Universitat de les Illes Balears

FINAL DEGREE REPORT

TETRIS NEURAL NET PLAYING IN THE NINTENDO SWITCH

Joan Dot Sastre

Degree in informatics engineering

Escola Politècnica Superior

Academic Year 2020-21

TETRIS NEURAL NET PLAYING IN THE NINTENDO SWITCH

Joan Dot Sastre

Final Degree Report

Escola Politècnica Superior

Universitat de les Illes Balears

Academic Year 2020-21

Key words: AI, deep learning, deep q learning, tetris

Tutor: José María Buades Rubio & Gabriel Moyá Alcover

Thanks to José María Buades, for suggesting such a nice idea of a final degree project and helping me throughout all the steps whenever I needed. I would also like to show appreciation to Gabriel Moyá Alcover for his swift correction and betterment suggestions of this report.

Contents

Acronyms

AI Artificial Inteligence

API Application Programming Interface

MDP Markov Decision Process

NPC Non-Player Character

PC Personal Computer

SRS Standard Rotation System

List of Figures

List of Tables

Project summary

The basis of this project consists in achieving an Artificial Inteligence (AI) capable of playing the game “Tetris 99” in the console known as “Switch”, manufactured by the famous game company Nintendo. The task may seem simple at the beginning, but the sole nature of having to intercommunicate two devices (Personal Computer (PC) and console), with non-existent tools commercially available to send info from thePC to the Switch, already shows us that this will not be a trivial matter. The AI has to be built and trained through our own custom PC environment in order to make the training process faster, and then be able to receive and send information to the console reliably through our also deviced intercommunication system.

CHAPTER 1

INTRODUCTION

1.1 Artificial intelligence in video games

AI has been present in video games since the very beginning. Its purpose has always been to improve the players experience and the methods that have been used to implement such behaviours are vast, ranging from finite state machines and increasingly more complex enemy movement patterns tied to the game difficulty/level, to combining different advanced methods like pathfinding and decision trees. Other techniques related to machine learning such as reinforcement learning can currently also be found in some games. All these methods are mostly used for Non-Player Character (NPC)s and the information they perceive from the environment can be given in two different ways, via sensors, which provide a limited vision of the game world, or via the game's own stored information e.g., the player's exact location.

Due to an increasing interest in artificial intelligence in recent years, people have started to try and beat their favourite games with it. When taking this approach, we must first consider how the agent is going to perceive the game, having the same two options we talked about before. This time we usually encounter a major inconvenience, we do not have direct access to the game information due to us not being the game developers, although thanks to some Application Programming Interface (API)(such as OpenAI Gym) we can access the game and hence base our agent's information on it. Unfortunately, those APIs mostly feature older games, which limits us to the ones provided by it. Hence comes the need for image processing tools to extract data, though this may not necessarily be done by us, as will be shown later.

Due to the increasingly more difficult games being beaten, has also come a need for more intricate agents, leading to the drop of simpler techniques in favour of reinforcement learning (many times paired with those old techniques as a means to provide the agent with basic behavioural guidance). This has ended up providing much better results than previously achieved in highly complex environments, and also helped discover new strategies in the own game. Even exploits in the system have sometimes been found, like in the case of an OpenAi project, where in a hide and seek game, the agents managed to abuse the physics engine in various ways [14].

1.2 Objectives and setbacks

The overall goal of the project has already been discussed, but what will be called a success has not yet been defined.

Building an AI capable of playing Tetris has already been done many times before with great success, although the challenge trying to be taken on has a few more major and minor hindrances.

First of all, as a minor inconvenience, the Tetris version we are building our AI on features the Standard Rotation System (SRS), which is a modern rotation system with some unconventional situational rotations. No implementation that can be used has been found, neither as an OpenAI gym nor as simple game. Because of this, an entire game replicating Tetris 99 must be built from scratch to train our model.

Secondly, there is not a standardized way to access the console's controller port from a PC, so a reliable workaround must be built and adapted. This is probably the biggest setback.

Lastly, and as a result of having to intercommunicate both devices, some extra delay, that we hope will not heavily interfere, will occur when bringing everything together.

Having mentioned the setbacks we first encounter, we expect to build an agent that plays Tetris to near perfection, never loosing a game and trying to make as many points as possible in the least amount of time. If anything, we expect only the conditions outside the agents power to make it perform badly, namely bad screen detections or missed inputs. This means that once a good enough agent has been built, our main focus will shift onto making it perform as closely as possible to its intended actions on the console.

