

The steps detailed below are run in a continuous cycle until the game ends.

Step 1: Pixel data of the game screen is transferred to the Python process from the emulator using a TCP/IP network connection.

Step 2 and 3: A series of functions analyse the pixel data to identify features. TF object detection is used to identify Mario and any potential enemies in the game. Simple pixel comparison functions are used to identify pipes, blocks, and numeric information.

Step 4 and 5: If the game is played in AI-mode, the identified objects and features are sent to the game-play neural network, which generates a game-play action. If the game is played in augmented human player mode, the keyboard is checked for input.

Step 6: The game play decision generated in steps 4 and 5 is sent to the emulator, after being translated to a joypad status code.

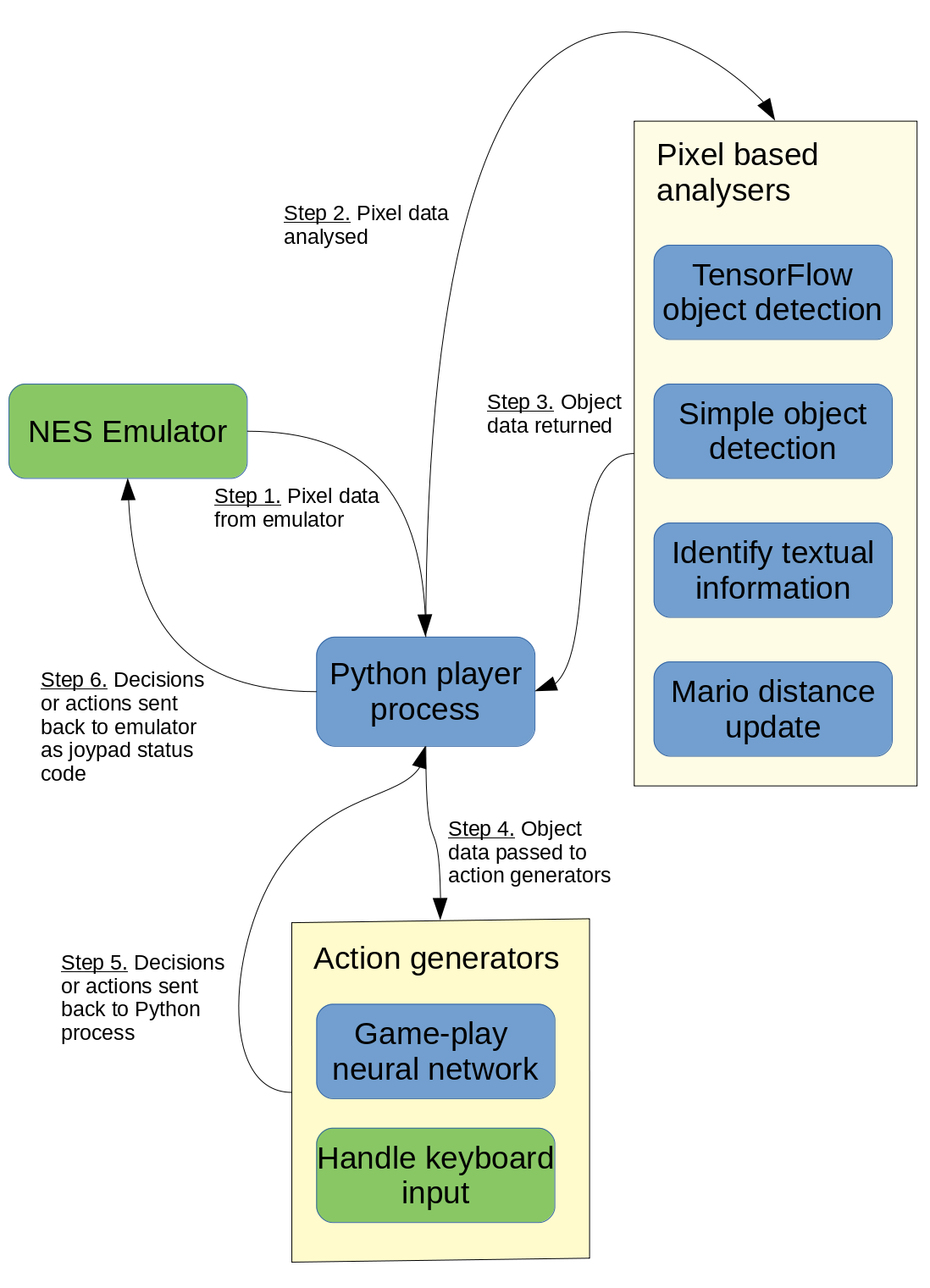


Figure 2.2 - The discrete steps of the Python player process. Pixel data is sent from the emulator. The pixel data is then analysed for features, and a list of features sent to the neural network. The neural network produces an action, which is sent to the emulator.

## 2.4 Speed of the system

In order to accurately simulate an actual player, the AI player has to interact with the game at roughly the same speed as a human player would. If the screen updates are too jerky or delayed, the theatricals of seeing the computer “play a game of Mario” are lost.

The speed requirement has implications for all the components involved in the whole project, and in some instances, accuracy has been sacrificed for speed. For example, the TensorFlow object detection model chosen is not as accurate as the most accurate models, but it provides reasonable accuracy whilst being fast enough to be usable for this project.

## 2.5 Image training set

TensorFlow object detection models use complicated neural networks for object recognition within images. In order to train the neural networks accurately, a large amount of labelled training must be made available to the network during a training-phase.

Generating the training set manually will take a long time, so a piece of software that can generate labelled training data automatically will be created.

# 3. Implementation

## 3.1 Modified emulator with API

There are various Open Source NES emulators available, but they do not typically provide an application programming interface (API) for connecting to them to provide player actions or provide a stream of images. To overcome this, the LaiNES (Orru, 2017) emulator was modified to provide a simple API over a network connection, using TCP/IP.

The LaiNES emulator was chosen because it is written in C, provides enough emulation of the NES console to play most games (Orru, 2017), and is fast enough to support the extra load of the network API and still play the NES game in real time.

The modifications to the emulator are:

* Removal of graphical menu system, which in the original emulator is used to select the file containing the NES game.
* TCP based API for controlling and communicating with the emulator. This includes functionality to perform tasks such as “press a joypad button” and “send screen contents to AI-player process”.
* Addition of command line parameter handling, so that the emulator can be started from a command line, with the correct game file loaded, listening to a specific TCP port.
* Removal of all functions and routines dealing with sound.

### 3.1.1 Emulator GUI removal and command line parameter handling

The unmodified LaiNES emulator does not handle command line parameters. Once the emulator is started, the user must use the arrow-keys on the keyboard to navigate a graphical user interface (GUI) and select a game file from a list.

The lack of command line parameters makes the original LaiNES emulator unsuitable for automation using Bash shell command scripts, and therefore command line parameter handling was added. The command line parameters allow a user, or other process, to start the LaiNES emulator with a specific game file loaded, and enable the emulator to listen to connections on a TCP port.

As the GUI portion is no longer needed, it was completely removed.

The source code for the command line parameter handling is available in Appendix B.1.3, in the function main().

### 3.1.2 Emulator TCP port connections and communication

The modified emulator can optionally accept network connections on TCP port 8005, where a player process can connect to issue text-based commands. To enable the TCP port connections, the -tcp command line parameter must be passed to LaiNES.

If the emulator is instructed to use TCP connections, it will wait to start executing the game until a TCP connection is initiated by the Python player process. This behaviour is useful for synchronising the game being run in the emulator, and the Python player process. With the pause, the AI-player connects to the emulator with the game in a well-defined state.

For the player process to be able to connect to the emulator using TCP connections, the emulator must open up a network socket, and listen to it. The code for opening a socket and listening to it is available in the function wait\_for\_client\_connection() in Appendix B.1.2. The wait\_for\_client\_connection() function waits until a connection has been established and sets the socket to non-blocking mode before returning. The non-blocking mode allows the emulator to attempt reading data from the socket using the read() system call, without it blocking if there is no data to read.

Once the game is running, the emulator will periodically check if any data has been received from the AI-player process on the TCP connection. If new data is available, it is added to an in-memory buffer until a complete message has been transmitted from the AI-player. Complete messages are read and removed from the in-memory buffer, after which they are processed.

Messages are read from the AI-player using the function handle\_remote\_input() (Appendix B.3.3). The handle\_remote\_input() function attempts to read up to 1000 bytes of data from the TCP connection, and if it succeeds in reading any, the data is added to a global data buffer. The function then attempts to detect complete messages in the global data buffer, and if one is found it is handled by the handle\_remote\_command() function (Appendix B.3.3).

The handle\_remote\_command() function deciphers the messages, and perform the appropriate action. Messages can:

* modify the joypad state.
* cause the emulator to respond with a textual, or binary, representation of the pixels representing the screen.
* reset the game being emulated, so that it starts from the beginning again.
* Turn the emulator off.

### 3.1.3 Emulator with API: Sending screen contents

To avoid overwhelming the AI player process with screen data, the screen content sends are performed on request only, rather than the emulator sending every frame across the network. The AI player process can request a copy of the NES screen contents using the API.

The NES screen resolution is 256\*240 pixels (width\*height). This is represented inside the emulator as a 256\*240\*4-byte array. The red, green and blue values of the pixels are represented by a full byte each, and the fourth byte is not used. The effective data amount in the array is thus 184320 bytes.

The API supports sending the screen contents in both binary and text modes. Text mode is useful for debugging purposes, as the textual information is easy to observe and interpret. Unfortunately, text mode transfer is also inherently slow both on the sending side and on the receiving side, as the TCP byte stream must be converted first to text, and then from text back to binary representation. This makes it too slow to use as a real time video stream.

Binary mode transfer avoids all the string conversions and is therefore much faster. In binary mode, the raw pixel values are extracted from the 4-byte pixel value using fast and simple bitwise operators. The code for send\_binary\_screen\_to\_remote() (Appendix B.3.2) is close to as simple as possible whilst still performing the task correctly, and it contains no optimizations at all.

The game frames in Super Mario Brothers typically contains big areas of identical pixels (See figure 3.1), and performing even a simple run-length-encoding compression (RLE) (Wikipedia, 2019) on the pixel values would probably yield significant compression of the amount of transferred data, but in practice the naive method works fast enough for the simple AI player.

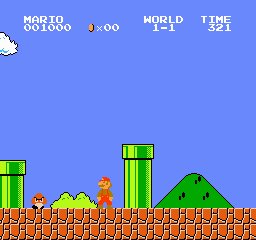


Figure 3.1 - Screenshot from game, showing large number of similar coloured pixels. Should RLE type compression be used, the amount of data transferred would shrink significantly.

### 3.1.4 Emulator with API: Handling player input

The API allows the AI-player to send joypad states to the emulator. The joypad state is represented by the emulator as a simple byte, defined in get\_joypad\_state() (Appendix B.3.4) and the API expects the AI player to calculate the correct bit pattern. The remote player bit pattern is simply bitwise-OR’d onto the emulator’s internal representation of the joypad, so that when the NES game reads the joypad status, it gets the state sent by the AI player. This is the process that allows the AI Player to “act” like a real player.

## 3.2 Python game play process – brief overview

The game-play process is implemented in Python. It connects to the emulator network port and uses the PyGame library (pygame.org, 2019) to draw content on screen.

The game-play process has two modes of operation. It can act in full Artificial Intelligence player mode (AI-mode), or also in an augmented human player mode (AHP). In both modes, the process connects to the emulator using the API, and it also displays the game screen contents on the computer’s screen.

The Augmented Human Player mode was the first mode to be implemented, as it allows verification of object detection accuracy, and verification of the game-state, which is passed to the game-play neural network in AI-mode (Chapter 3.9). AHP mode additionally allows a user to generate screenshots of the game, which are then used to create a training set for the TensorFlow object detection training (Chapter 3.6).

### 3.2.1 Python game play process: Augmented human player mode

In AHP mode, the Python process performs object detection, as described in chapter 3.3. The Python process augments the displayed screen by drawing bounding boxes around enemies, as shown in figure 3.2. All game play inputs are generated by the human player. This mode is useful for testing the object detection, and it also allows a human player to generate screenshots, which can be used as a base for the data set used for training of the object detection model.

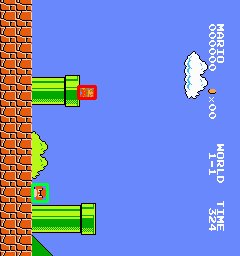


Figure 3.2 - Augmented game play showing Mario in a red bounding box and a “Goomba” in a green bounding box.

### 3.2.2 Python game play process: Full AI-mode

In full AI-mode, the player process collects the list of detected objects and their locations. The locations are translated into coordinates relative to Mario, normalised to a scale of [-1, 1], where the unit is the width or height of the screen (256 or 240 pixels). The relative locations are fed into a rudimentary artificial neural net (ANN), which produces an action for Mario to perform.

The AI-mode play terminates once Mario has lost his first life or completed the level.

Upon termination, the number of ‘game-play’ seconds left until time-up is printed, along with the number of pixels Mario has moved towards the, as shown in figure 3.3. The pixel number is used as a reward function (Chapter 3.8) in the training of the game play ANN.



Figure 3.3 - Output on terminal after the play in AI-mode has completed.

### 3.2.3 Full AI-mode NES game start-up process

In full-AI mode, a human cannot interact with the AI-player process, emulator, or game. As most games require a sequence of joypad buttons to be pressed before the game will start, the player process performs a start-up initialization routine with the emulator. In the case of Super Mario Brothers, the required sequence is:

* Wait for 1 second after connection to emulator established. This allows the game some time to start executing on the emulator.
* Send the key-code corresponding to the “start” button on the joypad.
* Wait for 1 second.
* Send the key-code corresponding to no buttons pressed on joypad. This simulates a human releasing the start button on the joypad.

The start-up sequence is handled by the do\_start\_sequence() function in player.py. The source-code is available in Appendix C.15.

### 3.2.4 Detecting the end of a game in full AI-mode

The AI-player process must terminate when Mario dies in the emulated game, or when he clears the first level. As the AI-player is not able to directly inspect the game-state in the game running in the emulator, it must be able to tell when to terminate from observing the pixel data.

Super Mario Brothers displays a black screen with the string ‘WORLD’ followed by a number when Mario has died or cleared a level. The method described in chapter 3.5 is used to detect this type of screen, and the function check\_black\_screen\_text() in player.py is used by the main\_loop() function in player.py to monitor for this condition.

When the ‘WORLD’ screen has been seen, the player process will exit with game-play statistics, as described in 3.2.2.

The source-code for the function check\_black\_screen\_text()is available in Appendix C.8, and the source-code for the main\_loop()is available in Appendix C.18.

## 3.3 Object detection using TensorFlow

In-game objects are detected using a mix of TF object-detection models trained on a custom training set (Google, 2019), and a simple algorithm involving checking individual pixels, as described in chapter 3.4.

Various pre-developed object detection models are available for download from the TensorFlow Object Detection Zoo (Google / TensorFlow, 2019). As the model detection API is consistent, a piece of software can evaluate different models by simply swapping out the TF model used.

The TF models vary in both speed and accuracy, and this project evaluated two object detection models; faster\_rcnn\_resnet101\_coco and ssdlite\_mobilenet\_v2\_coco. Both models were pre-trained on the publicly available CoCo image set (cocodataset.org, 2019), in the hope that training on the custom dataset would be speed up by means of transfer-learning (machinelearningmastery.com, 2019).

Testing showed that although the faster\_rcnn model was accurate, each frame took more than 1 second to analyse, which is far too slow for an interactive game. The ssdlite model is not quite as accurate. For example, the blocks with the question marks are sometimes mistaken for Koopa-troopa enemies, probably because the question mark looks similar to the neck of the Koopa-troopa (See figure 3.4).

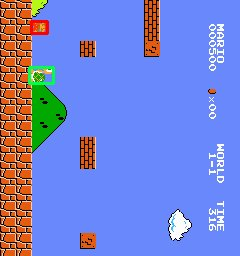


Figure 3.4 - Question mark block and Koopa-troopa have similarities (according to SSD-mobilenet).

Additional code, in Appendix C.13) was added to the player to verify if a detected Koopa-troopa is in fact a real enemy or a question mark. The algorithm simply looks for the colour green inside the bounding box, and if none is present, the bounding box is not that of a Koopa-troopa.

Although the ssdlite model gives up some accuracy compared to faster\_rcnn, it still performs better overall for the purpose of game playing, as it is much faster. The player process achieves about 9.7 frames per second overall, including object detection and game play decisions, which is adequate for the purpose of playing the game.

The TF object detection has proven problematic for detecting objects that vary in length or width. For example, a pipe can vary in height. When using a trained model for these kinds of objects, the pipe is often reported as several different objects with corresponding bounding boxes.

Occasionally the TF object detection models will completely miss objects in an image that is superficially very similar to an image where the objects are detected correctly. For example, Mario moving one pixel to the left or right, the bounding boxes sometimes change dramatically in size and number or disappear completely. The AI-player mitigates against this by remembering where Mario was last seen, so that if Mario is not detected in a frame, an educated guess about the correct location can be made.

## 3.4 Object detection using pixel comparison

Pipes in Super Mario Brothers have a consistent pattern to them across the width of the main pipe section. figure 3.5 shows the colour of the individual pixels, and a pipe section can be described as a sequence of the type “black pixels, followed by light green pixels, followed by dark green pixels, and so on”.

Blocks that stop Mario from moving forward are also susceptible to the same issues as pipes when using TF detection models, particularly as they often appear as chunks, as shown in figure 3.6.

Figure 3.5 - Pipe section with distinct colour patterns clearly shown.



Figure 3.6 - Blocks are often clumped together, which causes issues for the TF object

detection models.

The player process identifies pipes and blocks by scanning the horizontal line of pixels corresponding to Mario’s vertical location. The scanning is a simple for-loop, which looks at each pixel in turn to determine if it is part of a type of object. This algorithm is implemented in find\_horizontal\_objs(), available in Appendix C.12.

The process of comparing absolute pixel values against well-defined patterns works for NES games as the game-play screen can be thought of as a binary array of well defined, discrete values. That is, the objects the player process is attempting to detect always look the same along the X and Y axes, in terms of orientation, colour, and size. A similar for-loop used on images that are not as well defined, for example a photograph of an object, would be very error prone, as the physical size, lighting conditions, perspective, and angular orientation would vary between images. Because of this variability, a simple and accurate list of pixel values could not be constructed.