1.3 Task division

In order to reach our goal, we have to tackle the problems one by one. Thus, the means by which the results in the project have been obtained consist of dividing it into four different modules:

- Switch-PC interface: The way in which the PC is able to communicate with the console.
- Information capture: How the console's information is sent to the PC and then processed for use by the neural net.
- Deep learning: How the AI is able to learn. Includes the training environment explanation, the heuristic used and how it was chosen.
- Decision making: Defines how the information extracted by the information capture module is treated right before it is finally ready to be sent to the net. It also explains how the output is adapted and when it is transferred to the console.

A detailed diagram of our system can be seen in figure 1.1.

Also, the deep learning module can be seen further broken down in figure 1.3.

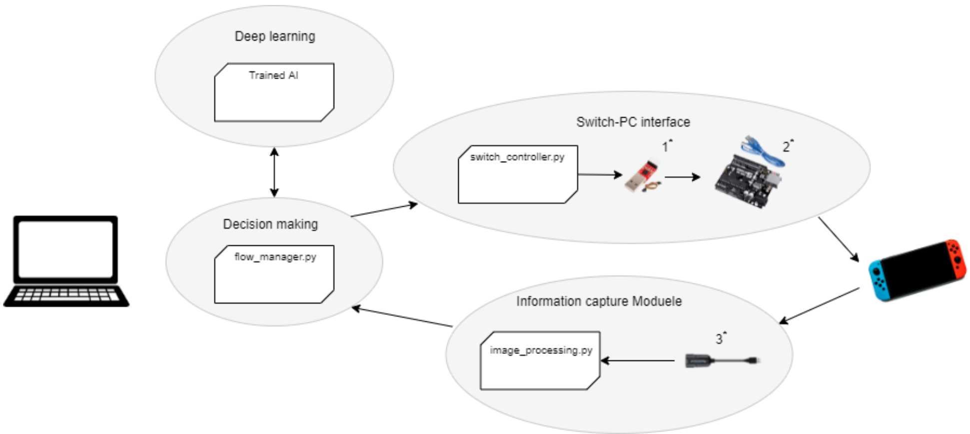

Figure 1.1: System structure

(a) 1* USB-TTL converter, seen in chapter 3

(b) 2* arduino, seen in chapter 3

(c) 3* Capture card from Switch to PC

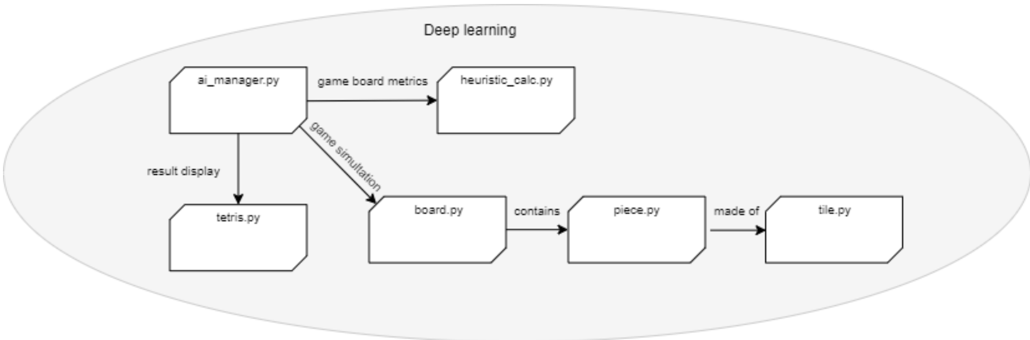

Figure 1.3: AI structure

The aforementioned modules will be further explained later, in their corresponding section in the document but, it needs to be mentions that the whole project has been made in python given its many machine learning library options.

Chapter 2

Tetris 99 and system built

2.1 Tetris version

Tetris is a long running game series that has been ongoing since 1984, when Alexey Pajitnov invented it. Ever since it was created, many iterations of the game have been made, with each one of those somewhat altering the rules or adding new mechanics to spice things up. As previously mentioned, the project is being made under the Tetris 99 version, which implements the SRS. This version has been chosen due to it being the most modern Tetris up to date and because of the challenge of having our AI work on another platform other than our own PC.

As many people already know, Tetris is a puzzle game consisting in trying to stack pieces up pieces and clear lines on a 10×20 grid. Whenever a line is filled to its maximum capacity it gets cleared and the blocks above it drop as many lines as were cleared. Whenever a piece is locked in place in an altitude higher than the game grid plus one you lose. A grid could look like figure 2.1.

2.2 Game basics

There is a total of 7 different pieces, each one of those having an associated colour that is usually maintained through all Tetris versions. Their names are I, J, L O, S, T and Z as seen in figure 2.2.

As we can see in the image just referenced, each piece has four different orientations which can be accessed sequentially back and forth in the order shown, the small circle indicating the axis the piece rotates in.

The I and O pieces are a special case considering they do not use an actual block as their anchor point to rotate, making the first one shift one block up or down depending on the current position and the second one not rotate at all.

Now that we know their shapes, we see that the maximum number of lines that can be cleared at once is four. This is crucial because the score we obtain does not increase

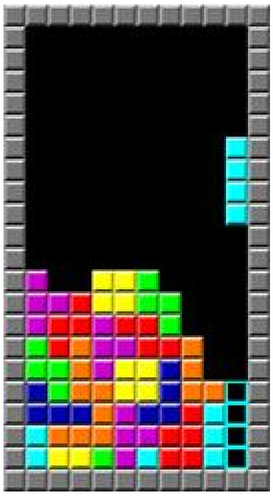

Figure 2.1: Tetris grid example. Figure extracted from [1]

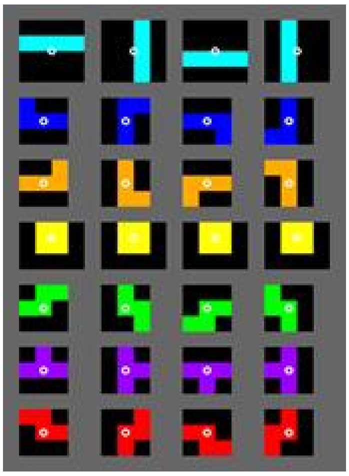

Figure 2.2: Tetris pieces and their rotations. Figure extracted from [2]

linearly with the number of lines cleared at once, it gives us higher scores the more lines we clear in one go following the formula in figure 2.1.

Single	100 × level difficulty
Double	300 × level difficulty
Triple	500 × level difficulty
Tetris (Quadruple)	800 × level difficulty

Table 2.1: Tetris score by lines cleared

More ways of obtaining points are soft drops (moving the piece down one cell), hard drops (letting the piece fall to the bottom) and combos (chaining line clears with different pieces), which go as 2.2

Finally, there is T-spins, which is a mechanic that will be spoken about at the end of this block, once all the information surrounding the SRS has been laid out.

Combo	50 × combo count × level
Soft drop	1 per cell
Hard drop	2 per cell

Table 2.2: Tetris score by movement and combos

The scoring system just described works on single player game modes like the 150 line marathon mode, which as the name implies, is completed by clearing said number of rows. The difficulty in this mode lies in that every 10 clears, the fall speed (or gravity) increases.

Tetris 99 also has another game mode which gives its name, it the involves 99 players concurrently battling against each other and here, line clears serve the purpose of sending “garbage lines” to the opponents. More on this system in the following section.

An actual game of Tetris 99 will look like fig2.3.

2.3 UI and specifics

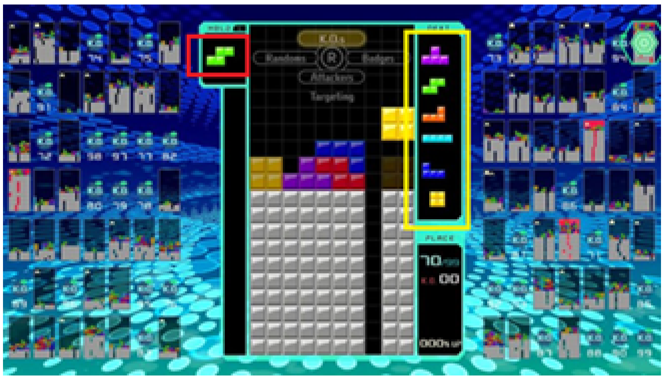

Figure 2.3: Actual Tetris 99 game. Figure extracted from [3]

The first thing that should be mentioned is that we can see the upcoming 6 pieces that will have to be placed on the board (highlighted by the yellow square). Those are chosen from a bag containing all seven pieces sorted randomly, which are extracted one by one without replacement. Therefore games will be much more predictable as luck will not be impacting the game as much as with a fully random selector.

Then, there is the piece storage block (encapsulated by the red square), which allows us to save the current piece and draw the next one or, if there already is a stored one, to swap it out.

Last and more importantly, the background shows many more smaller Tetris boards, which belong to other players who are competing against you. In this mode you cannot see your score, and your performance is based on surviving the longest. When clearing lines, you now send garbage lines (grey blocks) to whoever of those players you are targeting:

- Clear two lines: Send one line of garbage

- Clear three lines: Send two lines of garbage
- Clear four lines: Send four lines of garbage
- Clear the full board: +4 lines of garbage

Going back to the 99 player game mode, whenever you kill a player, a part of a badge is awarded to you. Each badge is increasingly more difficult to get, and you can only get up to four in total:

- Two knockouts: 25% garbage bonus
- Six knockouts: 50% garbage bonus
- 14 knockouts: 75% garbage bonus
- 30 knockouts: 100% garbage bonus

It may seem quite difficult to complete all badges, but the method is eased by being able to steal the badges from a player you have defeated. You can choose between five attacking modes to target different opponents:

- K.O.s: targets whoever is closer to losing the game.
- Randoms.
- Badges: targets whoever has more badges.
- Attackers: targets whoever is attacking you.
- Choice: manually select a specific player.