The code in find\_horizontal\_objs() is driven by a dictionary which records the order of colours as they appear in objects, for example the ‘pipe’ object is defined by the array of red, green and blue pixel values in dumb\_detection[‘pipe’][‘colseq’] (Figure 3.7).

dumb\_detection = \

{'pipe': {

'colseq': [[184, 248, 24], [0, 168, 0],

[184, 248, 24], [0, 168, 0],

[184,248,24], [0 ,168, 0],

[184, 248, 24]],

'width' : 29

},

'obstacle': {

'colseq' : [[240, 208, 176], [228, 92, 16],

[0, 0, 0]],

'width': 14

}

}

Figure 3.7 - Colour changes for simple objects are defined in the dumb\_detection dictionary.

Should more objects need to be detected in this manner, it would be easy to modify the code in figure 3.7 to add additional objects.

## 3.5 Object detection of numbers and words

Super Mario Brothers has a countdown timer in the top right corner of the screen. Once the timer reaches zero, the player loses a life and has to restart the level again. The timer is always displayed in the same location, using the same spacing between digits, and the digits are also well defined. That is, a particular digit is always the same shape, no matter where on screen it is displayed.

It is potentially beneficial for an AI model to know how many seconds of game-play time is left, so that it can start running, or taking riskier decisions in order to save time. As the numbers are always displayed with exactly the same pixels, the detection model employed for numbers by this project is a simple bitmap comparison of known numbers in the top right corner of the screen, using a nested for-loop in the check\_number() function (Appendix C.8). Whilst a nested for-loop sounds computationally heavy, in practice it is possible to exclude all but the correct number after comparing a small number of pixel lines.

The same approach is used to identify interesting words on the screen in the check\_black\_screen\_text() function (Appendix C.8). For example, the AI-player must be able to tell when Mario has died in the game, so that it can exit and report the number of pixels Mario moved to the right. The interesting words are displayed on a completely black background as shown in figure 3.8. The algorithm for finding the words of interest is to first sample 4 locations on the screen. If the locations are all black, see if the expected bitmap pattern exists in the expected location. The trigger word the AI-player looks for is ‘WORLD’ (figure 3.8), and once the trigger word has been seen, the AI-player process exits.

Figure 3.8 - The word ‘WORLD’ displayed on a black background at the start of the game, and also when Mario loses a life.



## 3.6 Creating a dataset for object detection

The TF object detection model needs significant amounts of training and test data in order to successfully train the neural network to identify objects. The training and test data for this project was generated automatically using a small supplementary Python script, generate\_training\_data.py (Appendix G), developed as part of the project.

The script generates one piece of training data by inserting an object onto a background image, thus creating an image similar to what the TF model will eventually operate on. The generated image is written to disk in JPG format, along with an XML file describing in what location the training object has been inserted. The XML file defines the bounding box, which is required by the training and testing process. The bounding box and object information for a piece of training data is known as the ‘ground truth’, and it defines what the model should predict for a piece of training data.

The backgrounds and objects were manually isolated, using graphics processing software, from screenshots produced by playing the game in human-player mode.

A secondary Bash shell command script, create-training-set-object-detection.sh (Appendix H) was developed to run the Python script multiple times, allowing a 4018-image training set to be generated in approximately 15 minutes.

Once a sufficiently large set of images had been produced, the data set was split into a training data set, and a testing data set. The training data set is used to train the TF model, and the testing data set is used for measuring the accuracy of the model on out of sample data. The Bash script split-training-test-data.sh (Appendix I) was created to split the data set into the two data sets, with 80% of the images being added to the training set, and the remainder to the test set.

## 3.7 Training the TensorFlow model to recognize objects

Once a custom training set has been generated, the TF model can be trained. TF comes with standard Python scripts for training a model on a custom dataset, but a simple Bash shell command script was developed as part of the project to invoke the TF Python scripts with the correct parameters. The shell command script is available in Appendix J.

TensorFlow produces statistics about model training progress, but the statistics are saved to a binary file. To view the data in a human-readable format, a piece of software called TensorBoard (TB) has been developed and bundled with the standard TensorFlow distribution. When running, TB analyses the TensorFlow statistics files and gives access to a wealth of information about the training performance, such Mean Average Precision (mAP) graphs.

mAP is a measure of how accurately the model detects objects in the images in the training set. For object detection, it measures how accurately a bounding box predicted by the model matches the ground truth in the training set. Figure 3.9 shows one of the mAP graphs produced by TB for the custom Mario dataset. The increase in mAP from the model learning the dataset is clearly visible.

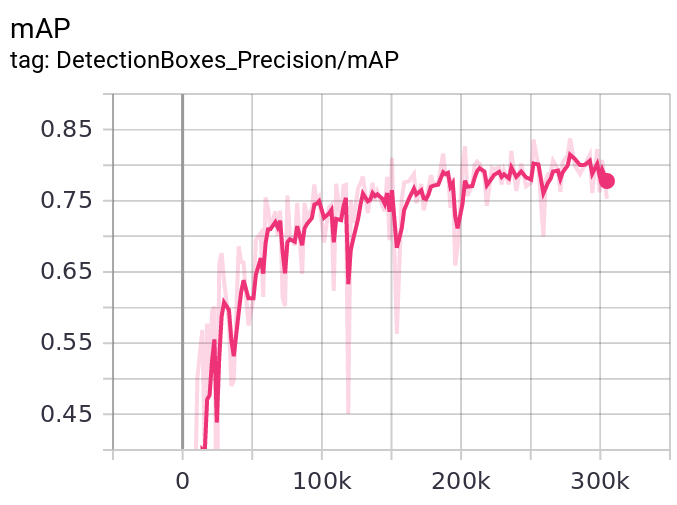


Figure 3.9 - The Mean Average Precision graph for the training data set. The X axis signifies the number of training “epochs” completed. An epoch in TensorFlow refers to one full pass through the data set.

The training process periodically evaluates the performance on the test data set. The TF training process samples images from the test data set, runs them through the detection process, and compares the output with the ground truth defined in the test data set. TB can then be used to display the difference between the detected objects and bounding boxes compared to the ground truth, as in figure 3.10.

The TF standard training scripts require a configuration file called the Pipeline configuration to be customized. The pipeline configuration used for this project is available in Appendix K.

If the game play images were standard photographs, objects in the photographs could reasonably be expected to appear in different orientations, be scaled in X or Y directions, be mirrored, or otherwise distorted. The TensorFlow object detection models take this into account when training, by randomly flipping, stretching or deforming the input images. When detecting objects in a data set where all the objects always appear in a well-behaved orientation, such as a game, this is superfluous, so the data augmentation options were turned off in the pipeline config file.



Figure 3.10 - Evaluation of a test set image. The detected objects and accuracy are displayed in the left-hand side, and the ground truth on the right hand side. For this particular test input, the model predicted the Goomba correctly (99% certainty).

## 3.8 Detecting how far Mario has run to the right using a reward function

In order to compare different game-play neural networks in the game-play training phase, it is necessary to know how “well” a particular neural net plays the game. As the AI-player is playing the game by interacting with an emulator, it cannot directly query game state, such as current score, or the position of Mario in the level, thus another measurement for progress must be devised.

A commonly used method for training neural nets via reinforcement learning (RL) is to use a function that measures how well a particular neural network solves a problem. This function is commonly referred to as a “reward function”. For Super Mario Brothers, an appropriate reward function is a measurement of how far to the right Mario has moved. As noted above, it is not possible to directly find this value, so a proxy must be devised.

As Mario moves to the right, the game screen keeps scrolling horizontally. A consequence of this is that the left-most column of pixels on the screen changes as progress in the level is made. The AI-player implements the check\_screen\_scroll()function (Appendix C.10), which observes the bottom 20 pixels in the left-most column of the screen, and every time the pixels change, a reward counter is increased.

The method is not without drawbacks. For example, when Mario runs at full speed towards the right, the screen in the emulator updates more often than the AI-player can sample (average sampling frequency is 9.7 frames per second), which means that the AI-player frequently misses updates.

A further drawback is that the pattern of the blocks is repeating (Figure 3.11), which means that there is a risk that two samples have the same pixel pattern even when Mario has moved to the right. However, even taking the issues into account, in actual gameplay and training, the counter shows a reasonably accurate relationship between distance covered by Mario and absolute value of the counter.



Figure 3.11 - Detail of the bottom left corner of the screen, where the repeating pattern of blocks is visible.

In testing of the game-play neural network, Mario quite often dies in exactly the same spot in the game by running into a Goomba, and the calculated reward for this spot ranged from 67 to 69 across all observed plays, which is accurate enough for training.

## 3.9 The gameplay neural network

The gameplay neural network (GPNN) is responsible for determining the action Mario should take at any given game-state. The state of the world is fed to the GPNN, and the GPNN calculates the desired action.

### 3.9.1 GPNN: Neurons into layers appropriate for the AI-player

An individual neuron in the GPNN is modelled using the small Neuron class (Appendix D.3), which is copied verbatim from (towardsdatascience.com, 2019). The Neuron class encapsulates a list of weights for its connected inputs, and a scalar bias value. It also provides an activate() method which returns the activation value, calculated using the sigmoid function (Appendix D.2)

In order to produce useful output a large set of neurons have to be connected. The GPNN framework developed as part of this project (Appendix D) handles fully connected ANNs, using the sigmoid activation function defined in the activate() function for all neurons. To facilitate testing of different network layouts, the neural net is defined in a Python dictionary, which is passed to the AI-player using a Java Simple Object Notation (JSON) formatted dictionary in a text file.

The neural net dictionary contains a key, which refers to a list of layer dictionaries. The layer dictionaries contain a 2-dimensional list of weights for each neuron in the layer and a list of biases. A sample network layer dictionary is shown in figure 3.12.

{ 'name': 'input',

'activation': 'sigmoid',

'num\_neurons': 8,

# as many weights as there are inputs, per neuron

'weights': [[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

],

# as many biases as there are neurons

'bias': [0, 0, 0, 0, 0, 0, 0, 0 ],

'neurons': []

},

Figure 3.12 - Sample input layer in the neural net definition. The layer consists of 8 neurons, which means that there will be 8 lists of weights. The individual weight lists are 10 elements long, as the network has 10 inputs. The ‘neurons’ key is currently empty, but upon initialization, it will be filled with Neuron objects.

Each neuron in the layer has its own list of weights, which are stored in ‘weights’ key. As the network is fully connected, the number of weights per neuron must match the number of neurons in the preceding layer, or the number of inputs if the layer is the first one. The reason for this is that a dot product is calculated between the weights and inputs for each neuron.

A strict limitation for a GPNN layout is that the network must have five output neurons, as the AI-player expects one out of five possible actions to be taken, however the GPNN framework can handle any number of hidden layers, with any number of neurons.

### 3.9.2 GPNN: Generating an action in the game

In the gameplay:run\_ann() function, the game state collected by the AI-player is passed to the GPNN in a Python array of 10 elements. The information includes the locations of the closest enemies, hole start and end positions, distance to nearest obstacle, and location of any pipes.

The game state is fed forward through the GPNN using the feed\_forward\_net() function (Appendix D.6), thus calculating activation values throughout the network.

Mario’s action is determined by the output layer, where each neuron corresponds to one of the actions “do nothing”, “move left”, “move right”, “move left and jump”, or “move right and jump”. The action chosen is the one corresponding to the neuron with the highest activation value.

## 3.10 Training the gameplay neural network

A traditional neural network is typically trained using a back-propagation mechanism, where neuron weights and biases are updated in a way that makes an input pattern, for a given network, generate an output pattern that more closely resembles the desired output value. That is, it is a supervised training method.

It would be possible to employ a similar training method for the GPNN, but it would require manually building up a big data set of input values (game state) to the neural network, with the corresponding desired output action. In its current form, the AI-player only reports the reward score at the end of the game, which makes a back-propagation method unfeasible. In its current form, the GPNN is a feed-forward only network, which does not support the back-propagation training method.

As described in 3.8, the AI-player process reports the number of pixels Mario has moved to the right upon completion of a game in AI-mode. A naive training method could be to simply generate a network with random weights and biases and play the game with the network. If the network has adequate performance, stop, otherwise generate a new network and try again.

There are two major drawbacks to this method – As the game is played in near real-time, the evaluation of a neural network takes a long time (up to 300 seconds), and there are many variables in even a small network. Combined, these drawbacks mean that finding a reasonable network using this naive method may take a very long time indeed.

The training method employed by this project builds on a simplified version of the ideas relating to genetic algorithms (GA) for neural network training, as described by Negnevitsky (Negnevitsky, 2002).

### 3.10.1 Negnetvitsky’s Genetic Algorithm for neural network training

In Negnevitsky’s algorithm, binary-encoded weights and biases of neural nets are encoded into a data structure, which Negnevitsky refers to as a “chromosome”.

Negnevitsky’s algorithm defines an initial set of “chromosomes” (neural networks). The chromosomes are all individually trained using backpropagation. The trained chromosomes have their fitness evaluated using a fitness function. A new set of chromosomes is generated by combining chromosomes and applying a mutation function and a cross-over function. The new set is now trained, evaluated, and so on.

The likelihood of a chromosome being chosen for combining is proportional to the fitness level, which should favour chromosomes that are fitter over those that are not, and over time the population of chromosomes should all become fitter as a result.

Negnevitsky’s cross-over function cuts the chromosomes into smaller sections, and the resulting sections are then combined into two new chromosomes. The mutation function randomly flips a small number of individual bits in the binary-encoded weight or bias values of newly created chromosome.

### 3.10.2 Simplified Genetic Algorithm

The algorithm used for GPNN training draws inspiration from the algorithm described by Negnevitsky, with some crucial simplifications. The algorithm can be summarised in the following steps:

1. Generate a set of base networks, with set size N. This set of networks is generation G0.

2. Play the game in AI-mode using each network in the current set of networks and record the output of the reward function.

3. Using the network with the highest reward from step 2 as a base, generate N networks by applying mutation only. This step is described in more detail in chapter 3.10.3.

4. Add the fittest network un-altered to the new set.

5. Label the newly generated set as generation G+1

6. Repeat from step 2, until cancelled by the user.

The main differences to Negnetvitsky’s algorithm are that no combining of networks is performed, and the individual networks are not trained. The method relies completely on a stochastic search for an optimal network, and the reward function filters out generated networks that are less fit than the current fittest network. Over time, the fitness of the best network in a set tends towards a maximum fitness point.

This algorithm was chosen as it is quick and easy to implement. In particular, it does not require back-propagation to work, as individual networks are not trained. The lack of backpropagation simplifies the implementation of the GPNN significantly, and it also negates the need for a large training set, as described in chapter 3.10.

A potential pitfall of this algorithm is that it may get stuck in local maxima, where each new generation of networks has no member that is better than the best network from the previous generation, and this behaviour was observed in initial testing. The maximum involved Mario running towards the right, and straight into the first enemy, and all the networks generated from this network did either the same, or stayed stationary until time ran out. Increasing the rate of mutation between generations solved this issue and allowed the training process to break out from the local maximum.

### 3.10.3 Generating new networks from a parent

Training the GPNN requires new networks to be generated from the fittest network in the previous generation. The generate\_network.py program is used to read a JSON formatted neural network definition, modify the network, and write the modified network to a file on disk.

The program accepts parameters to define the likelihood of a weight or bias being modified, and also the maximum modification allowed per weight or bias.

The current training values used are 40% mutation likelihood, and the maximum modification of weights and biases is 0.01. The 40% mutation likelihood is very large, and a large proportion of the generated networks perform significantly worse than the parent network, but it also means that there is a chance that the generated networks will break out of local maxima.

The actual modification of the network is performed in the mutate\_net() function (Appendix E) with a nested for loop, which loops across all the layers in the network, and across all the weights in each layer. A random number between 0 and 1 is generated, and if the value is smaller than the mutation likelihood, then a random value in [-max weight modification, +max weight modification] is added to the weight. Neuron biases are modified in the same fashion, except the random value is drawn from [-max bias modification, +max bias modification.

The full source-code for generate\_network.py is available in Appendix E.

### 3.10.4 Automating GPNN training

As training the player is a time-consuming task, a Bash script, train-player.sh (Appendix E), was developed to automate testing. It implements the algorithm described in chapter 3.10.2, using the generate\_network.py script described in chapter 3.10.3

Generated networks and fitness scores are recorded into text files in a directory structure, so that performance can later be evaluated, and the networks played again using the AI-player process in AI-mode.

As the evaluation of the networks in a generation is done in real-time, with one instance of a neural network at a time, the training phase takes a significant amount of time. To mitigate the impact of the risk of having to stop the training process mid-way before a suitable network has been found, the script was developed to be able to restart training from the latest generation of networks.

The automated testing script greatly speeds training up and makes it possible to train the model in an unattended fashion.

### 3.10.5 GPNN: Layout evaluation and playing performance

Two different network layouts were evaluated. The first layout consisted of 8 input layer neurons, no hidden layer, and the required 5 output neurons. The network was trained over 168 generations, and the best performance recorded by the reward function was 151.

The second layout consisted of 8 input layer neurons, one hidden layer of 6 neurons, and the required 5 output neurons. The best performance from only 32 generations is 231, and a human completing the whole first level will record a score of around 350.

One possible cause for the discrepancy in reward function outputs is that the first network is not flexible enough to learn enough about the game-state patterns and corresponding actions, where-as the second network can adapt to a better fit to the data, due to its hidden layer. Given that the training method currently evaluates performance in Level 1 only, it could be argued that the increased flexibility allows the network to overfit to Level 1. It is possible that a network that performs very well on Level 1 would perform worse than the simpler networks on other levels in the game.

A graphical representation of the second layout is shown in figure 3.13.