If you are targeted by multiple opponents, a boost to attack power is received:

- 2 Opponents: +1 Bonus lines sent
- 3 Opponents: +3 Bonus lines sent
- 4 Opponents: +5 Bonus lines sent
- 5 Opponents: +7 Bonus lines sent
- 6+ Opponents: +9 Bonus lines sent

This boost is applied before the badge attack boost.

It should also be noted that when receiving garbage lines, those will first be shown in the column right under your piece storage, and only be added to your board after some time. The time is indicated by 3 colour stages, being grey, yellow, and red, from best to worst. Garbage lines can also be cleared before they are added to your board by simply clearing lines.

2.4 SRS (Standard rotation system)

Now we can focus on the most intricate part of the game, the rotation system. The basics of this system have already been mentioned, however there is a much deeper pattern to it, which allows us to rotate pieces into places we would not normally be able to. These situations occur when a rotation that is not possible because a collision is detected, and the system tries to move the piece into four different offsets sequentially, sticking to whichever one works first. There are mainly two kinds of offsets, the ones that straight up ignore some collisions and allow you to rotate passing through blocks, and the ones that move you to another location. When an offset displaces you, it is known in game terms as a “kick”, and it should be noted that kicks can be performed against walls and pieces equally, propelling you in on or even two directions at the same time, even upwards. Because of the existence of upward kicks, a system limiting the number that can be performed had to be implemented to avoid infinite stalling.

As there is a very large variety of rotations and kicks that can be performed, only a few examples that represent most cases can be seen in figures 2.4, 2.5, 2.6, 2.7.

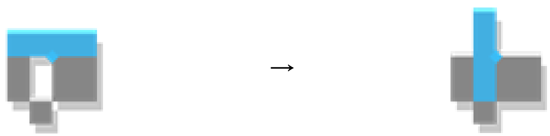

Figure 2.4: No kick phase. Figure extracted [4]

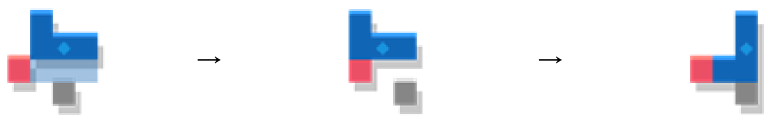

Figure 2.5: Right kick phase. Figure extracted from [4]

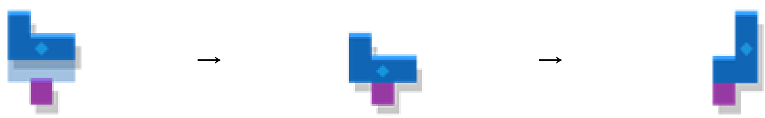

Figure 2.6: Up right kick phase. Figure extracted from [4]

Now that we have showed some examples, we can talk about T-spins. As its own name implies T-spins are performed using the T piece, and they happen whenever we manage to offset the piece into clearing 1, 2 or three lines, giving us 2, 4 and 6 garbage lines/800 × level, 1200 × level, 1600 × level respectively.

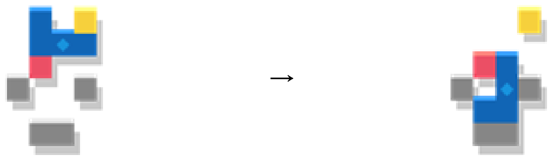

Figure 2.7: Down kick phase, from [4]

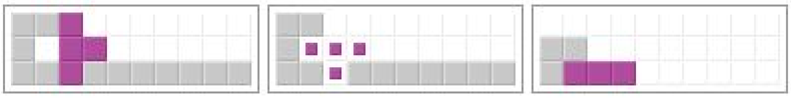

Figure 2.8: T-spin single. Figure extracted rom [5]

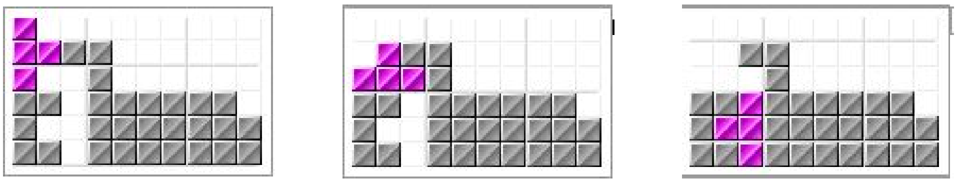

Figure 2.9: T-spin double. Figure extracted from [6]

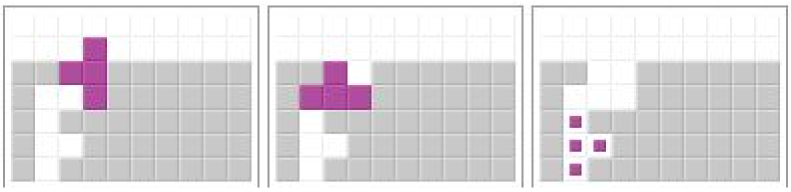

Figure 2.10: T-spin triple. Figure extracted from [7]

2.5 Tetris controller scheme

To play Tetris 99 we only need three main buttons plus the directional pad and the right joystick. The directional pad allows us to move the piece left and right, soft drop when pressing down and hard drop when pressing up. For the main buttons, we can press "A" to rotate right, "B" to rotate left and the right trigger, located at the top right part of the console, to store the piece. Finally, the left stick will only be useful when playing against other players by allowing us to target whoever we want to send our grey blocks to.The mentioned buttons can all be seen in figure 2.11

2.6 Our system

Having defined all important aspects of tetris 99, an environment similar enough had to be built in order for the AI not to find itself in a foreign situation when being tested in the Nintendo Switch. Thus a search for similar looking tetris implementations had to be done. Unfortunately, as mentioned when talking about the project setbacks, no complete candidates where found.

Figure 2.11: Switch

Thanks to the information found regarding how the SRS works and a project giving an example implementation in C# [15], we built from scratch a similar looking system that fulfilled our requirements. The system is constituted by three classes, board, piece and tile, each one being controlled and used by the one mentioned before it. Finally, the the script called "tetris.py" unites all of them together and creates a visual interface, made from a modified version of simpler Tetris game [16]. With this we can now play and show the net's output whenever requested.

To explain the following data structure a bottom up approach will be used in order to follow the thought process behind its implementation.

2.6.1 Tile

Tile is a class very short class containing a position in two dimensions (x,y) and a colour in rgb format. It also includes a method that given a center point and a direction, rotates its own coordinates once.

2.6.2 Piece

A piece object is constituted by "n" number of tiles depending on the piece type. There is a piece child class for each possible piece type and every one of them contains a list of each of the displacements or kicks that can be performed given a rotation. The method rotate piece loops through each tile calling their own rotate method using the center tile as the anchor.

Two extra methods that allow us to get a piece type from a number ranged from 0 to 6 and vice versa have been implemented for convenience and future use.

2.6.3 Board

Board is the biggest class of the three constituting the data structure. It is in charge of implementing all the Tetris 99 rules there are. Here there is also a piece rotating method, which is in charge of checking whether a kick can be performed or not, given it has the position of each of the blocks and the walls. Despite having implemented the t-spin moves a system that punctuates them has not been built, as it will not be used to reward the neural net (more information on that in the neural net section).

2.6.4 Tetris game

Finally, we have the "tetris.py" script, made using pygame. There is two different implementations within it which will be used to our convenience. The first one calls the main game loop and allows us to play the game normally, reading our keyboard input and calling the appropriate board methods to perform each one of them:

- ↓: "Soft drop", drops the piece down by one line.
- ↑: "Hard drop", drop the piece as far down as it can.
- ←: Moves left by one column if possible.
- →: Moves right by one column if possible.
- space bar: Stores the current piece if a piece was not just stored.
- "a": Rotates left.
- "d": Rotates right.

The second one is used by the neural net to show each of its moves. This means that the board game object is manipulated solely by the AI and the scrpit is only used to render the visual information of it by calling the applicable methods.

Either way, both can be stopped by exiting the window clicking the top right "X".

CHAPTER 3

SWITCH-PC INTERFACE

A key aspect to the project consists in how we connect the PC and the console. Getting the images from the game will be deal dealt with in the following chapter, but now comes the most important part. We need to somehow make the neural net's output get to the console.

3.1 Options and solution

As mentioned in the introduction, there is no way to control the Nintendo switch besides using its own controller. The first thing that came to mind was building a robot capable of pressing the buttons itself whenever we told it to. A rough way of implementing it was thought about and some actual piece candidates where found, but nothing came of it, as the difficulty of building such device could be a final project of its own.

Parallely, a way of faking the pc as a controller was investigated. A project that could record input onto an arduino and then send it to the switch by connecting it directly into the console's controller port was found, which told us that accessing the console was possible. This works by using the same checking protocol sequence as a Nintendo controller when trying to connect the device.

Thanks our discovery, we managed to modify the project so that instead of recording some input sequence, we could send the signal through the PC. As the project comes with a python script that contains the whole controller scheme and methods to test and ensure a good connection, we can plug the script in our project for its future use.