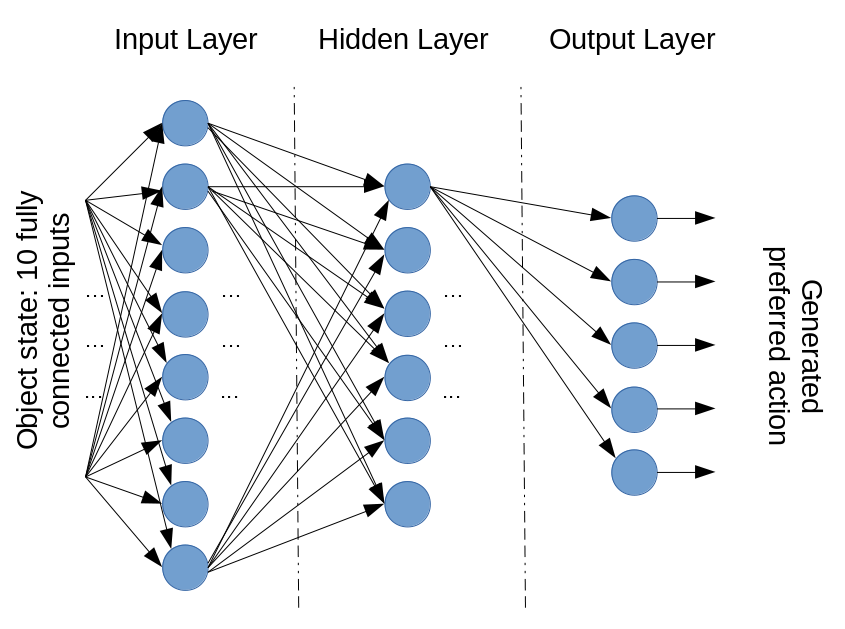


Figure 3.13 - Graphical representation of the data flow and connections between neurons in the more successful of the two neural net layouts tested. Each layer is fully connected, and the input eventually activates the 5 output neurons, where the actual action generated is the output neuron with the highest activation value.

### 3.10.6 The effect of timing jitter on training and GPNN performance

There is no synchronization between the emulator and AI-player, apart from the AI-player requesting screen updates from the emulator. Due to the lack of synchronization, the AI-player receives slightly different pixel-data from run to run.

The different pixel-data causes the TF object detection to take a varying amount of time, and as all the objects have to be detected by the AI-player before it can make a game-play decision using the GPNN, occasionally the player ends up running into enemies that a previous AI-mode run with the same GPNN cleared successfully. The jitter can be easily seen in Table 3.1, which shows the performance (pixels moved to the right) of consecutive runs of the AI-player in AI-mode, using the same GPNN.

Table 3.1: Table showing game-play performance in terms of pixels moved to the right by Mario. The same GPNN was used for all runs, and the GPNN performance jitter is visible in the right column.

|  |  |
| --- | --- |
| **Run #** | **Pixels to right** |
| 1 | 58 |
| 2 | 60 |
| 3 | 79 |
| 4 | 108 |
| 5 | 102 |
| 6 | 106 |
| 7 | 104 |
| 8 | 105 |
| 9 | 105 |
| 10 | 107 |

The jitter makes it hard to know if a network is genuinely good or bad, or if it was simply “lucky”/ ”unlucky” for a particular run. Mitigations are suggested in chapters 5.2 and 5.4.

# 4. Evaluation of project

The stated goal of the project is to create a stand-alone AI program, capable of acting as a player of Super Mario Brothers. The project comprises a number of software components, most of which are completely original, and others which have been modified from other Open Source projects.

## 4.1 Object recognition

Object recognition has been successfully implemented using both TensorFlow models, and also by simple analysis of pixel data. The TF object detection models are surprisingly inaccurate, requiring the AI-player to remember where Mario has been detected last, in case the TF object detection model cannot detect him. Another surprising weakness is that blocks with a question mark are detected as Koopa-Troopa enemies, even though to a human, they appear very dissimilar, as seen in figure 3.4.

Pixel based object detection works very well in the AI-player. This is due to the exact nature of pixels in a game of this style. Every object has a very well-defined layout, and the edges are completely crisp. If there were even a small amount of variation in pixel colour values, this method would not work at all, as it exactly compares the colour values of the pixels found on the screen with the colour values defined in the pattern being searched for.

## 4.2 Gameplay neural network performance

The game-play decision making neural networks as expected. As it is a feed-forward network only, training using the stochastic method implemented is reasonable. Unfortunately training takes a long time as the game is played in real-time even for training.

In many instances the new networks generated by generate\_network.py (Appendix E) are worse than the original network they were generated from. The worse-performing networks are filtered out by the train-player.sh process, but it would possibly be better to generate a better network in the first place, rather than relying on pure chance to improve an existing network.

The GPNN may be too shallow to allow it to capture all the intricacies of playing Mario successfully. The performance of the network improved markedly between no hidden layers, and one hidden layer, and it is possible that further hidden layers would improve the performance further. Unfortunately, due to the long time spent in training, training a bigger network is a big task, which would require capabilities of parallel training in the cloud.

The GPNN should have inputs describing historical state. At present, the state of the world is a static snapshot of all objects, and there is no information about object movements or velocity passed to the network, which means that the network cannot differentiate between an enemy moving away from Mario or towards him, and at what speed. If some historical state was passed in, perhaps the network could learn to differentiate between direction of travel for objects. This would be similar to DeepMind’s solution, which feeds four game frames to their neural networks for playing Atari games (Mnih, et al., 2013).

## 4.3 Emulator with API capabilities

The modified emulator works very well. The added network capabilities with the API make it possible to run a particular game over-and-over without human intervention, which would not have been possible before the modifications to the emulator. A useful addition would be an emulator mode which stops the emulator from drawing anything to screen, as described in chapter 5.1. This would allow the emulator to run on a system without a graphical display, which would be an advantage for training in a cloud-type setting, where a display is often not available.

A further useful extension to the emulator would be a function allowing the AI-player to step through a game frame by frame, rather than playing the game in real-time. This would eliminate the jitter described in chapter 3.10.6, which would give make the whole system deterministic, albeit not work for real-time game-playing.

# 5. Potential improvements

## 5.1 Headless mode

It would be beneficial for training purposes if the whole system could operate without a display, colloquially known as “headless mode”. Currently this is not possible, as the SDL2 library (libsdl.org, 2019), which the emulator uses for graphical operations, requires a valid screen device. The AI-player process also requires a screen device, as it uses the PyGame library (pygame.org, 2019).

If the system could run in headless mode, one could package both the emulator and AI-player pieces into, for example, a Docker software container (Docker Inc., 2019), which could then be scheduled in a public cloud for training, with the results periodically gathered either manually or by means of a simple shell script.

Although completely orthogonal to this project, a headless emulator could also be used as a backend for a web-service offering classic Nintendo games playable in a browser, particularly if the screen data were also compressed, as described in chapter 3.1.3.

## 5.2 Frame synchronization between emulator and AI-player

To reduce the training jitter and improve AI game-play accuracy, some form of frame synchronization should be implemented. The emulator could have a new stepping mode added to it, such that it pauses, for example, every time it redraws its internal screen buffer. Once the AI-player requested a screen update, the emulator would send the screen contents, and then run again until it redrew its screen buffer.

This would make the games and AI-player actions deterministic, which should result in better training performance. It would also result in games running exactly the same on different hardware with different performance profiles.

## 5.3 GPNN training – learning from a human

Currently the GPNN is trained in a stochastic fashion, without the use of backpropagation, and training is slow and inconsistent. A human player is much faster at learning how to play the game than the current GPNN, and it would be a very interesting project indeed to implement a form of learning mechanism that learns from a human player.

As the AI-player already recognises objects in a game-frame, one could imagine a setup where a human plays the game in interactive mode, and the player process records the state of the world, and also the action the human player takes as a response to that state.

The recorded information could then be used for training of the GPNN. As the dataset would have both the state of the world and the desired action, backpropagation learning methods could be used. This would necessitate an extension to the GPNN framework, or it could simply be switched out for a TensorFlow network.

A further benefit would be that the training would not have to be done in real-time, which would allow a GPU to be used. This should greatly speed training up.

## 5.4 GPNN performance – reducing jitter

The GPNN performance is currently jittery. The same network can sometimes produce wildly different results, as shown in Table 3.1. Chapter 5.2 suggests frame synchronization to make the game and AI-player behave deterministically, but at the cost of real-time performance.

A different approach to reducing jitter in the training phase would be to evaluate each network in each generation several times and average the results. If a GPNN is genuinely better than the other GPNNs in its generation, it is likely to produce a higher average score than the worse networks.

# 6. Issues encountered

During the project, several small issues were encountered. A subset of these issues is documented below.

## 6.1 Numpy matrix cell addressing

TensorFlow uses the Python library Numpy to perform mathematical operations faster than the standard Python functions allow. The cells in the Numpy matrices used to represent images in the TensorFlow object detection models are indexed by [row][column] (SciPy, 2019). When the AI-player reads screen contents from the emulator, the contents are in a [column][row] format, as that is what is common in graphics programming. To compound the confusion, the AI-player represents the screen contents in a Numpy array.

Getting the required matrix transformations correct, so that TF would train correctly, and the AI-player display the contents to screen correctly proved tricky. Significant time was spent trying different transformations, until a correct solution was found.

## 6.2 TensorFlow object detection training speed

The TensorFlow object detection models are incredibly slow to train. Attempts to train the models using a laptop computer produced terrible results, at about 50% object detection accuracy, even after a week of training.

To combat the long training time, the number of different objects to learn was shrunk from 10 to 5. Unfortunately, this did not improve the speed of training, and in the end, a GPU powered server had to be used for the training of the TF model.

## 6.3 The reward function

The reward function is a critical piece of the GPNN training algorithm. Three different reward functions were tested:

1. Measure the amount of time Mario stayed alive.

2. Measure how many times the GPNN generated an action that should result in Mario moving to the right. The two applicable actions are “run right”, and “run and jump right”.

3. Measure pixel changes at either the left or the right of the screen, as described in chapter 3.8.

The first reward function resulted in Mario always staying put, or running to the left, as moving to the right results in Mario’s death at some point. As Mario usually died by the first enemy, all the neural GPNNs converged towards a model that stayed put for the entire duration of the game.

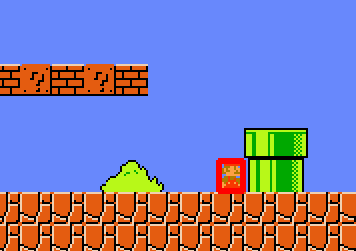
The second reward function performs better, but it rewards behaviour that results in the GPNN issuing a command to move right, even if Mario is stuck next to an object, as shown in figure 6.1.

Figure 6.1 - Mario is stuck next to an object, but as the incorrect reward function counts every action to move right, this state achieves high-score.

The third reward function measures changes in the pixel values in the left-most column of pixels. This appears to be a fair reward function, as it addresses the issue of the second reward function and awards no points for staying still.

After the initial implementation of the third reward function was complete, a curious behaviour was observed; Mario would run to the left edge of the screen and keep running. As the animation of Mario causes the left-most pixel column to change, this resulted in a very high score. The bug was addressed by observing only a small number of pixels in the bottom left corner of the screen.

Once the third reward function had been implemented, the GPNN training resulted in networks that cause Mario to move to the right, and it quickly learnt to jump when in the vicinity of enemies.

# 7. Final conclusions

This project has successfully implemented an artificially intelligent player for a classic arcade game in a short timeframe. The project separates the identification of objects from the game-play decisions, and therefore avoids having to use very deep neural networks, which can both identify objects and generate decisions.

The project utilises a range of Open Source tools and projects, which is typical of many software projects being developed today. The project goals would have been significantly harder to achieve, perhaps impossible, in the timeframe given, had a large number of talented individuals and companies not donated their source code to various Open Source projects, often for no financial reward. It could be argued, in many ways, that this project truly rests on the shoulders of giants. In the hope that somebody else will find the source code useful or interesting, in due course the source code and report for the project will be released on the GitHub web site at https://github.com/hbilar/.

A potentially interesting area of research would be a GPNN that could learn from observing a human player, with a method like that described in chapter 5.3.

# References

cocodataset.org, 2019. *COCO - Common Objects in Context.* [Online]   
Available at: http://cocodataset.org/#overview  
[Accessed 23 08 2019].

Docker Inc., 2019. *Docker : What is a Container.* [Online]   
Available at: https://www.docker.com/resources/what-container  
[Accessed 10 09 2019].

Google / TensorFlow, 2019. *TensorFlow Object Detection Zoo.* [Online]   
Available at: https://github.com/tensorflow/models/blob/master/research/object\_detection/g3doc/detection\_model\_zoo.md  
[Accessed 25 03 2019].

Google, 2019. *Tensorflow project on GitHub.* [Online]   
Available at: https://github.com/tensorflow/models/tree/master/research/object\_detection  
[Accessed 25 03 2019].

libsdl.org, 2019. *SDL : Simple Directmedia Layer.* [Online]   
Available at: https://www.libsdl.org/index.php  
[Accessed 25 08 2019].

machinelearningmastery.com, 2019. *Gentle Introduction to Transfer Learning for Deep learning.* [Online]   
Available at: https://machinelearningmastery.com/transfer-learning-for-deep-learning/  
[Accessed 03 09 2019].

Mnih, V. et al., 2013. *arXiv.* [Online]   
Available at: https://arxiv.org/pdf/1312.5602v1.pdf  
[Accessed 13 09 2019].

Negnevitsky, M., 2002. In: *Artificial Intelligence : A Guide to Intelligent Systems.* Harlow: Pearson Education Limited, pp. 286-289.

Orru, A., 2017. *LaiNES compatibility chart.* [Online]   
Available at: https://github.com/AndreaOrru/LaiNES#compatibility  
[Accessed 13 09 2019].

Orru, A., 2017. *LaiNES emulator.* [Online]   
Available at: https://github.com/AndreaOrru/LaiNES  
[Accessed 25 03 2019].

pygame.org, 2019. *PyGame : About.* [Online]   
Available at: https://www.pygame.org/wiki/about  
[Accessed 14 09 2019].

SciPy, 2019. *Numpy documentation : numpy.matrix.item.* [Online]   
Available at: https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.item.html  
[Accessed 03 09 2019].

towardsdatascience.com, 2019. *Machine Learning for Beginners: An Introduction to Neural Networks.* [Online]   
Available at: https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9  
[Accessed 23 08 2019].

Wikipedia, 2019. *Run-length encoding.* [Online]   
Available at: https://en.wikipedia.org/wiki/Run-length\_encoding  
[Accessed 13 09 2019].

Wikipedia, 2019. *Wikipedia : Nintendo Entertainment System.* [Online]   
Available at: https://en.wikipedia.org/wiki/Nintendo\_Entertainment\_System  
[Accessed 01 03 2019].

# Appendix A: Instructions for using the software

### Appendix A.1: Software Pre-requirements

* You must be logged on to a LINUX desktop (with a graphical user interface)
* You must have the USB drive mounted somewhere accessible. For example, on some Linux distributions, the USB drive will be available in /media after insertion. If it is not available, the USB stick has to be "mounted" onto some directory.
* The path to the directory where the USB stick is visible must be set in the environment variable PATH\_TO\_USB (See instructions below).
* There must be about 2Gb of space available in the directory where you intend on running the software from. Note, the destination directory for the instructions below can be changed by setting the MARIO\_HOME environment variable toa different value.

### Appendix A.2: Installation of AI PLAYER software

1. Open up a command line terminal.

2. In the terminal, enter the following commands:

# Change MARIO\_HOME to a different path if there is less than

# about 2Gb free space in your home directory.

export MARIO\_HOME=~/henrik-bilar-project

# NOTE: THIS NEEDS TO BE CHANGED TO THE ACTUAL PATH WHERE

# THE USB DRIVE IS MOUNTED

export PATH\_TO\_USB=/mnt/usb

# create directory and CD into it.

mkdir -p $MARIO\_HOME

cd $MARIO\_HOME

# Untar the ai-player.tar.gz file into a specific directory

tar xvf $PATH\_TO\_USB/ai-player.tar.gz

# enter the ai-player directory

cd ai-player

# Create Python 3 virtualenv

virtualenv -p python3 testenv

source testenv/bin/activate

# Install the python libraries

./install-required-python-libraries.sh

Appendix A.3: Installation of the NES Emulator

1. Open up a NEW terminal window (do not use the same one

as for the AI Player).

2. In the new terminal, enter the following commands:

# NOTE: Set MARIO\_HOME to the same value as when you

# installed the AI player software in a previous step

export MARIO\_HOME=~/henrik-bilar-project

# NOTE: THIS NEEDS TO BE CHANGED TO THE ACTUAL PATH WHERE

# THE USB DRIVE IS MOUNTED

export PATH\_TO\_USB=/mnt/usb

# CD into the MARIO\_HOME directory

cd $MARIO\_HOME

# Extract the NES emulator

tar zxvf $PATH\_TO\_USB/nes-emulator.tar.gz

### Appendix A.4: Running the NES Emulator

If you have restarted your terminal window since installation,

run the following command:

cd THE\_DIRECTORY\_WHERE\_YOU\_INSTALLED\_THE\_SOFTWARE

Note, the default directory for installation is ~/henrik-bilar-project . The tilde (~) sign is important, so type that too if you went for the default location.

To start the emulator in loop mode, run the following command:

./nes-emulator/nes-emulator-on-loop.sh

You should see output like:

Running nes emulator in loop mode

Setting up a tcp socket on port 8005

Trying to bind to port 8005

Waiting for client to connect on port 8005

### Appendix A.5: Running the player

First of all, make sure the NES emulator is up and running.

If you have restarted your terminal window since installation,

run these three commands:

cd THE\_DIRECTORY\_WHERE\_YOU\_INSTALLED\_THE\_SOFTWARE

cd ai-player

source testenv/bin/activate

Note, the default directory for installation is ~/henrik-bilar-project. The tilde (~) sign is important, so type that too if you went for the default location.

Running in human player mode:

python3 player.py

In human player mode, you have to control Mario. Make sure

the player.py window is selected and use the arrow keys

to move left and right.

The enter key is mapped to the start button on the joy pad

and you can use 'a' to jump, and 'b' to run (press 'b' together

with either left or right).

Running in AI-mode with a network:

python3 player.py oneshot gameplay\_neural\_nets/13/3.nn

You can test other networks by substituting the path.

You will find different generations of networks in the

gameplay\_neural\_nets directory (sub directories with

numbers). Each sub directory has a file called 'scores'

which lists the network name, and what the reward function

returned for the network when played on my test setup.

Due to the jitter mentioned in the project report, the

performance may vary significantly between runs, but

also depending on how heavily loaded the computer you

are running on is.

# Appendix B: LaiNES source code modifications

Appendix B, and sub-sections to B, contain the modifications to the LaiNES emulator performed as part of this project.

Yellow highlighted lines of code are from the original LaiNES source code available at <https://github.com/AndreaOrru/LaiNES>, and they were not written as part of this project.

In general, if a line has been modified to take into account a new function signature, then the line is not shown in the code-listing appendices here but are visible in the full “diff” output, which is available on the supplied USB drive.

The Appendix code listings are not complete listings of the full LaiNES system, and only shows functions that have been modified as part of the project. For a full source code listing, see the supplied USB drive.

## Appendix B.1 LaiNES main.cpp

The modifications to main.cpp are:

- enable command line parameter parsing

- usage information in case incorrect parameters are passed

- enable listening to a TCP socket if applicable.

### Appendix B.1.1 general changes to main.cpp

#include "gui.hpp"

#include "config.hpp"

#include "remote\_client.hpp"

#include <libgen.h>

#include <sys/socket.h>

#include <fcntl.h>

#include <netinet/in.h>

#include <unistd.h>

/\* the TCP controller port \*/

int controller\_fd = -1;

void die\_with\_error\_message(char \*s)

{

printf("\n\n\nFATAL ERROR: %s\n\n\n", s);

exit(1);

}

### Appendix B.1.2 Set up TCP listening socket

/\* Create a socket, listening to it for a TCP connections, and then

\_\_wait for a connection\_\_ to it.

Returns the FD of the CLIENT tcp connection! \*/

int wait\_for\_client\_connection(int port)

{

/\* socket to listen to \*/

int socket\_fd;

if ((socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) <= 0) {

DIE("Could not create socket");

}

/\* set up the sockaddr\_in struct properly \*/

struct sockaddr\_in ipv4\_addr;

ipv4\_addr.sin\_family = AF\_INET; // specify the IPV4 protocol family

ipv4\_addr.sin\_addr.s\_addr = INADDR\_ANY; // listen to any available interface

/\* If the bind fails, keep increasing until we find a port we can bind on \*/

bool done = false;

while (! done) {

ipv4\_addr.sin\_port = htons(port);

if (port > 9000) {

printf("Couldn't bind any ports up to %d\n", port);

DIE("failed to bind");

}

printf("Trying to bind to port %d\n", port);

/\* bind to the socket, and then start listening for connections \*/

if (bind(socket\_fd, (struct sockaddr \*)&ipv4\_addr, sizeof(ipv4\_addr)) < 0) {

//DIE("Failed to bind()");

printf("Failed to bind() - trying next port!");

port ++;

continue;

}

if (listen(socket\_fd, 3) < 0) {

//DIE("Failed to listen()");

printf("Failed to listen() - trying next port!");

port ++;

continue;

}

done = true;

}

socklen\_t addrlen = sizeof(ipv4\_addr);

/\* Now, wait for a connection before returning \*/

int client\_fd;

printf("Waiting for client to connect on port %d\n", port);

if ((client\_fd = accept(socket\_fd, (struct sockaddr \*) &ipv4\_addr,

(socklen\_t\*) &addrlen)) < 0) {

DIE("Failed to accept() connection!");

}

/\* Make the socket non blocking, so that when we come to read the state

in the get\_joypad\_state function, we don't block \*/

int cur\_flags = fcntl(client\_fd, F\_GETFL, nullptr);

if (fcntl(client\_fd, F\_SETFL, cur\_flags | O\_NONBLOCK) < 0) {

DIE("Failed to make socket non\_blocking");

}

/\* Close the socket we're listening to \*/

close(socket\_fd);

return client\_fd;

}

### Appendix B.1.3 Command line parameter handling

/\* print usage\_and\_die information and exit the program with exit code 1 \*/

void usage\_and\_die(char \*progname, char \*errmsg)

{

printf("\n\n\n");

if (errmsg != nullptr){

printf("ERROR: %s\n\n\n", errmsg);

}

printf("Usage: %s [optional flags] <rom>\n\n", progname);

printf("Flags:\n");

printf(" -help | --help display this help\n");

printf(" -screensize:<n> screen size (1 <= n <= 4)");

printf(" -tcp:<xx> listen to port xx for controller instructions");

printf("\n\n\n");

exit(1);

}

int main(int argc, char \*argv[])

{

char \*the\_rom = nullptr;

int screen\_size = 1;

// figure out program name (and take base name of the path passed in)

char \*progname = basename(argv[0]);

/\* check to see if user passed any command line params \*/

char \*\*p = argv;

for (int p = 1; p < argc; p++) {

char \*cur\_p = (char\*)argv[p];

if (cur\_p[0] == '-') {

// this is a flag

/\* if the parameters is of the type -paramname:value, find value \*/

char \*value = nullptr;

value = strchr(cur\_p, ':');

if (value)

value++;

if (streq(cur\_p, "--help") || streq(cur\_p, "-help")) {

usage\_and\_die(progname, nullptr);

}

else if (strstarts("-screensize:", cur\_p)) {

/\* code \*/

printf("Setting screen size!\n");

screen\_size = atoi((char \*)value);

}

else if (strstarts("-tcp:", cur\_p)) {

/\* Set up a tcp socket for remote control \*/

int p = atoi((char \*) value);

printf("Setting up a tcp socket on port %d\n", p);

controller\_fd = wait\_for\_client\_connection(p);

}

} else {

// assume rom path

the\_rom = argv[p];

}

}

if (the\_rom == nullptr) {

usage\_and\_die(progname, (char\*)"specify rom on command line");

}

GUI::load\_settings(screen\_size);

GUI::init(the\_rom);

GUI::run();

return 0;

}

## Appendix B.2 LaiNES config.cpp

The config.cpp file in the original source code deals with reading and writing LaiNES configuration files. The modifications to config.cpp are:

* Removed the functions get\_config\_path() and save\_settings(). These functions dealt with reading and writing configuration files, and the functionality is no longer needed.
* Modified the load\_settings() function to hardcode the keyboard scan codes corresponding to joypad buttons.

/\* Load settings \*/

void load\_settings(int screen\_size)

{

set\_size(screen\_size);

/\* Control settings \*/

for (int p = 0; p < 1; p++)

{

const char\* section = p==0?"controls p1":"controls p2";

KEY\_UP[p] = (SDL\_Scancode)82;

KEY\_DOWN[p] = (SDL\_Scancode)81;

KEY\_LEFT[p] = (SDL\_Scancode)80;

KEY\_RIGHT[p] = (SDL\_Scancode)79;

KEY\_A[p] = (SDL\_Scancode)4;

KEY\_B[p] = (SDL\_Scancode)22;

KEY\_SELECT[p] = (SDL\_Scancode)44;

KEY\_START[p] = (SDL\_Scancode)40;

}

}

## Appendix B.3 LaiNES gui.cpp

The gui.cpp file contains functions related to drawing on screen and getting input from a user. The modifications to gui.cpp are:

* The TCP API.
* Get\_joypad\_state() modified to also take into account key presses received from API.
* The init() function immediately loads a game ROM, without calling any menu functions.

### Appendix B.3.1 general changes to gui.cpp

/\* Initialize GUI \*/

void init(char \*rom\_path)

{

// Initialize graphics system:

SDL\_Init(SDL\_INIT\_VIDEO);

SDL\_SetHint(SDL\_HINT\_RENDER\_SCALE\_QUALITY, "linear");

TTF\_Init();

APU::init();

// Initialize graphics structures:

window = SDL\_CreateWindow ("LaiNES",

SDL\_WINDOWPOS\_CENTERED, SDL\_WINDOWPOS\_CENTERED,

WIDTH\*last\_window\_size,HEIGHT\*last\_window\_size, 0);

renderer = SDL\_CreateRenderer(window, -1,

SDL\_RENDERER\_ACCELERATED | SDL\_RENDERER\_PRESENTVSYNC);

SDL\_RenderSetLogicalSize(renderer, WIDTH, HEIGHT);

gameTexture = SDL\_CreateTexture (renderer,

SDL\_PIXELFORMAT\_ARGB8888, SDL\_TEXTUREACCESS\_STREAMING,

WIDTH, HEIGHT);

int w, h;

Uint32 pix\_format;

SDL\_QueryTexture(gameTexture, &pix\_format, nullptr, &w, &h);

pixel\_format.format = pix\_format;

font = TTF\_OpenFont("res/font.ttf", FONT\_SZ);

keys = SDL\_GetKeyboardState(0);

// Initial background:

SDL\_Surface\* backSurface = IMG\_Load("res/init.png");

background = SDL\_CreateTextureFromSurface(renderer, backSurface);

SDL\_SetTextureColorMod(background, 60, 60, 60);

SDL\_FreeSurface(backSurface);

/\* henrik \*/

Cartridge::load(rom\_path);

toggle\_pause();

return;

}

bool display\_screen\_to\_client = true;

static u8 remote\_joypad\_state[2] = {0, 0}; // initial state of the remote joypad

### Appendix B.3.2 Functions to send data to the player

void send\_message\_to\_remote(char \*msg)

{

char \*start\_msg = (char \*)"message:";

char \*end\_msg = (char \*)"\n";

write(controller\_fd, start\_msg, strlen(start\_msg));

write(controller\_fd, msg, strlen(msg));

write(controller\_fd, end\_msg, strlen(end\_msg));

}

/\* Send the contents of the screen to the remote.

Sends: 256x240 pixels from screen[], encoded as a list of

r,g,b,r2,g2,b2 ... \*/

void send\_screen\_to\_remote()

{

/\* Dump the 'pixels' over TCP to the client \*/

char \*header = (char \*)"display:\n";

write(controller\_fd, header, strlen(header));

char bigbuf[WIDTH \* HEIGHT \* 3 \* 5] = ""; /\* 3 rgb values, with max 4 chars per colour \*/

char \*bigbuf\_ptr = bigbuf;

/\*\*\*\* USE BIGBUFPTR TO AVOID CALLING WRITE TOO OFTEN \*\*\*\*/

printf("SENDING SCREEN\n");

for (int r = 0; r < HEIGHT; r++) {

int offset = r \* WIDTH;

char linebuf[10000] = "";

linebuf[0] = '\0';

char \*lineptr = linebuf;

// printf(" line number %d:\n", r);

for (int c = 0; c < WIDTH; c++) {

Uint8 r, g, b;

Uint32 pixel = PPU::pixels[offset + c];

r = (0x00ff0000 & pixel) >> 16;

g = (0x0000ff00 & pixel) >> 8;

b = (0x000000ff & pixel);

char minibuf[100];

snprintf(minibuf, sizeof(minibuf) - 1, "%hhu,%hhu,%hhu,", r, g, b);

//strcat(linebuf, minibuf);

char \*p = minibuf;

while (\*p) {

\*lineptr = \*p;

lineptr++;

p++;

}

}

//strcat(linebuf, "\n");

\*(lineptr++) = '\n';

\*(lineptr++) = '\0';

write(controller\_fd, linebuf, strlen(linebuf));

}

send\_message\_to\_remote((char \*)"Done");

printf("bigbuf = >%s<", bigbuf);

}

/\* Send the contents of the screen to the remote in binary.

Sends: 256x240 pixels from screen[], in binary format \*/

void send\_binary\_screen\_to\_remote()

{

/\* Dump the 'pixels' over TCP to the client \*/

Uint8 buf[WIDTH \* HEIGHT \* 3];

Uint8 \*ptr = buf;

for (int c = 0; c < HEIGHT \* WIDTH; c++) {

Uint8 r, g, b;

Uint32 pixel = PPU::pixels[c];

r = (0x00ff0000 & pixel) >> 16;

g = (0x0000ff00 & pixel) >> 8;

b = (0x000000ff & pixel);

\*(ptr++) = r;

\*(ptr++) = g;

\*(ptr++) = b;

}

write(controller\_fd, buf, sizeof(buf));

}

### Apppendix B.3.3 Handling of remote commands

/\* Parse an actual command string that's come in \*/

void handle\_remote\_command(char \*cmd)

{

if (strstarts("j ", cmd)) {

/\* joypad value \*/

char \*p = cmd;

p += 2;

unsigned short pad\_value, pad;

int num\_read = sscanf(p, "%hu %hu", &pad\_value, &pad);

if (num\_read == 0) {

printf("Malformed input!");

send\_message\_to\_remote((char \*)"malformed input. Expected 'p <value> [pad (0 / 1]'");

return;

}

if (num\_read == 1) {

// user didn't specify pad, assume 0

pad = 0;

}

printf("cmd = >>%s<<, Setting remote joypad state to %d\n", cmd, pad\_value);

remote\_joypad\_state[pad] = pad\_value;

}

else if (strstarts("reset", cmd)) {

/\* reset \*/

send\_message\_to\_remote((char \*)"Resetting console!");

printf("Resetting console!\n");

CPU::power();

PPU::reset();

APU::reset();

}

else if (strstarts("screen", cmd)) {

/\* dump contents of the screen \*/

send\_screen\_to\_remote();

}

else if (strstarts("binscreen", cmd)) {

/\* dump contents of the screen in binary \*/

send\_binary\_screen\_to\_remote();

}

else if (strstarts("poweroff", cmd)) {

/\* kill ourselves \*/

printf("Powering off console\n");

exit(0);

}

}

char remote\_input\_buffer[1000] = "";

char \*remote\_buf\_input\_ptr = remote\_input\_buffer;

/\* handle the remote player \*/

void handle\_remote\_input()

{

/\* Read commands from tcp/ip \*/

/\* try to read from the tcp socket (non-blocking), to see if

there's more user input \*/

char tmp\_buf[1000];

int read\_bytes = 0;

bzero(tmp\_buf, 1000);

read\_bytes = read(controller\_fd, tmp\_buf, sizeof(tmp\_buf));

if (read\_bytes > 0) {

/\* append tmp\_buf to remote\_input buffer, and see if there is a

complete command to process \*/

strncat(remote\_input\_buffer, tmp\_buf, sizeof(remote\_input\_buffer) - 1);

char \*p = (char \*)(tmp\_buf + strlen(tmp\_buf) - 1);

bool found\_command;

do {

/\* see if there's a command \*/

char \*p = remote\_input\_buffer;

found\_command = false;

while (\*p && found\_command == false) {

if (\*p == '\n') {

found\_command = true;

} else {

p++;

}

}

if (found\_command) {

/\* command exists between beginning of remote\_input\_buffer and p \*/

char cmd[1000];

strncpy(cmd, remote\_input\_buffer, p - remote\_input\_buffer);

cmd[p - remote\_input\_buffer] = '\0';

//printf("REMOTE COMMAND: >>>%s<<<\n", cmd);

handle\_remote\_command(cmd);

/\* copy contents from p until '\0' is encounted to a buffer, then

copy the buffer contents to remote\_input\_buffer \*/

if (\*p == '\n')

p++; // we need to get rid of the \n in \*p if it's there

memcpy(remote\_input\_buffer, p, sizeof(remote\_input\_buffer) - 1);

}

} while (found\_command);

}

}

### Appendix B.3.4 Get state of the remote joypad

The get\_joypad\_state() function is used to calculate the bit-pattern that NES software uses to represent joypad state. In its original form, the bit-pattern is calculated by checking what keyboard keys are pressed. The modifications cause the joypad bit-pattern set by the remote AI-player, via the TCP API, to be taken into account.

u8 get\_joypad\_state\_from\_tcp(int n)

{

return remote\_joypad\_state[n];

}

/\* Get the joypad state from SDL \*/

u8 get\_joypad\_state(int n)

{

static int been\_here\_before = 0;

u8 j = 0;

j |= (keys[KEY\_A[n]]) << 0;

j |= (keys[KEY\_B[n]]) << 1;

j |= (keys[KEY\_SELECT[n]]) << 2;

j |= (keys[KEY\_START[n]]) << 3;

j |= (keys[KEY\_UP[n]]) << 4;

j |= (keys[KEY\_DOWN[n]]) << 5;

j |= (keys[KEY\_LEFT[n]]) << 6;

j |= (keys[KEY\_RIGHT[n]]) << 7;

u8 keyboard\_state = j;

if (controller\_fd > 0) {

/\* tcp connection has been set up \*/

/\* note, we OR this here, so that we can still use the

keyboard to play with when testing... \*/

j |= get\_joypad\_state\_from\_tcp(n);

// printf("Remote state after tcp: %d\n", j);

}

// printf("keyboard\_state: %d, get\_joypad\_state: %d\n", keyboard\_state, j);

return j;

}

## Appendix C: The AI-player: player.py

The player.py, and all other source code is available on the supplied USB drive, but it is also reproduced here. The source code has been broken up into several sub-appendices to improve readability. In places, commented out lines of code have been removed, and long lines have been broken up. Functions relevant to each other have also been grouped together into one appendix section.