3.2 Controller script

For the console to be able to read the commands, it has to receive them one by one if they belong to the same category, meaning that we cannot send all directional inputs or rotations at once although we can combine the two of them. This is because the system

B	0x0000000000000002
A	0x0000000000000004
L	0x0000000000000010
PAD CENTER	0x0000000000000000
PAD U	0x0000000000010000
PAD R	0x0000000000020000
PAD D	0x0000000000040000
PAD L	0x0000000000080000
PAD UR	PAD U + PAD R
PAD DR	PAD D + PAD R
PAD UL	PAD U + PAD L
PAD DL	PAD D + PAD L
RSTICK CENTER	0x0000000000000000
RSTICK R	0x000FF00000000000
RSTICK UR	0x02DFF00000000000
RSTICK U	0x05AFF00000000000
RSTICK UL	0x087FF00000000000
RSTICK L	0x0B4FF00000000000
RSTICK DL	0x0E1FF00000000000
RSTICK D	0x10EFF00000000000
RSTICK DR	0x13BFF00000000000
PLUS	0x0000000000000200

Table 3.1: Switch input hexadecimal sequences

works by reading a bit sequence that gets updated by us whenever we need to, it does not store a queue of requested commands.

In table 3.1, we see a list of all the hexadecimal values of each of the buttons we will use (PLUS or Pause is added for testings future convenience). As we can see, pad buttons, joystick buttons and the main buttons refer to two different hexadecimal spaces so both of them could be sent at the same time by adding the two sequences as done for diagonal directions.

3.3 Arduino tools

The original project only used the arduino seen in image 3.1, which already had the commands imprinted in a loop. This time around, as it is connected to our PC using a USB-TTL converter (A.4), seen in 3.2, the loop waits for the input and sends it to the console.

The USB-TTL converter must be connected according to the information given by the own device, which means that contrary to what is most common, RX to TX and TX to RX, it is done backwards. This is only because the converter tells us where the connection must go and not what it actually is. As for the ground and 5V cables there is no room for doubt.

Figure 3.1: Arduino ELEGOO UNO R3. Figure extracted from [8]

Figure 3.2: USB to TTL CP2102. Figure extracted from [8]

Chapter 4

Information capture

If we want to play on the Switch, a system that allows us to decrypt what is on screen, to feed it to the neural net is needed. Our first approach was to use some sort of video device, like a webcam, to record what was on screen while simultaneously sending that information to the PC. That idea was quickly scrapped, as using a normal capture card was the obvious easiest go to. Thanks to that, we can use Opencv (see more in A.1) to get the images directly in real time with minimal delay in order to perform whichever operations we need to. OpenCV was chosen as our image processing tool mainly because of its vast number of examples and information regarding it.

4.1 Detection

As previously mentioned, a game of Tetris 99 looks like the image in 4.1:

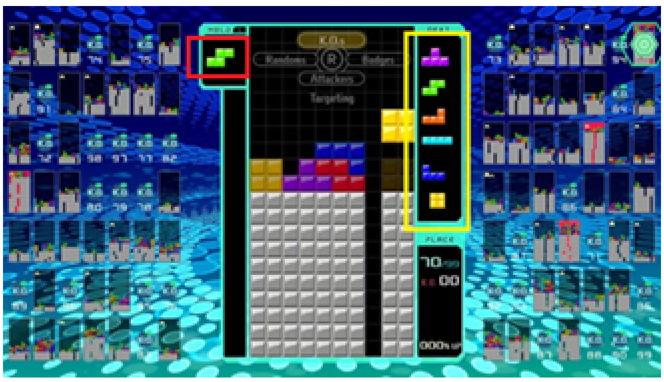

Figure 4.1: Highlighted detection areas. Figure extracted from [2]

Owing to the fact that the most important information we need is what are the pieces currently in play, a method to detect whether there is one and which type it is had to be devised. At first it was thought this could be guessed by the piece's own shape but, given rotations and having access to the colour channel, a detection by the latter method was much faster and easier to do. Hence, came the idea of calculating the mean colour of each shape in a 5×5 matrix from its center point, including empty blocks and grey pieces. Still, as seen in the previous image, when a piece is placed, it turns a darker tone of its former colour, making us have to factor those in. Thanks to this minor inconvenience, we will later be able to distinguish the main piece from an already placed one if we need to do so. Finally, a small leeway to the mean colour of each piece had to be taken into account when checking for matches due to other elements in the game board influencing the colour of the own piece with shines or shades.

The information we get is also presented back to the user in real time by drawing the conclusions on top of the processed frame. This helps us understand what is being detected and therefore being fed to the neural net. Depending on the detection a string matching the detection will be shown:

- "e": Means empty block.
- "S", "Z", "I", "T", "J", "L", "O": Refers to each of the possible pieces found in a tetris game.
- "gr": Means grey block.