### Appendix C.1: player.py : globals, imports, and enums.

This section covers all the import statements, globals used throughout (for example screen size), the bit patterns used to detect numbers, words, pipes, and so on.

import asyncio

from enum import Enum

import click

import json

import pprint

import pygame

import random

import time

import sys

import socket

import numpy as np

import struct

import select

import stat

import os

import datetime

import tensorflow as tf

import gameplay

NES\_WIDTH = 256

NES\_HEIGHT = 240

screen\_size = [3 \* NES\_WIDTH, 3 \* NES\_HEIGHT ]

# neural net for the game play

gameplay\_nn = None

# Class of enum to differentiate between the types of black screens we can see

# e.g. start of game (displaying "world n-m", fully black, displaying game over etc)

class BlackScreen(Enum):

NotBlack = 0

JustBlack = 1

World = 2

GameOver = 3

# "Oneshot" play - exit after mario dies or completes the first level (second time the

# game displays a "world" on a black background

# FIXME: Should be a parameter

oneshot\_play = True

screenshot\_dir = "./screenshots"

tensorflow\_frozen\_graph = "tensorflow/mario-model-simple-2019-07-17/frozen\_inference\_graph.pb"

# The brown block obstacles go from from block\_col\_1 to block\_col\_2

obstacle\_block\_col\_1 = [ 240, 208, 176 ]

obstacle\_block\_col\_2 = [ 228, 92, 16 ]

obstacle\_block\_max\_dist = 50 # How many pixels to check in front of mario

# Used by the find\_horizontal\_objs to differentiate objects.

# colseq is basically the expected colour order of an object, e.g. a pipe

# goes from black, to light green, to dark green, to light green etc.

# Width is the width of the object

dumb\_detection = {'pipe': {

'colseq': [[184, 248, 24], [0, 168, 0], [184, 248, 24],

[0, 168, 0], [184,248,24], [0 ,168, 0], [184, 248, 24]],

'width' : 29

},

'obstacle': {

'colseq' : [ [240, 208, 176], [228, 92, 16], [0, 0, 0] ],

'width': 14

}

}

# pixels for various words

black\_text\_words = [

{ 'label': 'world',

'id' : BlackScreen.World,

'x1': 88,

'y1': 80,

'x2': 126,

'y2': 86,

'pix': [[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0],

[252, 252, 252], [0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [0, 0, 0], [252, 252, 252]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [0, 0, 0], [0, 0, 0],

[0, 0, 0], [252, 252, 252]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]] ]

},

{ 'label': 'game',

'id' : BlackScreen.GameOver,

'x1': 88,

'y1': 128,

'x2': 118,

'y2': 134,

'pix': [[[0, 0, 0], [0, 0, 0], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [0, 0, 0]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [0, 0, 0], [0, 0, 0],

[0, 0, 0], [252, 252, 252]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [252, 252, 252],

[0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [0, 0, 0], [0, 0, 0],

[252, 252, 252], [0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [0, 0, 0],

[252, 252, 252], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [0, 0, 0], [0, 0, 0],

[252, 252, 252], [0, 0, 0]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[0, 0, 0], [0, 0, 0]],

[[0, 0, 0], [0, 0, 0], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [0, 0, 0]],

[[0, 0, 0], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [252, 252, 252], [0, 0, 0],

[0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [252, 252, 252], [0, 0, 0],

[0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [252, 252, 252], [0, 0, 0],

[0, 0, 0]],

[[252, 252, 252], [0, 0, 0], [0, 0, 0], [252, 252, 252], [0, 0, 0],

[0, 0, 0]]]

}

]

# These are pixel value maps for the various numbers displayed e.g. for the time

# None = don't care about the actual pixel on screen for a particular position when

# comparing patterns

number\_pixels = {

'0': [[None, None, [252, 252, 252], [252, 252, 252], [252, 252, 252], None],

[None, [252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], None, None, None, None, [252, 252, 252]],

[[252, 252, 252], None, None, None, None, None],

[[252, 252, 252], [252, 252, 252], None, None, None, None],

[None, [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]]],

'1': [[None, None, None, None, None, None],

[None, None, None, None, None, None],

[None, [252, 252, 252], None, None, None, None],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[None, None, None, None, None, None]],

'2': [[None, [252, 252, 252], None, None, None, [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], None, None, [252, 252, 252],

[252, 252, 252]],

[[252, 252, 252], None, None, [252, 252, 252], [252, 252, 252],

[252, 252, 252]],

[[252, 252, 252], None, None, [252, 252, 252], [252, 252, 252], None],

[[252, 252, 252], None, [252, 252, 252], [252, 252, 252],

[252, 252, 252], None],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

None, None]],

'3': [[None, None, None, None, None, [252, 252, 252]],

[[252, 252, 252], None, None, None, None, [252, 252, 252]],

[[252, 252, 252], None, None, [252, 252, 252], None, None],

[[252, 252, 252], None, [252, 252, 252], [252, 252, 252], None, None],

[[252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], None, None],

[[252, 252, 252], [252, 252, 252], None, [252, 252, 252],

[252, 252, 252], [252, 252, 252]]],

'4': [[None, None, None, [252, 252, 252], [252, 252, 252], None],

[None, None, [252, 252, 252], [252, 252, 252], [252, 252, 252], None],

[None, [252, 252, 252], [252, 252, 252], None, [252, 252, 252], None],

[[252, 252, 252], [252, 252, 252], None, None, [252, 252, 252], None],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]]],

'5': [[None, [252, 252, 252], [252, 252, 252], None, None, [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], None, None,

[252, 252, 252]],

[[252, 252, 252], None, [252, 252, 252], None, None, None],

[[252, 252, 252], None, [252, 252, 252], None, None, None],

[[252, 252, 252], None, [252, 252, 252], None, None, None],

[[252, 252, 252], None, [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]]],

'6': [[None, None, [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252]],

[None, [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], None, [252, 252, 252], None, None],

[[252, 252, 252], None, None, [252, 252, 252], None, None],

[[252, 252, 252], None, None, [252, 252, 252], None, None],

[[252, 252, 252], None, None, [252, 252, 252], [252, 252, 252],

[252, 252, 252]]],

'7': [[None, [252, 252, 252], None, None, None, None],

[[252, 252, 252], [252, 252, 252], None, None, None, None],

[[252, 252, 252], None, None, None, [252, 252, 252], [252, 252, 252]],

[[252, 252, 252], None, None, [252, 252, 252], [252, 252, 252],

[252, 252, 252]],

[[252, 252, 252], None, [252, 252, 252], [252, 252, 252], None, None],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], None, None, None]],

'8': [[None, [252, 252, 252], [252, 252, 252], None, [252, 252, 252],

[252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]],

[[252, 252, 252], None, None, [252, 252, 252], None, None],

[[252, 252, 252], None, None, [252, 252, 252], None, None],

[[252, 252, 252], None, None, [252, 252, 252], None, None],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]]],

'9': [[None, [252, 252, 252], [252, 252, 252], None, None, None],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

None, None],

[[252, 252, 252], None, None, [252, 252, 252], None, None],

[[252, 252, 252], None, None, [252, 252, 252], None, None],

[[252, 252, 252], None, None, [252, 252, 252], None, [252, 252, 252]],

[[252, 252, 252], [252, 252, 252], [252, 252, 252], [252, 252, 252],

[252, 252, 252], [252, 252, 252]]],

}

### Appendix C.2: player.py : drawing to screen

This section contains code relevant to drawing pixels on the actual screen.

def setup\_screen():

pygame.init()

return pygame.display.set\_mode(screen\_size)

def draw\_nes\_screen(screen, nes\_screen):

""" Draw the nes screen buffer to the screen """

pygame.surfarray.blit\_array(screen, nes\_screen)

### Appendix C.3: player.py : Connect to emulator and send TCP data

This section contains code relevant to connecting to the emulator and sending strings of data to the emulator.

def connect\_to\_game\_server(address, port):

# type: (str, int) -> socket

""" Connect to a game server. Returns a socket.

Raises: ConectionRefusedError

"""

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

sock.connect((address, port))

return sock

def clear\_socket(sock):

""" Read from socket until there's nothing else to read (i.e. drain it) """

while True:

(readable, writable, exceptions) = select.select([sock], [], [], 0)

if len(readable) > 0:

# there is data to be read

data = sock.recv(4096)

if len(data) <= 0:

# actually, no there wasn't...

break

else:

# no data to read (a read would block)

break

def send\_to\_socket(sock, message):

# type: (socket, str) -> None

""" Send the message to the socket. """

while len(message) > 0:

bytes\_sent = sock.send(message.encode())

message = message[bytes\_sent:]

### Appendix C.4: player.py : Get emulator screen data

This section contains functions relevant to getting the screen contents from the emulator.

def get\_nes\_screen\_binary(sock):

""" Get the latest screen from the NES server using the

binary protocol (basically just a dump of the screen

buffer).

Returns a numpy 3d array of [x][y][r,g,b].

"""

# send command

clear\_socket(sock)

send\_to\_socket(sock, "binscreen\n")

recvd\_bytes = 0

pixels = bytearray()

# FIXME: Make this non-blocking (using the same select as in the clear\_socket

while (recvd\_bytes < NES\_HEIGHT\*NES\_WIDTH\*3):

# keep looping until full message is received

data = sock.recv(4096)

recvd\_bytes = recvd\_bytes + len(data)

pixels += data

# Make the bytearray a numpy array. Also, we sometimes get some

# extra data back (I blame the \n characters..), so we make sure that

# the array is the right size as well (otherwise reshape etc fails)

pixels\_np = np.array(pixels, dtype=np.uint8)

# Our array needs to be reshaped and then transposed, because otherwise

# we end up with each line offset (256-240) pixels, and also drawn 90 degrees

# rotated anti clock wise :(

screen = pixels\_np.reshape((NES\_HEIGHT, NES\_WIDTH, -1))

return screen

def get\_nes\_screen(sock):

# type: (socket) -> list

""" Get the latest screen from the NES server.

FIXME: This should really be non-blocking...

Returns a numpy 3d array of [x][y][r,g,b] to represent the pixels.

The size in y is NES\_HEIGHT, and NES\_WIDTH in x.

"""

# Send the command to get the remote to send the screen data

send\_to\_socket(sock, "screen\n")

done = False

# numpy array to hold the nintendo pixels

screen = np.zeros(shape=(NES\_WIDTH, NES\_HEIGHT, 3), dtype=np.int32)

message\_parts = []

while(not done):

bytes = sock.recv(4096)

decoded\_bytes = bytes.decode()

if len(bytes) > 0:

message\_parts.append(decoded\_bytes)

# look for the 'e' in "Done" - FIXME: Ugly..

if decoded\_bytes.find('e') > 0:

done = True

if bytes == 0:

done = True

received\_message = "".join(message\_parts)

# The string coming back has \n and spaces in it - lets get rid of them.

clean\_msg = received\_message.replace('display:', '').replace('\n', '').replace(' ', '')

# split the clean\_msg on comma

tokens = clean\_msg.split(',')

# remove the first 'display:' bit.

# FIXME: should probably check if we actually got valid screen back rather than just dropping

if tokens[0] == 'display:':

tokens.pop(0)

if tokens[-1] == 'message:Done':

tokens.pop(-1)

cur\_pos = 0

for y in range(NES\_HEIGHT):

for x in range(NES\_WIDTH):

(r, g, b) = tokens[cur\_pos:cur\_pos + 3]

screen[x][y] = (int(r), int(g), int(b))

cur\_pos = cur\_pos + 3

return screen

### Appendix C.5: player.py : Send key press, power off, and reset messages

This section contains functions relevant to sending complete messages to the emulator API.

def send\_key\_to\_emulator(sock, key\_state):

# type: (socket, int) -> None

""" Send a new joypad state to the emulator """

clear\_socket(sock)

send\_to\_socket(sock, "j {}\n".format(key\_state))

def send\_reset\_to\_emulator(sock):

# type: (socket) -> None

""" Send a reset to the emulator """

clear\_socket(sock)

send\_to\_socket(sock, "reset\n")

def send\_poweroff\_to\_emulator(sock):

# type: (socket) -> None

""" Send a poweroff to the emulator """

clear\_socket(sock)

send\_to\_socket(sock, "poweroff\n")

def calculate\_key\_value(key\_states):

# type: (dict) -> int

""" From the key\_states dict, calculate what the status

byte to send the emulator should be. """

j = 1 if key\_states.get('a') else 0

j = j | (2 if key\_states.get('b') else 0)

j = j | (4 if key\_states.get('select') else 0)

j = j | (8 if key\_states.get('start') else 0)

j = j | (16 if key\_states.get('up') else 0)

j = j | (32 if key\_states.get('down') else 0)

j = j | (64 if key\_states.get('left') else 0)

j = j | (128 if key\_states.get('right') else 0)

return j

### Appendix C.6: player.py : Taking screenshots

Functions to take screenshots and save them to disk.

def take\_screenshot(surface, path=screenshot\_dir):

# type: (surface, str) -> None

""" Save the surface as a screen shot"""

# create directory if it doesn't exist

try:

stat\_info = os.stat(path)

except OSError:

# failed to stat - path probably does not exist

try:

os.mkdir(path)

except:

# raise all exceptions here...

raise

time\_now = datetime.datetime.now()

screenshot\_name = "screenshot-{:04d}-{:02d}-{:02d}-{:02d}{:02d}{:02d}.jpg".format(time\_now.year, time\_now.month,

time\_now.day, time\_now.hour,

time\_now.minute, time\_now.second)

print("Screenshot name = {}".format(screenshot\_name))

pygame.image.save(surface, "{}/{}".format(path, screenshot\_name))

### Appendix C.7: player.py : TensorFlow object detetion

Appendix C.7 contains TensorFlow functions for object detection. It is based on the TensorFlow object detection tutorial at <https://github.com/tensorflow/models/blob/master/research/object_detection/object_detection_tutorial.ipynb>, but the code has been refactored from a monolithic function into several sub-functions.

There is also code produced in this project inter-woven into the code in this appendix, and to make it clear which lines belong to which category, the code that should be rightfully attributed to the TensorFlow object detection tutorial has been highlighted in yellow.

def load\_tensorflow\_graph(path):

# type: str -> tf.Graph

""" Load a frozen tensorflow graph at 'path'.

Note, the original version of this function came from the TensorFlow tutorials at

https://github.com/tensorflow/models/blob/master/research/object\_detection/object\_detection\_tutorial.ipynb

"""

print("load\_tensorflow\_graph: path = {}".format(path))

detection\_graph = tf.Graph()

with detection\_graph.as\_default():

od\_graph\_def = tf.GraphDef()

with tf.gfile.GFile(path, 'rb') as fid:

serialized\_graph = fid.read()

od\_graph\_def.ParseFromString(serialized\_graph)

tf.import\_graph\_def(od\_graph\_def, name='')

return detection\_graph

def detect\_objects\_in\_surface(surface, graph, image\_tensor, tensor\_dict, tf\_session):

# type: (pygame.Surface, tf.Graph) -> list

""" Detect objects in the pygame surface.

This function used a function from object\_detection\_tutorial

"""

return\_list = []

# get the pygame surface as a 3d (r,g,b) array

new\_surf = pygame.transform.scale(surface, (256\*3, 240\*3))

image = pygame.surfarray.array3d(new\_surf)

output\_dict = tf\_session.run(tensor\_dict, feed\_dict={image\_tensor: np.expand\_dims(image, 0)})

# all outputs are float32 numpy arrays, so convert types as appropriate

output\_dict['num\_detections'] = int(output\_dict['num\_detections'][0])

output\_dict['detection\_classes'] = output\_dict[

'detection\_classes'][0].astype(np.uint8)

output\_dict['detection\_boxes'] = output\_dict['detection\_boxes'][0]

output\_dict['detection\_scores'] = output\_dict['detection\_scores'][0]

if 'detection\_masks' in output\_dict:

output\_dict['detection\_masks'] = output\_dict['detection\_masks'][0]

for i in range(0, output\_dict['num\_detections']):

obj\_id = int(output\_dict['detection\_classes'][i])

y = int(output\_dict['detection\_boxes'][i][0] \* NES\_HEIGHT)

x = int(output\_dict['detection\_boxes'][i][1] \* NES\_WIDTH)

y2 = int(output\_dict['detection\_boxes'][i][2] \* NES\_HEIGHT)

x2 = int(output\_dict['detection\_boxes'][i][3] \* NES\_WIDTH)

score = output\_dict['detection\_scores'][i]

return\_list.append([y, x, y2, x2, obj\_id, score])

return return\_list

def setup\_tf\_detection\_vars(graph):

with graph.as\_default():

# Get handles to input and output tensors

ops = tf.get\_default\_graph().get\_operations()

all\_tensor\_names = {output.name for op in ops for output in op.outputs}

tensor\_dict = {}

for key in [

'num\_detections', 'detection\_boxes', 'detection\_scores',

'detection\_classes', 'detection\_masks'

]:

tensor\_name = key + ':0'

if tensor\_name in all\_tensor\_names:

tensor\_dict[key] = tf.get\_default\_graph().get\_tensor\_by\_name(

tensor\_name)

if 'detection\_masks' in tensor\_dict:

# The following processing is only for single image

detection\_boxes = tf.squeeze(tensor\_dict['detection\_boxes'], [0])

detection\_masks = tf.squeeze(tensor\_dict['detection\_masks'], [0])

# Reframe is required to translate mask from box coordinates to image coordinates and fit the image size.

real\_num\_detection = tf.cast(tensor\_dict['num\_detections'][0], tf.int32)

detection\_boxes = tf.slice(detection\_boxes, [0, 0], [real\_num\_detection, -1])

detection\_masks = tf.slice(detection\_masks, [0, 0, 0], [real\_num\_detection, -1, -1])

detection\_masks\_reframed = utils\_ops.reframe\_box\_masks\_to\_image\_masks(

detection\_masks, detection\_boxes, image.shape[0], image.shape[1])

detection\_masks\_reframed = tf.cast(

tf.greater(detection\_masks\_reframed, 0.5), tf.uint8)

# Follow the convention by adding back the batch dimension

tensor\_dict['detection\_masks'] = tf.expand\_dims(

detection\_masks\_reframed, 0)

image\_tensor = tf.get\_default\_graph().get\_tensor\_by\_name('image\_tensor:0')

return (image\_tensor, tensor\_dict)

### Appendix C.8: player.py : Pixel based word and number object detetion

Functions for detecting words and numbers on the NES screen using pixel-comparison methods.

def is\_black\_screen(surface):

""" Check to see if the display is mainly black. Checks a small number of pixels to

see if any of them are non-black """

sample\_locs = [[255, 100], [100, 100], [0, 239]]

pix\_arr = pygame.surfarray.pixels3d(surface)

for p in sample\_locs:

pix = pix\_arr[p[0], p[1]]

pv = pix[0] + pix[1] + pix[2]

if pv > 0:

return False

return True

def check\_black\_screen\_text(surface):

""" Check to see if the display is black, and if so if any text is displayed """

rv = BlackScreen.NotBlack

if is\_black\_screen(surface):

rv = BlackScreen.JustBlack # Default if the screen is black

pix\_arr = pygame.surfarray.pixels3d(surface)

# Check to see if any of the black text words matxh in our expected positions

for btw in black\_text\_words:

sub\_pixels = pix\_arr[btw['x1']:btw['x2'], btw['y1']:btw['y2']].tolist()

if sub\_pixels == btw['pix']:

rv = btw['id']

break

return rv

def check\_number(pix\_arr, loc):

""" Check to see if a number exists in loc in the pixel array.

Note, this will not work close to the left or bottom screen edge!

Returns: number, or None """

for n\_id in number\_pixels:

n = number\_pixels[n\_id]

n\_width = len(n[0])

n\_height = len(n)

sub\_pixels = pix\_arr[loc[0]:loc[0] + n\_width, loc[1]:loc[1] + n\_height].tolist()

# Check if the sub\_pixels match the pattern in the number\_pixels arrays

match = True

for r\_id in range(0, n\_height):

for c\_id in range(0, n\_width):

if number\_pixels[n\_id][r\_id][c\_id] is None:

# Don't compare Nones

continue

elif number\_pixels[n\_id][r\_id][c\_id] == sub\_pixels[r\_id][c\_id]:

continue

else:

# No match - break out

match = False

break

if match:

return int(n\_id)

# Nothing found...

return None

def get\_time\_remaining(surface):

""" Check to see if any numbers are present in the time section of the screen """

# These are the pixel locations where the time numbers start

sample\_locs = [[208, 24], [216, 24], [224, 24]]

pix\_arr = pygame.surfarray.pixels3d(surface)

numbers\_found = ''

for loc in sample\_locs:

n = check\_number(pix\_arr, loc)

if n is not None:

numbers\_found += str(n)

if numbers\_found:

return int(numbers\_found)

else:

return None

### Appendix C.9: player.py : Detection of holes

Functions for detecting holes that Mario can fall down into and die.

def detect\_holes(surface):

""" Detect 'holes' that Mario can fall in to (and die).

Returns list of pairs of holes, where the first element is the beginning

of the hole in the X axis, and the second value is the the end of the hole

in the X axis (all in pixels) """

holes = []

pix\_arr = pygame.surfarray.pixels3d(surface)

# We only care about the bottom row, and we calculate everything that is blue on

# the lowest line as a hole.

lowest\_line = pix\_arr[:,NES\_HEIGHT-1]

# The colour of 'holes'

hole\_col = np.array([104, 136, 252], dtype=np.uint8)

# little state machine for picking out the holes

in\_hole = list(lowest\_line[0]) == list(hole\_col)

hole\_start = 0

# iterate across all pixels

for p\_idx in range(0, len(lowest\_line)):

p = lowest\_line[p\_idx]

if in\_hole and list(p) != list(hole\_col):

# transition to out of hole

holes.append([hole\_start, p\_idx - 1])

in\_hole = False

elif (not in\_hole) and list(p) == list(hole\_col):

# new hole detected

hole\_start = p\_idx

in\_hole = True

if in\_hole:

# If we end with a hole, we capture the end of the last hole here

holes.append([hole\_start, NES\_WIDTH - 1])

return holes

### Appendix C.10: player.py : Check screen scroll

Check if the screen has scrolled, because Mario has moved to the right.

def check\_screen\_scroll(surface, moves\_to\_right, leftmost\_pixels):

""" Check lower portion of leftmost column of pixels. If there is

any change to the previously seen values, assume we've moved

the screen, and in that case return an increased moves\_to\_right

counter. """

pix\_arr = pygame.surfarray.pixels3d(surface)

# If leftmost\_pixels is None, this is the first time we are called.

if leftmost\_pixels is None:

leftmost\_pixels = pix\_arr[0, NES\_HEIGHT-20:NES\_HEIGHT].copy()

return (0, leftmost\_pixels)

else:

new\_pixels = pix\_arr[0, NES\_HEIGHT-20:NES\_HEIGHT].copy()

for p\_idx in range(0, len(new\_pixels)):

if list(new\_pixels[p\_idx]) != list(leftmost\_pixels[p\_idx]):

# Not the same, so we must have moved

moves\_to\_right += 1

break

return (moves\_to\_right, new\_pixels)

### Appendix C.11: player.py : Check for blocks in front of Mario

Function to check if there is a block in front of Mario.

def check\_forward\_obstacles(surface, mario\_pos):

""" Check if there's a block in front of Mario (to the right only).

If yes, return the distance in pixels. If no, return NES\_WIDTH """

pix\_arr = pygame.surfarray.pixels3d(surface)

if mario\_pos is None:

return NES\_WIDTH

y\_loc = (mario\_pos[2] - 10) % NES\_HEIGHT

x\_loc = (mario\_pos[3] + 10) % NES\_WIDTH

# Check if we see the pattern block\_col1, then block\_col2, and if so assume we are in

# front of a block

for p in range(x\_loc, (x\_loc + obstacle\_block\_max\_dist) % NES\_WIDTH):

if list(pix\_arr[p, y\_loc]) == obstacle\_block\_col\_1:

# Check 6 pixels ahead

next\_loc = p + 6

if next\_loc >= NES\_WIDTH:

# Out of the screen

break

if list(pix\_arr[next\_loc, y\_loc]) == obstacle\_block\_col\_2:

# Found a block

return max(p - x\_loc, 0) # Delta distance between mario and beginning of block

return NES\_WIDTH

### Appendix C.12: player.py : Find horizontal objects in front of Mario

Function to check if well-defined objects exist in front of Mario. This function uses the dictionary of objects defined at the top of player.py (Appendix C.1) for expected pixel values for a particular type of object. If the pixel values match anywhere along a horizontal line drawn across the middle of Mario, an object is assumed to exist, and the location is returned.

def find\_horizontal\_objs(surface, mario\_pos):

""" See if there are defined objects in front or back of mario.

Returns list of:

{ type\_id : <type>, pos\_x: <int>, pos\_y: <int>, start: <int>, end: <int> } """

pix\_arr = pygame.surfarray.pixels3d(surface)

if mario\_pos is None:

return []

y\_loc = (mario\_pos[2] - 10) % NES\_HEIGHT

x\_loc = (mario\_pos[3] + 10) % NES\_WIDTH

objs\_detected = []

pix\_iterator = iter(range(0, NES\_WIDTH -1))

for p in pix\_iterator:

for type\_id in dumb\_detection.keys():

obj\_detected = False

# Check if first colour matches, otherwise try next pix

if list(pix\_arr[p, y\_loc]) == dumb\_detection[type\_id]['colseq'][0]:

# Iterate from here, seeing if we can spot the second colour

cur\_col = dumb\_detection[type\_id]['colseq'][0]

col\_seq = dumb\_detection[type\_id]['colseq'].copy()

col\_seq.pop(0) # Remove first colour

expected\_next\_colour = col\_seq.pop(0)

for p\_in\_pipe in range(p + 1, min(p + 30, NES\_WIDTH)):

# colour of current pixel

new\_pix = pix\_arr[p\_in\_pipe, y\_loc]

if list(new\_pix) == cur\_col:

# Still same colour - skip

continue

if list(new\_pix) == expected\_next\_colour:

# Still in sequence, but new colour

if len(col\_seq) == 0:

# We've run out of colours, this is a complete section

##print("Detected object at {}".format(p))

obj\_detected = True

break

else:

cur\_col = list(new\_pix)

expected\_next\_colour = col\_seq.pop(0)

else:

# Other colour - not our object

break

if obj\_detected:

###print("Object detected at {}".format(p))

objs\_detected.append({'type\_id': type\_id, 'pos\_x': p, 'pos\_y': y\_loc })

# Now, also skip width pixels

for \_ in range(0, dumb\_detection[type\_id]['width']):

next(pix\_iterator, None)

for od in objs\_detected:

type\_id = od['type\_id']

end\_x = min(od['pos\_x'] + dumb\_detection[type\_id]['width'], NES\_WIDTH)

od['start'] = od['pos\_x']

od['end'] = end\_x

return objs\_detected

### Appendix C.13: player.py : Filter false TensorFlow matches

TensorFlow has problems differentiating normal blocks with a question mark on and the Koopa-Troopa enemy, even though a human has no problem distinguishing the two objects. This function checks if the particular white colour of a Koopa-Troopa’s neck exists in the bounding box to determine if the bounding box contains a real Koopa-Troopa, or a false positive.

def filter\_false\_positives(surface, obj\_boxes):

""" Try to filter out false positives. For example, the secret boxes with

question marks are often detected as koopa troopas (probably due to the

question mark looking a bit like a koopas neck)."""

# We're going to be studying individual pixels.

pix\_arr = pygame.surfarray.pixels3d(surface)

# List to hold the objects we need to delete from the list of objs

# (i.e. false positives)

drop\_idx = []

# analyse objects, and filter out problematic / incorrect ones.

# So far, only the question mark blocks seem to be mis-identified

# as koopa-troopas, so they get filtered out here.

for b\_idx in range(0, len(obj\_boxes)):

b = obj\_boxes[b\_idx]

if b[4] == 3:

# Potential koopa-troopa

is\_koopa = False

for x in range(b[0], b[2]):

for y in range(b[1], b[3]):

# Koopa Troopas are unique compared to the blocks, in having the

# white colour 252, 252, 252

if list(pix\_arr[x, y]) == [252, 252, 252]:

is\_koopa = True

break

if is\_koopa:

break

if not is\_koopa:

drop\_idx.append(b\_idx)

# Finally, drop the detections that were determined to be false

for d in sorted(drop\_idx, reverse=True):

del obj\_boxes[d]

return obj\_boxes

### Appendix C.14: player.py : Draw bounding boxes

This function converts the TensorFlow detection ID numbers and bounding boxes into a list of a dictionaries, which is then used internally in the player.py.

The function also draws bounding boxes in the appropriate colours on the screen.

def build\_detected\_objects\_dict(surface, obj\_boxes):

detected\_objects = {'mario': [],

'goomba': [],

'koopa-troopa': [],

'pipe': [],

'flag': []

}

for b in obj\_boxes:

# Make a dict up with the detected objects

colour = (0, 255, 25 \* b[4])

obj\_id = None

if b[4] == 1:

# Mario - only one hopefully

obj\_id = 'mario'

colour = (255, 0, 0)

elif b[4] == 2:

obj\_id = 'goomba'

elif b[4] == 3:

obj\_id = 'koopa-troopa'

elif b[4] == 4:

obj\_id = 'flag'

if obj\_id:

detected\_objects[obj\_id].append({'pos': [b[0], b[1], b[2], b[3]],

'score': b[5]})

pygame.draw.rect(surface, colour, (b[0], b[1], b[2] - b[0], b[3] - b[1]), 3)

else:

pygame.draw.rect(surface, (0, 0, 255), (b[0], b[1], b[2] - b[0], b[3] - b[1]), 3)

return detected\_objects

### Appendix C.15: player.py : Start sequence

This function simulates a press of the “start” button, to start the game.

def do\_start\_sequence(sock):

""" Perform the mario start sequence (press start, wait for a bit) """

print("Doing start sequence")

# Sleep 1 second

print("Sleeping one second")

time.sleep(1)

# Press start button

print("sending start")

send\_key\_to\_emulator(sock, calculate\_key\_value({'start': True}))

time.sleep(1)

print("sending empty")

send\_key\_to\_emulator(sock, calculate\_key\_value({}))

### Appendix C.16: player.py : Build key state dictionary

This function converts PyGame key press events to a key\_state dictionary.

def handle\_pygame\_key\_events(event, key\_states):

""" Given a pygame event, and a key\_states dict, modify key\_states dict

and return the modified version """

if event.type == pygame.KEYDOWN or event.type == pygame.KEYUP:

if event.key == pygame.K\_UP:

key\_states['up'] = event.type == pygame.KEYDOWN

elif event.key == pygame.K\_DOWN:

key\_states['down'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_LEFT:

key\_states['left'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_RIGHT:

key\_states['right'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_a:

key\_states['a'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_s: # NOTE s, not b

key\_states['b'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_RETURN:

key\_states['start'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_q: # Use q for select (because space for screen shot)

key\_states['select'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_h:

key\_states['h'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_j:

key\_states['j'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_k:

key\_states['k'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_l:

key\_states['l'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_y:

key\_states['y'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_u:

key\_states['u'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_i:

key\_states['i'] = event.type == pygame.KEYDOWN

if event.key == pygame.K\_o:

key\_states['o'] = event.type == pygame.KEYDOWN

return key\_states

### Appendix C.17: player.py : Normalise object detection boxes

These functions normalise detected objects to be centred around Mario

def get\_mid\_of\_box(pos):

""" Calculate the middle of a position box """

return [(pos[2] - pos[0]) / 2 + pos[0],

(pos[3] - pos[1]) / 2 + pos[1]]

def obj\_detection\_boxes\_normalise(detected\_objects, mario\_mid):

for obj\_id in detected\_objects:

for b in detected\_objects[obj\_id]:

obj\_mid = get\_mid\_of\_box(b['pos'])

obj\_width = b['pos'][2] - b['pos'][0]

# Rel pos: negative when object is to the right of mario, and down

rel\_pos = [mario\_mid[1] - obj\_mid[1], mario\_mid[0] - obj\_mid[0]]

b['rel'] = rel\_pos

b['width'] = obj\_width

b['norm\_width'] = b['width'] / NES\_WIDTH

# Only care about 100 pix to either side of mario

rel\_x\_dist = b['rel'][0] / NES\_WIDTH + 0.5

rel\_y\_dist = b['rel'][1] / NES\_WIDTH + 0.5

b['norm\_pos'] = [rel\_x\_dist, rel\_y\_dist]

return detected\_objects

### Appendix C.18: player.py : Main player loop

The main\_loop() function runs until the Mario dies (if in AI-mode), or until the human player quits the game. It is responsible for:

* Getting a screen update
* Call functions to do object detection (TensorFlow and pixel based)
* Call the gameplay neural network for an action
* Sending actions and key presses to the emulator

def main\_loop(screen, sock):

""" Main game loop """

# create a surface for the NES

nes\_surface = pygame.Surface((NES\_HEIGHT, NES\_WIDTH))

nes\_surface.convert()

nes\_surface.fill((0, 0, 128)) # dark blue background

# Load a frozen tensorflow graph

object\_detection\_graph = load\_tensorflow\_graph(tensorflow\_frozen\_graph)

print("object\_detection\_graph = {}".format(object\_detection\_graph))

# Cerate a tf.Session object, so that we don't have to recreate it every time we

# run inference

(image\_tensor, tensor\_dict) = setup\_tf\_detection\_vars(object\_detection\_graph)

clock = pygame.time.Clock()

running = True

# The possible joypad states

key\_states = { 'up': False,

'down': False,

'left': False,

'right': False,

'a': False,

'b': False,

'start': False,

'select': False }

key\_state = 0

get\_new\_screen\_contents = True

old\_screen\_contents = None

# "game" time left

remaining\_seconds = None

# Indicator dot

dot\_x\_y = [0, 0]

mark\_p1 = [0, 0]

mark\_p2 = [0, 0]

moves\_to\_right = 0 # Count how many times the leftmost column of pixels has changed

# as a proxy for movement towards the right.

# How many times has screen transitioned to BlackScreen.World

trans\_to\_blackscreen\_world = 0

previous\_blackscreen = None

# What action should Mario take next

next\_action = None

# If true, we'll use the neural net inputs for game play

do\_ai\_play = False

if oneshot\_play:

do\_start\_sequence(sock)

do\_ai\_play = True

consecutive\_jumps = 0 # How many iterations have we held down the jump key

frame\_counter = 0

time\_start = time.time()

with object\_detection\_graph.as\_default():

leftmost\_pixels = None

old\_mario\_pos = None

tf\_session = tf.Session()

while running:

frame\_counter += 1

if get\_new\_screen\_contents:

nes\_screen\_contents = get\_nes\_screen\_binary(sock)

old\_screen\_contents = nes\_screen\_contents

else:

nes\_screen\_contents = old\_screen\_contents

# make a surface out of the screen contents

draw\_nes\_screen(nes\_surface, nes\_screen\_contents)

# try to detect objects in nes\_surface and draw bounding boxes

obj\_boxes = detect\_objects\_in\_surface(nes\_surface, object\_detection\_graph, image\_tensor,

tensor\_dict, tf\_session)

# Filter out false positives (e.g. question marks that are detecte as

# koopa troopas

obj\_boxes = filter\_false\_positives(nes\_surface, obj\_boxes)

# Categorize the detected objects

detected\_objects = build\_detected\_objects\_dict(nes\_surface, obj\_boxes)

# Make the surface point the right way - note if we do this before passing it into the

# object detection, we get terrible detection accuracy.

nes\_surface = pygame.transform.flip(nes\_surface, True, False)

# Now, rotate it 90 degrees anti-clock-wise

rotated\_surface = pygame.transform.rotate(nes\_surface, 90)

# Reference to the pixel array

pix\_arr = pygame.surfarray.pixels3d(rotated\_surface)

# Move indicator dot

if key\_states.get('h', False):

new\_x = dot\_x\_y[0] - 1 if dot\_x\_y[0] > 0 else 0

dot\_x\_y = [new\_x, dot\_x\_y[1]]

elif key\_states.get('l', False):

new\_x = dot\_x\_y[0] + 1 if dot\_x\_y[0] < 255 else 255

dot\_x\_y = [new\_x, dot\_x\_y[1]]

elif key\_states.get('j', False):

new\_y = dot\_x\_y[1] + 1 if dot\_x\_y[1] < 240 else 240

dot\_x\_y = [dot\_x\_y[0], new\_y]

elif key\_states.get('k', False):

new\_y = dot\_x\_y[1] - 1 if dot\_x\_y[1] > 0 else 0

dot\_x\_y = [dot\_x\_y[0], new\_y]

# Stop doing screen updates on 'y'

if key\_states.get('y', False):

print("Toggling screen updates")

get\_new\_screen\_contents = False if get\_new\_screen\_contents is True else True

# Record mark on i

if key\_states.get('i', False):

mark\_p1 = [dot\_x\_y[0], dot\_x\_y[1]]

print("MARK 1: {}".format(mark\_p1))

# Record mark on o

if key\_states.get('o', False):

mark\_p2 = [dot\_x\_y[0], dot\_x\_y[1]]

print("MARK 2: {}".format(mark\_p2))

# Dump subsection of screen as json on u

if key\_states.get('u', False):

print("Dumping section of screen")

sub\_pixels = pix\_arr[mark\_p1[0]:mark\_p2[0],

mark\_p1[1]:mark\_p2[1]].tolist()

print(json.dumps(sub\_pixels))

with open('dumpfile.txt', 'a+') as fd:

fd.write("{}\n\n".format(json.dumps(sub\_pixels)))

time.sleep(1)

#########################################################

# Figure out actual game state (apart from obj detection)

#########################################################

# Check if the screen has moved right

(moves\_to\_right, leftmost\_pixels) = check\_screen\_scroll(rotated\_surface, moves\_to\_right,

leftmost\_pixels)

# Check for game over screens etc (i.e. screens that are mostly black)

black\_screen\_state = check\_black\_screen\_text(rotated\_surface)

if black\_screen\_state != previous\_blackscreen:

# Transition to a different type of screen

previous\_blackscreen = black\_screen\_state

if black\_screen\_state == BlackScreen.World:

# We count how many times we have seen the "World" screen.

# When we first start playing, this counter is one at the time

# we get to play. If Mario dies (or the level completes),

# this counter will increment, and we bomb out with the

# current reward.

trans\_to\_blackscreen\_world += 1

if oneshot\_play and trans\_to\_blackscreen\_world >= 2:

# This is the end of the game.

print("Found black world screen for the second time. exiting.")

send\_poweroff\_to\_emulator(sock)

running = False # Causes game to quit.