There is one more element that will not be shown as a letter, "No match", which will be displayed using the last letter found in the block in white, otherwise their respective piece colour or black for the empty will be shown.

4.1.1 Game grid

The first element we try to detect is the game grid. It can be done thanks to having manually found where the cells center pixel is, and how wide and tall each cell is. By applying a for loop, we can then iterate through each cell and store the information in two arrays with the size of the game grid (10×20). The first array contains 0's for empty cells, 1's for blocks and 2's for the main piece (game_matrix), while the second one contains information related to the colour, including a “no match” variable (info_matrix). On each iteration, game_matrix block will be updated only if a match was found, else it will assume the board's state has not changed.

4.1.2 Out of the grid

The next element we detect is a line upwards out of the main game matrix (row 21). This must be done due to the main piece spawn position going up when it cannot be placed at a certain height, the maximum being 21. This was done separately due to the background colour not being black and because only the main piece is displayed at that height, with placed blocks being hidden by the borders 4.2, 4.3.

4.1.3 Check stored piece

To detect which piece (or if none) is in storage, a system that casts two rows of three detection points was built. Each one of those points corresponds to a possible place of

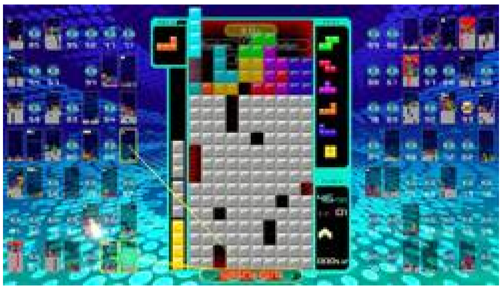

Figure 4.2: Main piece out of grid. Figure extracted from [9]

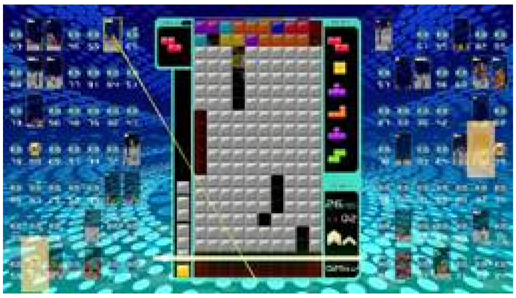

Figure 4.3: Blocks out of grid. Figure extracted from [10]

a block and, in case it is filled, it updates two matrices with the same system that was built for the game grid detection.

4.1.4 Check upcoming pieces

In order to know what is coming up next, we reused the system to detect stored pieces, this time iterating through a for loop once for each of the upcoming six pieces. Each of the matrices detected is then stored in an array in the same order they are detected (top to bottom).

4.2 How noise affects detection

As mentioned before, we cannot differentiate piece colours without adding a small margin of error for the detection to be consistent in the majority of cases. Unfortunately not only are the colours influenced by other elements, but also visual effects spawn all across the game board depending on the actions done.

To begin with, there is an effect that tells us where our piece is going to be placed (see figure 4.4). Luckily this feature was found to be able to be turned off in the game options, although it is the only one that can be filtered out this easily.

As for the other effects, things like red screen borders, arrows pointing other players and glitter when dropping a piece or sending grey blocks to opponents can also be found interfering with detections (some also seen in 4.4). Many of those directly block what is behind them, so no image processing or margin can be set to minimize or eliminate the obstruction. The best solution we came across is not updating the board information whenever a foreign object is detected, which ends up working pretty well as many of the effects disappear pretty quickly.

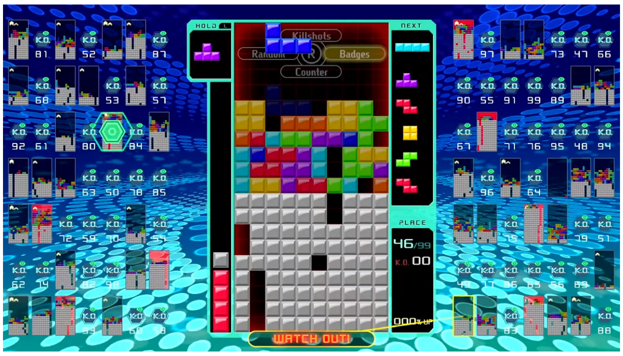

Figure 4.4: Game effects. Figure extracted from [11]

4.3 Information adaptation for the neural net

Once all the information needed has been collected, a way of adapting it so that the neural net understands it had to be made. To do so, we had to convert the data into a Board object of our own Tetris game, with the same configuration detected.