# Check how many seconds are left on the clock

seconds\_left = get\_time\_remaining(rotated\_surface)

if seconds\_left:

# Save this value, so that we can reference it when we

# quit (and report game stats)

remaining\_seconds = seconds\_left

# Find all holes

holes = detect\_holes(rotated\_surface)

# Get current position of mario

mario\_pos = None

if len(detected\_objects['mario']) > 0:

mario\_pos = detected\_objects['mario'][0]['pos']

old\_mario\_pos = mario\_pos

else:

# If we can't find mario, assume he's in the old pos

mario\_pos = old\_mario\_pos

if not mario\_pos:

# Can't find mario, and we've never seen him

mario\_pos = [0, 0, 20, 20] # dummy values

# Check to see if there are any blocks/pipes etc in front

horizontal\_objects = find\_horizontal\_objs(rotated\_surface, mario\_pos)

# Build relative distances to mario of the objects

mario\_mid = get\_mid\_of\_box(mario\_pos)

# and now for the tensorflow detected objects

detected\_objects = obj\_detection\_boxes\_normalise(detected\_objects, mario\_mid)

for obj in horizontal\_objects:

type\_id = obj['type\_id']

if not detected\_objects.get(type\_id, None):

detected\_objects[type\_id] = []

obj\_width = obj['end'] - obj['start']

d = { 'rel': [ mario\_mid[1] - obj['pos\_x'] - obj\_width/2, 0],

'width': obj\_width, 'norm\_width': obj\_width / NES\_WIDTH

}

# Only care about 100 pix to either side of mario

rel\_dist = min(max(d['rel'][0], -100), 100)

d['norm\_pos'] = [ rel\_dist/200 + 0.5, 0.5] # 200 steps in total, centered around 0.5

detected\_objects[type\_id].append(d)

for hole in holes:

if not detected\_objects.get('hole', None):

detected\_objects['hole'] = []

width = hole[1] - hole[0]

d = {'rel': [mario\_mid[1] - hole[0] + width / 2, 0],

'width': hole[1] - hole[0],

'norm\_width': (hole[1] - hole[0]) / NES\_WIDTH

}

rel\_dist = min(max(d['rel'][0], -NES\_WIDTH), NES\_WIDTH)

d['norm\_pos'] = [ rel\_dist/(2 \* NES\_WIDTH) + 0.5, 0] # 200 steps in total, centered around 0.5

detected\_objects['hole'].append(d)

# determine the next action

next\_action = gameplay.run\_ann(detected\_objects, gameplay\_nn)

# Handle pygame events (button presses etc)

for event in pygame.event.get():

if event.type == pygame.QUIT:

send\_poweroff\_to\_emulator(sock)

running = False

# joypad buttons

key\_states = handle\_pygame\_key\_events(event, key\_states)

# Translate the "next action" generated to a key state

if do\_ai\_play:

if next\_action == 0:

# Do nothing - unset all buttons

key\_states['up'] = False

key\_states['down'] = False

key\_states['left'] = False

key\_states['right'] = False

key\_states['a'] = False

else:

# Note: next\_action values are:

# 0 = do nothing

# 1 = left

# 2 = right

# 3 = left and jump

# 4 = right and jump

key\_states['up'] = False

key\_states['down'] = False

key\_states['left'] = next\_action in [1, 3]

key\_states['right'] = next\_action in [2, 4]

key\_states['a'] = next\_action in [3, 4]

if key\_states['a']:

# Stop the model from holding down the jump key indefinitely

consecutive\_jumps += 1

if consecutive\_jumps > 3:

consecutive\_jumps = 0

key\_states['a'] = False

else:

consecutive\_jumps = 0

# Reset NES

if event.type == pygame.KEYDOWN and event.key == pygame.K\_ESCAPE:

send\_reset\_to\_emulator(sock)

elif event.type == pygame.KEYDOWN and event.key == pygame.K\_SPACE:

# Take screen shot

take\_screenshot(nes\_surface, path=screenshot\_dir)

elif event.type == pygame.KEYDOWN and event.key == pygame.K\_p:

# kill remote emulator and ourselves

send\_poweroff\_to\_emulator(sock)

running = False

# Now that we know our desired key state, calculate what value this

# corresponds to in the emulator (i.e. the bitmap).

# If it's different to the previous value, update the emulator

tmp\_key\_state = calculate\_key\_value(key\_states)

if tmp\_key\_state != key\_state:

# update the emulator

key\_state = tmp\_key\_state

send\_key\_to\_emulator(sock, key\_state)

# Draw indicator dots

pix\_arr[dot\_x\_y[0], dot\_x\_y[1]] = [ 0, 255, 0]

pix\_arr[mark\_p1[0], mark\_p1[1]] = [ 0, 255, 255]

pix\_arr[mark\_p2[0], mark\_p2[1]] = [ 255, 255, 0]

# scale and blit to screen

screen.blit(pygame.transform.scale(rotated\_surface, (2\*NES\_WIDTH, 2\*NES\_HEIGHT)), (0, 0))

pygame.display.flip()

# If we end up here, we have either chosen to quit, or we're in oneshot mode and

# Mario has died. Dump out the status

print("\n\n")

print("Seconds left: {}".format(remaining\_seconds))

print("Moves to the right: {}".format(moves\_to\_right))

print("frame count: {}".format(frame\_counter))

print("time start: {}, time\_now: {}".format(time\_start, time.time()))

### Appendix C.19: player.py : Initialization of the player process

This section of code reads command line parameters, initializes the screen, loads the gameplay neural network, connects to the emulator, and finally calls the main\_loop().

if \_\_name\_\_ == "\_\_main\_\_":

# Load the base neural net

my\_net = gameplay.neural\_net\_base\_def

# Parse some super simple command line args

oneshot\_play = 'oneshot' in sys.argv # Turn on ai player

if oneshot\_play:

# Load the neural network definition if we're playing in oneshot mode

try:

neural\_net\_file = sys.argv[2]

except:

print("Need to pass the name of the neural net file as 2nd param")

sys.exit(1)

my\_net = gameplay.load\_net\_from\_file(sys.argv[2])

gameplay\_nn = gameplay.build\_neural\_net(my\_net)

pprint.pprint(gameplay\_nn)

screen = setup\_screen()

sock = None

for i in range(8005, 8050):

try:

sock = connect\_to\_game\_server('localhost', i)

except ConnectionRefusedError:

# try next port

pass

if sock is None:

print("Did you forget to start the emulator?")

sys.exit(0)

main\_loop(screen, sock)

## Appendix D: gameplay.py : Neural network function for game-play

The gameplay.py file contains functions used by the AI-player to determine the next action to take. It implements a simple feed-forward network, as backpropagation is not used.

### Appendix D.1: gameplay.py : Globals and imports

Globals and imports for the neural net are defined here

""" Gameplay functions for super mario goes in here"""

import math

import numpy as np

import pprint

import json

import player

INFINITY\_DIST=1

### Appendix D.2: gameplay.py : Sigmoid function

The sigmoid function has been copied verbatim from <https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9>. It is used in the project but is not developed as part of it.

# From https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9

def sigmoid(x):

# Our activation function: f(x) = 1 / (1 + e^(-x))

return 1 / (1 + np.exp(-x))

### Appendix D.3: gameplay.py : Neuron class

The Neuron class has been copied verbatim from <https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9>. It is used in the project but is not developed as part of it.

# From https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9

class Neuron:

def \_\_init\_\_(self, weights, bias, activation):

self.weights = weights

self.bias = bias

self.activation = activation

def activate(self, inputs):

# Weight inputs, add bias, then use the activation function

total = np.dot(self.weights, inputs) + self.bias

return sigmoid(total) # FIXME: Always using sigmoid

### Appendix D.4: gameplay.py : Example neural net

Example neural net, showing the layers, weights and biases. The inputs to the gameplay networks are also shown.

""" Inputs:

goomba1\_x

goomba1\_y

goomba2\_x

goomba2\_y

koopa1\_x

koopa1\_y

object1 # x dist to the right

hole1\_x # hole x centre

holewidth # How wide the hole is

pipe\_x # nearest right pipe

"""

neural\_net\_base\_def = {

'inputs' : [], # list of 10 values

'generation': 1, # How many generations has this network gone through from the base

'layers': [

{ 'name': 'input',

'activation': 'sigmoid',

'num\_neurons': 8,

# as many weights as there are inputs, per neuron

'weights': [[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

[0.4, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1 ],

],

# as many biases as there are neurons

'bias': [0, 0, 0, 0, 0, 0, 0, 0 ],

'neurons': []

},

{'name': 'hidden1',

'activation': 'sigmoid',

'num\_neurons': 6,

'weights': [

[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],

[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],

[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],

[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],

[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],

[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],

[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],

[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],

],

'bias': [0, 0, 0, 0, 0, 0 ],

'neurons': []

},

{'name': 'output',

'activation': 'sigmoid',

'num\_neurons': 5,

'weights': [

[0.1, 0.1, 0.1, 0.1, 0.1, 0.1],

[0.1, 0.1, 0.1, 0.1, 0.1, 0.1],

[0.2, 0.2, 0.2, 0.2, 0.2, 0.2],

[0.1, 0.1, 0.1, 0.1, 0.1, 0.1],

[0.1, 0.1, 0.1, 0.1, 0.1, 0.1],

[0.1, 0.1, 0.1, 0.1, 0.1, 0.1],

],

'bias': [0, 0, 0, 0, 0 ],

'neurons': []

}

],

}

### Appendix D.5: gameplay.py : Save and load networks to file

Functions to save or load a neural network to/from file.

def dump\_net\_to\_file(net\_def, filename):

""" Write the neural net to filename (in JSON) """

print("DUMP TO FILE:\n{}\n".format(net\_def))

with open(filename, "w") as fd:

line = json.dumps(net\_def)

fd.write("{}\n".format(line))

print("Wrote neural net to {}".format(filename))

def load\_net\_from\_file(filename):

""" JSON load the contents of a file, and return the dict.

Used to load custom neural networks """

print("Loading neural net from {}".format(filename))

with open(filename, "r") as fd:

net = json.load(fd)

print("net = {}".format(pprint.pformat(net)))

return net

def build\_neural\_net(net\_def):

""" Return a fully populated neural net def """

populated\_def = net\_def.copy()

for layer in populated\_def['layers']:

for n in range(0, layer['num\_neurons']):

weights = layer['weights'][n]

bias = layer['bias'][n]

neuron = Neuron(weights, bias, layer['activation'])

layer['neurons'].append(neuron)

return populated\_def

### Appendix D.6: gameplay.py : Processing inputs and generating an output using neural net

These functions implement the feed forward calculations for the network layers. The returned value from feed\_forward\_net is the index of the neuron with the highest activation in the output layer.

def process\_layer(layer\_def, inputs):

""" Using the input, activate the layer, and return list of results """

outputs = []

for n in layer\_def['neurons']:

n\_res = n.activate(inputs)

outputs.append(n\_res)

return outputs

def feed\_forward\_net(net\_def, inputs):

""" Feed forward some inputs through the net. Returns the index of the element in the final layer

that has the largest value (so it kind of classifies a bit).

E.g. if output for the last layer is [ 0.3, 0.1, 0.8, 0.3 ], the function returns 2.

"""

inp = inputs.copy()

for n in range(0, len(net\_def['layers'])):

outputs = process\_layer(net\_def['layers'][n], inp)

inp = outputs.copy()

# Index of largest value

return np.argmax(inp)

def sorted\_objects(detected\_objects, keyname):

""" Sort the 'keyname' objects in detected\_objects by relative

distance to mario (ascending) """

if detected\_objects.get(keyname):

return sorted(detected\_objects[keyname],

key=lambda u: math.sqrt(u['norm\_pos'][0] \*\* 2 + u['norm\_pos'][1] \*\* 2))

else:

return []

### Appendix D.7: gameplay.py : Convert a game state to an action

The run\_ann() function is called from the player.py script, and it

def run\_ann(detected\_objects, nn):

""" Produce an action for mario, given the state of the world in terms of

detected objects """

inputs = []

# Find goombas

sorted\_goombas = sorted\_objects(detected\_objects, 'goomba')

if len(sorted\_goombas) > 0:

goomba1 = sorted\_goombas[0]

inputs.append(goomba1['norm\_pos'][0])

inputs.append(goomba1['norm\_pos'][1])

else:

inputs.append(INFINITY\_DIST)

inputs.append(INFINITY\_DIST)

if len(sorted\_goombas) > 1:

goomba2 = sorted\_goombas[1]

inputs.append(goomba2['norm\_pos'][0])

inputs.append(goomba2['norm\_pos'][1])

else:

inputs.append(INFINITY\_DIST)

inputs.append(INFINITY\_DIST)

# Find koopa troopas

sorted\_koopa = sorted\_objects(detected\_objects, 'koopa-troopa')

if len(sorted\_koopa) > 0:

koopa1 = sorted\_koopa[0]

inputs.append(koopa1['norm\_pos'][0])

inputs.append(koopa1['norm\_pos'][1])

else:

inputs.append(INFINITY\_DIST)

inputs.append(INFINITY\_DIST)

# Find obstacles

sorted\_obstacles = sorted\_objects(detected\_objects, 'obstacle')

if len(sorted\_obstacles) > 0:

object1 = sorted\_obstacles[0]

inputs.append(object1['norm\_pos'][0])

else:

inputs.append(INFINITY\_DIST)

# Find pipes

sorted\_pipes = sorted\_objects(detected\_objects, 'pipe')

if len(sorted\_pipes) > 0:

pipe = sorted\_pipes[0]

inputs.append(pipe['norm\_pos'][0])

else:

inputs.append(INFINITY\_DIST)

# Find holes

sorted\_holes = sorted\_objects(detected\_objects, 'hole')

if len(sorted\_holes) > 0:

hole = sorted\_holes[0]

inputs.append(hole['norm\_pos'][0])

inputs.append(hole['width'] / player.NES\_WIDTH)

else:

inputs.append(INFINITY\_DIST)

inputs.append(0)

action = feed\_forward\_net(nn, inputs)

return action

## Appendix E: Generate modified networks with generate\_network.py

The generate\_network.py script is used to modify an existing GPNN. It loads the specified existing network from file, modifies the values as described in chapter 3.10.3, and then saves the modified GPNN to disk.

""" This program generates a new network, using another network as a base. """

import click

import pprint

import random

import gameplay

def mutate\_net(net, max\_weight\_jitter, max\_bias\_jitter, mutation\_rate):

""" Modify a neural net by adding random jitter to it.

The jitter added is -max\_jitter to +max\_jitter, and

the likelihood of mutation occuring per weight and bias is given

by mutation rate"""

new\_net = net.copy()

for n in range(0, len(new\_net['layers'])):

layer = new\_net['layers'][n]

# Add jitter to the weights in the net

for w1 in layer['weights']:

for w2\_idx in range(0, len(w1)):

r = random.uniform(0, 1)

if r < mutation\_rate:

w1[w2\_idx] += random.uniform(-max\_weight\_jitter, max\_weight\_jitter)

# Add jitter to the biases

for b\_idx in range(0, len(layer['bias'])):

r = random.uniform(0, 1)

if r < mutation\_rate:

layer['bias'][b\_idx] += random.uniform(-max\_bias\_jitter, max\_bias\_jitter)

return new\_net

@click.command()

@click.option('--input', help="Use this network as a base, otherwise use built in base def")

@click.option('--weight-max-jitter', default=0.05, help="What's the maximum amount the weights will be adjusted")

@click.option('--bias-max-jitter', default=0.0, help="What's the maximum amount the bias will be adjusted")

@click.option('--mutation-likelihood', default=0.05, help="How likely is a given weight to change")

@click.argument('output')

def handle\_cmdline(input, weight\_max\_jitter, bias\_max\_jitter, mutation\_likelihood, output):

print("input = {}, output = {}".format(input, output))

if input:

input\_net = gameplay.load\_net\_from\_file(input)

else:

input\_net = gameplay.neural\_net\_base\_def

new\_net = mutate\_net(input\_net, weight\_max\_jitter, bias\_max\_jitter,

mutation\_likelihood)

new\_net['generation'] += 1

if output is not "":

gameplay.dump\_net\_to\_file(new\_net, output)

if \_\_name\_\_ == "\_\_main\_\_":

handle\_cmdline()

## Appendix F: Automate training of GPNN with train-player.sh

The train-player.sh script is used to automate the training of the GPNNs. It performs the following actions:

* Generate new networks from the network with the highest score in the previous generation
* Play the player.py script in AI-mode with all the newly generated networks and record the scores
* Repeat

#!/bin/bash

# Simple script to train the AI player networks.

#

# Networks are stored in the $net\_dir/nn, where nn is the generation of the networks (starting at 0).

# This script finds the last run, and reads the 'scores' file to find out which network in the

# generation performed the best.

#

# Once the best network has been identified, the generate\_network.py script is called to generate

# another set of networks a new generation directory. The highest scoring network is also copied

# to the new directory.

#

# After generating the nets, this script plays all the new nets (+ parent) and records their scores.

#

# Once the networks have been played, the script finds the highest score and generates net networks

# that are played, and so on.

#

# Variables:

# nets\_per\_base: determine how many networks should be generated for a given generation.

# top\_n\_nets: use the top\_n\_nets highest scoring nets as the basis for the next generation

# weigth\_jitter: Adjust any weight by a maximum +/- weight\_jitter.

# bias\_jitter: Adjust any bias by a max +/- bias\_jitter.

# mutation\_likelihood: The chance that any particular neuron gets mutated

#

# Use the top\_n\_nets as basis for the next set of nets

top\_n\_nets=1

# These many nets per base\_net

nets\_per\_base=10

weight\_jitter=0.01

bias\_jitter=0.01

mutation\_likelihood=0.4

# Where the neural nets are stored (or rather, the generational directories)

net\_dir="gameplay\_neural\_nets"

function find\_last\_gen()

{

# Find the last generation

last\_gen=$(ls -1 $net\_dir | egrep '[0-9]+' | sort -n | tail -n 1)

echo "$last\_gen"

}

function die()

{

# Print error message and exit

echo "FATAL: $\*" > /dev/stderr

exit 1

}

function generate\_new\_nets()

{

last\_gen=$1

next\_gen=$2

# Generate new networks

cur\_net\_idx=1

for net in `cat $net\_dir/$last\_gen/scores | sort -n -k 2 -t: -r | head -n $top\_n\_nets`; do

net\_file=`echo $net | cut -f1 -d:`

score=`echo $net | cut -f2 -d:`

mkdir -p $net\_dir/$next\_gen || die "Failed to mkdir"

echo "nets\_per\_base = $nets\_per\_base"

for i in `seq 1 $nets\_per\_base`; do

python3 generate\_network.py --input $net\_dir/$last\_gen/$net\_file \

--weight-max-jitter=${weight\_jitter} \

--bias-max-jitter=${bias\_jitter} \

--mutation-likelihood=${mutation\_likelihood} \

$net\_dir/$next\_gen/$cur\_net\_idx.nn

cur\_net\_idx=$(expr $cur\_net\_idx + 1)

done

# Also copy the current base net in, so that we always keep the best

# performing two nets for the next generation

cp $net\_dir/$last\_gen/$net\_file $net\_dir/$next\_gen/grand\_father\_$net\_file

done

}

function play\_one\_game()

{

net=$1

local net\_dir=$(dirname $net)

local net\_basename=$(basename $net)

python3 player.py oneshot $net | tee $net.play.out

# get score

score=$(tail -n 10 $net.play.out | awk ' /Moves to the right/ { print $NF } ')

echo $net\_basename:$score >> $net\_dir/scores

}

function play\_all\_nets()

{

next\_gen=$1

for n in $net\_dir/$next\_gen/\*.nn; do

echo "#####################"

echo "Playing neural net $n"

sleep 1

play\_one\_game $n

done

}

# Write settings to neural nets dir

(

echo "============================================"

echo "Starting run at `date`"

echo "Params:"

echo "top\_n\_nets=$top\_n\_nets"

echo "nets\_per\_base=$nets\_per\_base"

echo "weight\_jitter=$weight\_jitter"

echo "bias\_jitter=$bias\_jitter"

echo "mutation\_likelihood=$mutation\_likelihood"

echo

) >> $net\_dir/train-settings

done=0

while [[ $done -eq 0 ]]; do

last\_gen=$(find\_last\_gen)

next\_gen=$(expr $last\_gen + 1)

# Find scores in last\_gen, and find top\_n\_nets

generate\_new\_nets $last\_gen $next\_gen

echo "########################################################"

echo "last\_gen = $last\_gen"

echo "next\_gen = $next\_gen"

echo "Nets:"

ls -l $net\_dir/$next\_gen/

sleep 1

play\_all\_nets $next\_gen

done

## Appendix G: Automatic generation of training data with create\_training\_data.py

The create\_training\_data.py is used to generate one piece of training data for the TensorFlow object detection training. The script places an image of an object (Mario, an enemy, blocks etc) onto a background of a random colour. If the user prefers, the coloured background can be replaced by a screenshot from the game. The modified image is saved to disk, together with an XML file describing the bounding box and type for the inserted image.

#

# Little utility program that allows you to place an object (i.e. an

# enemy or mario etc) on a background in some location, and also generates

# the coordinate file that the training process needs.

import pygame

import click

import time

import random

import os

import datetime

import pathlib

NES\_WIDTH = 256

NES\_HEIGHT = 240

def get\_bg\_surface(background\_cols):

""" Generate a background surface as per the command line params """

# Create a surface to work with

bg\_surface = pygame.Surface((NES\_WIDTH, NES\_HEIGHT))

bg\_surface.convert()

# set background image as defined in the background-cols parameter

colours = background\_cols.split(',')

if colours[0] == "None":

# generate random numbers

r = random.randint(0, 255)

g = random.randint(0, 255)

b = random.randint(0, 255)

else:

r = int(colours[0])

g = int(colours[1])

b = int(colours[2])

bg\_surface.fill((r, g, b))

return bg\_surface

def get\_xy(object\_loc, img\_size):

""" Get an X/Y location for the object (generate one if not specified in object\_loc) """

(obj\_x, obj\_y) = (None, None)

if object\_loc is not None:

# specified on the command line

(obj\_x, obj\_y) = object\_loc.split(',')

obj\_x = int(obj\_x)

obj\_y = int(obj\_y)

# location wasn't specified

if obj\_x is None or obj\_y is None:

obj\_x = random.randint(0, NES\_WIDTH - img\_size[0])

obj\_y = random.randint(0, NES\_HEIGHT - img\_size[1])

return (obj\_x, obj\_y)

def get\_image(img\_path):

""" Load and return an image if the parameter is not None """

if img\_path is not None:

img = pygame.image.load(img\_path)

return (img, img.get\_rect().size)

else:

return (None, (0, 0))

def create\_path\_if\_not\_exist(path):

""" Create a directory "path", if it doesn't already exist """

pathlib.Path(path).mkdir(parents=True, exist\_ok=True)

def generate\_label\_file(screenshot\_name, label, object\_img, obj\_x, obj\_y, scale):

""" Generate the training annotation xml file.

Yes, this function should use the xml libraries.

FIXME: Use XML libraries instead.

FIXME: don't assume background image size

FIXME: Don't assume that file ending is the last three characters (?) """

basename = os.path.basename(screenshot\_name)

obj\_size = object\_img.get\_rect().size

xml\_filename = screenshot\_name[:-4] + ".xml"

annotation\_string = """

<annotation>

<folder>{}</folder>

<filename>{}</filename>

<path>{}</path>

<source>

<database>Unknown</database>

</source>

<size>

<width>{}</width>

<height>{}</height>

<depth>3</depth>

</size>

<segmented>0</segmented>

<object>

<name>{}</name>

<pose>Unspecified</pose>

<truncated>0</truncated>

<difficult>0</difficult>

<bndbox>

<xmin>{}</xmin>

<ymin>{}</ymin>

<xmax>{}</xmax>

<ymax>{}</ymax>

</bndbox>

</object>

</annotation>

""".format(label, basename, screenshot\_name, int(NES\_WIDTH \* scale), int(NES\_HEIGHT \* scale),

label, int(obj\_x \* scale), int(obj\_y \* scale), int((obj\_x + obj\_size[0]) \* scale),

int((obj\_y + obj\_size[1]) \* scale))

with open(xml\_filename, "w") as fn:

fn.write(annotation\_string)

print("Annotation string: {}".format(annotation\_string))

def generate\_screenshot(screen, bg\_surface, background\_img, object\_img, obj\_x=None, obj\_y=None,

outdir=".", label="None", scale=2.0):

screen.blit(bg\_surface, (0, 0))

if background\_img is not None:

screen.blit(background\_img, (0, 0))

screen.blit(object\_img, (obj\_x, obj\_y))

pygame.display.flip()

for event in pygame.event.get():

pass # we need to get pygame.event.get() in order to display

path = "{}/{}".format(outdir, label)

create\_path\_if\_not\_exist(path)

time\_now = datetime.datetime.now()

screenshot\_name = "{}/{}-{:04d}-{:02d}-{:02d}-{:02d}{:02d}{:02d}-{}.jpg".format(path, label,

time\_now.year,

time\_now.month,

time\_now.day,

time\_now.hour,

time\_now.minute,

time\_now.second,

random.randint(0,10000))

print("Screenshot name = {}".format(screenshot\_name))

print("path = {}".format(path))

(x\_size, y\_size) = screen.get\_rect().size

pygame.image.save(pygame.transform.scale(screen, (int(x\_size \* scale), int(y\_size \* scale))), "{}".format(screenshot\_name))

# add the xml label file

generate\_label\_file(screenshot\_name, label, object\_img, obj\_x, obj\_y, scale)

@click.command()

@click.argument('object')

@click.option('--background-img', help="Background image to use")

@click.option('--background-cols', help="Comma separated list of r,g,b (Example: 0,0,128). "

"If None, a random colour is generated", default="None")

@click.option('--object-loc', help="x,y coordinates of where to place object (Example: 50,80)")

@click.option('--many', help="How many images to generate", default=1)

@click.option('--outdir', help="Where to store the training data", default="training")

@click.option('--label', help="Label for the images", default="None")

@click.option('--sleeptime', help="delay between each image generation", default=None)

@click.option('--scale', help="amount to scale the images by", default=2.0)

def handle\_cmdline(object, background\_img, background\_cols, object\_loc, many, outdir,

label, sleeptime, scale):

""" A little utility program that lets you generate training data for the ai-player

object detection.

The basic idea is that you pass a background image and 'object' image, and the

program will then place the object on top of the background image in a random

location, and save the generated image together with the training mask.

"""

print("in handle\_cmdline: object = {}, background = {}".format(object, background\_img))

# initialise pygame first (sadly this is required...)

pygame.init()

screen = pygame.display.set\_mode((NES\_WIDTH, NES\_HEIGHT))

# load image if defined

(bg\_img, \_) = get\_image(background\_img)

# load the object image

print("Loading image")

(object\_img, img\_size) = get\_image(object)

for i in range(0, many):

# background surface - sits in here because we sometimes have random

# background colours

bg\_surface = get\_bg\_surface(background\_cols)

print("Generating image {}".format(i))

# find out where to place the object

(obj\_x, obj\_y) = get\_xy(object\_loc, img\_size)

generate\_screenshot(screen, bg\_surface, bg\_img, object\_img, obj\_x, obj\_y, outdir,

label, scale)

if sleeptime is not None:

time.sleep(float(sleeptime))

if \_\_name\_\_ == "\_\_main\_\_":

handle\_cmdline()

## Appendix H: Automating the automatic generation of training data images

The create\_training\_data.py script in Appendix G generates one piece of training data. The create-training-set-object-detection.sh in this appendix utilizes the create\_training\_data.py script to generate a large amount of training data in a small amount of time.

#

#!/bin/bash

#

# Wrapper to call the create\_training\_data.py script and generate a bunch

# of training data for the object detection models.

#

# Reads gifs from training-sprites/<type>-\*.gif, where type is e.g. goomba,

# mario-small, koopa-troopa etc

#

# Writes training data (images with the sprites pasted into) into a directory

# callex training.

generate\_image\_script="create\_training\_data.py"

outdir="training"

types="goomba koopa-troopa mario gameover flag"

cd $(dirname $0)

for t in $types; do

echo "type $t"

# figure out how many different images we have of a specific type

num\_files=$(echo training-sprites/${t}-\*.gif | wc -w)

how\_many=$(expr 50 / $num\_files)

echo "num\_files = $num\_files"

echo "how\_many = $how\_many"

# generate a bunch of objects on random coloured background

for f in training-sprites/${t}-\*.gif; do

echo "F = $f"

python3 $generate\_image\_script $f --many=$how\_many --outdir=training --label=$t

done

# generate a bunch of objects on a set of the empty background images

how\_many\_per\_background=10

for f in training-sprites/${t}-\*.gif; do

for bg in empty-images/empty-level\*.jpg; do

echo "f(empty image) = $f, bg = $bo"

python3 $generate\_image\_script $f --background-img=$bg --many=$how\_many\_per\_background --outdir=training --label=$t

done

done

done

## Appendix I: Script to split training into training data and test data

The TensorFlow object detection training process requires the data set to be split into training data and test data. The split-training-test-data.sh script splits the data into approximately 80% training data and 20% test data. The split ratio is approximate because the script uses integers for calculations.

#!/bin/bash

# Split a training, residing in $sourcedir, up into 80% training

# and 20% testing sets, and create the split set in $destdir.

#

# Note, moves the files

sourcedir=/home/henrik/files/project/ai-player/training

destdir=/home/henrik/files/project/models/research/data

train\_test\_split=8 # 8 10ths of data for testing

for d in $sourcedir/\*; do

dn=$(basename $d)

echo d = $d

count\_jpg=$(ls -1 $d/\*.jpg | wc -l)

count\_train=$(( $count\_jpg \* $train\_test\_split \* 10 / 100 ))

echo "count = $count\_jpg count\_train = $count\_train"

# move training data

for p in $(ls -1 $d/\*.jpg | head -n $count\_train); do

base=$(echo $p | sed -e 's/\.jpg$//')

if [[ ! -d $destdir/train/$dn ]]; then

mkdir -p $destdir/train/$dn

fi

mv $base.jpg $base.xml $destdir/train/$dn

done

# move test data

if [[ ! -d $destdir/test ]]; then

mkdir -p $destdir/test

fi

mv $d $destdir/test

done

## Appendix J: Bash Shell Wrapper for training TensorFlow

TensorFlow comes packaged with Python scripts for training object detection models, however they are tedious to run by hand. This project also includes a very small shell script, train-model.sh, to invoke the Python scripts with the correct parameters.

The shell script is reproduced below.

#!/bin/bash

export MODEL\_BASE=~/files/project/models/research

export MODEL\_DIR=data/mario-model

export PIPELINE=data/pipeline.config

cd $MODEL\_BASE

python3 object\_detection/model\_main.py --alsologtostderr \

--pipeline\_config\_path=$PIPELINE --model\_dir=$MODEL\_DIR

## Appendix K: Pipeline config for TensorFlow object detection training

The TensorFlow training for object detection requires a pipeline configuration script, which defines the location of training and test data, the number of classes, what modifications should be performed on the images, and so on.

This project slightly modified the standard pipeline configuration file for SSD mobilenet, but almost all the configuration below is originally from the pipeline configuration distributed in <http://download.tensorflow.org/models/object_detection/ssdlite_mobilenet_v2_coco_2018_05_09.tar.gz>

The pipeline configuration is included for completeness. **Modified** lines are highlighted in green.

model {

ssd {

num\_classes: 9

image\_resizer {

fixed\_shape\_resizer {

height: 300

width: 300

}

}

feature\_extractor {

type: "ssd\_mobilenet\_v2"

depth\_multiplier: 1.0

min\_depth: 16

conv\_hyperparams {

regularizer {

l2\_regularizer {

weight: 3.99999989895e-05

}

}

initializer {

truncated\_normal\_initializer {

mean: 0.0

stddev: 0.0299999993294

}

}

activation: RELU\_6

batch\_norm {

decay: 0.999700009823

center: true

scale: true

epsilon: 0.0010000000475

train: true

}

}

use\_depthwise: true

}

box\_coder {

faster\_rcnn\_box\_coder {

y\_scale: 10.0

x\_scale: 10.0

height\_scale: 5.0

width\_scale: 5.0

}

}

matcher {

argmax\_matcher {

matched\_threshold: 0.5

unmatched\_threshold: 0.5

ignore\_thresholds: false

negatives\_lower\_than\_unmatched: true

force\_match\_for\_each\_row: true

}

}

similarity\_calculator {

iou\_similarity {

}

}

box\_predictor {

convolutional\_box\_predictor {

conv\_hyperparams {

regularizer {

l2\_regularizer {

weight: 3.99999989895e-05

}

}

initializer {

truncated\_normal\_initializer {

mean: 0.0

stddev: 0.0299999993294

}

}

activation: RELU\_6

batch\_norm {

decay: 0.999700009823

center: true

scale: true

epsilon: 0.0010000000475

train: true

}

}

min\_depth: 0

max\_depth: 0

num\_layers\_before\_predictor: 0

use\_dropout: false

dropout\_keep\_probability: 0.800000011921

kernel\_size: 3

box\_code\_size: 4

apply\_sigmoid\_to\_scores: false

use\_depthwise: true

}

}

anchor\_generator {

ssd\_anchor\_generator {

num\_layers: 6

min\_scale: 0.20000000298

max\_scale: 0.949999988079

aspect\_ratios: 1.0

aspect\_ratios: 2.0

aspect\_ratios: 0.5

#aspect\_ratios: 3.0

#aspect\_ratios: 0.333299994469

}

}

post\_processing {

batch\_non\_max\_suppression {

score\_threshold: 0.300000011921

iou\_threshold: 0.600000023842

max\_detections\_per\_class: 100

max\_total\_detections: 100

}

score\_converter: SIGMOID

}

normalize\_loss\_by\_num\_matches: true

loss {

localization\_loss {

weighted\_smooth\_l1 {

}

}

classification\_loss {

weighted\_sigmoid {

}

}

hard\_example\_miner {

num\_hard\_examples: 3000

iou\_threshold: 0.990000009537

loss\_type: CLASSIFICATION

max\_negatives\_per\_positive: 3

min\_negatives\_per\_image: 3

}

classification\_weight: 1.0

localization\_weight: 1.0

}

}

}

train\_config {

batch\_size: 24

# data\_augmentation\_options {

# random\_horizontal\_flip {

# }

# }

data\_augmentation\_options {

ssd\_random\_crop {

}

}

optimizer {

rms\_prop\_optimizer {

learning\_rate {

exponential\_decay\_learning\_rate {

initial\_learning\_rate: 0.00400000018999

decay\_steps: 800720

decay\_factor: 0.949999988079

}

}

momentum\_optimizer\_value: 0.899999976158

decay: 0.899999976158

epsilon: 1.0

}

}

fine\_tune\_checkpoint: "object\_detection/ssdlite\_mobilenet\_v2\_coco\_2018\_05\_09/model.ckpt"

num\_steps: 200000

fine\_tune\_checkpoint\_type: "detection"

}

train\_input\_reader {

label\_map\_path: "data/object-detection.pbtxt"

tf\_record\_input\_reader {

input\_path: "data/train.tfrecord"

}

}

eval\_config {

num\_examples: 8000

max\_evals: 120

use\_moving\_averages: false

num\_visualizations: 100

}

eval\_input\_reader {

label\_map\_path: "data/object-detection.pbtxt"

shuffle: true

num\_readers: 1

tf\_record\_input\_reader {

input\_path: "data/test.tfrecord"

}

}