As our neural net works by finding the best combination of consecutive rotations and horizontal movement, and then dropping the piece, we only need to ask for the neural net's response once (when the piece just spawned). This way we only try to detect new information when the last sequence of movements is completed and when a new piece is detected at the top five rows of the grid, which are the only rows a piece can spawn in plus two more, in case there is a late detection.

Only when a new main piece is detected, which as mentioned before can be done thanks to it being able to be told apart from the rest because of being slightly brighter, will we construct a new Board object. This is to avoid repeating the creation of the same object over and over again. With what we have just detected, a Piece object with its type and position can now be created. The type can easily be guessed by just checking the first blocks's colour but the position is trickier. As all pieces always spawn in the same exact rotation and x position, we can always assign it to be the same, "4", as for the height, we it will be the one of the first block found -1, given that each piece's center block is always one row down except for the "I" piece.

We can now proceed to the creation of the game grid that will be added to the Board object. It is quite simple to do so because we have a 10×20 array stored with information regarding each cell. We now only have to filter out the main piece, add four empty rows at the top and reverse it to match the object's model. The actual colour of the placed blocks does not matter, as it is merely an aesthetic element.

Finally, we check if there is a stored piece. If positive, we must then check its colour to know if it is an option for it to be placed or not. That information is then passed on to the Board object.

CHAPTER 5

DEEP LEARNING MODULE

A crucial aspect of the project is whether we achieve a neural net capable of doing as many Tetris (four line clears) as possible, while never losing. For this, many routes could be taken in order to obtain good results.

5.1 Possible approaches

When trying to build an AI capable of performing a task we have many options to consider. However the nature of the task itself can give us a few hints as to where we may want to head our way. With this in mind, we already know that we are going to use some kind of deep learning A.5 technique.

5.1.1 AI with previous training data

We may first think that training an AI with the records of the best Tetris games ever might be a good idea given that many times an AI learns by finding common "rules" that might lead to a successful result within certain conditions. In this case, as we want to build a superhuman agent, when trying to make it through already explored routes we are narrowing its possibilities down to at most peak human capabilities.

Not only is the aforementioned point an issue, but also when going down this route we encounter another key obstacle, obtaining the dataset. This task is on its own another whole beast of a problem, considering that Tetris comes in many formats. We would have to manually decide which games are fit to be fed to the neural net, and also find a way to extract knowledge from each of those so as to make the data chewable. In case we only wanted to use Tetris 99 games to feed our neural net, we could somehow reuse our already built image detection system explained in chapter 4, and expand it. But then again, as noise in the images is a problem, many more negative things could come out of this.

To sum up, not only would we have to build or expand our whole image detection system, aside from building the agent, but the results would perform, at most, as good as the best human players.

5.1.2 AI without previous training data

Casting aside the last option, two other methods that do not require us to have a database with previously played games emerge, reinforcement learning (specifically Q-learning) (A.6) and genetic algorithms (A.7). Both being completely viable options, being that some Tetris AIs have already been built using these methods, means that either of them would have probably been a good option. But, as the majority of the information regarding Tetris agents found is related to the former, our first and final option was decided to be reinforcement learning, as it meant an easier path given our goal.

5.2 Our system

Having specified the technique we are using, we now have a vast range of options to choose from to feed the agent. There are four main points we will have to take care of: training method, reward function, game state representation and neural net layers.

To get to the point where we now have a very good agent, we have had to explore many options that will now be detailed.

It is worth bearing in mind that moving the pieces is considered a trivial matter in order to get faster results. Hence, the neural net only evaluates each game state and is not bothered with making random moves till it finds a way to fit them to the expected final state.

5.2.1 Nuno Faria AI

[17] Nuno-faria/tetris-ai is a GitHub repository that implements a possible solution for a tetris q-learning agent using keras (A.3) with tensorflow (A.2). The result this person got is an agent capable of clearing a very high number of lines, although not performing many tetris clears.

So as to get all future states, a function that explores all possibilities is used. The method tries all left and right moves combined with each piece's possible rotations (discarding repeated configurations e.g., "I" piece only needs to be rotated once), and then dropping the piece.

The previous way of exploring states also benefits us because it will make synchronizing the AI with the console much easier, as we will only need to ask for a sequence of movements once at the beginning of each turn instead of having to do so after each move.

The four main points mentioned before look like the following:

- Training method: The agent is trained for two thousand games where the value epsilon starts at 1, meaning all actions are random, decreasing linearly towards game one thousand five hundred, at which point all actions will be the nets output. During the training, each state will be stored to memory with its reward and, only when the memory buffer is full the training begins, performing a training cycle for each game or loop iteration.

- Reward function: The reward function looked like the following: $score = 1 + (lines\_cleared^2) \times Tetris.BOARD\_WIDTH$ and in case the agent lost the game it subtracted 2. The theory behind this is to incentivize the agent to keep on surviving, the reason being is that the agent adds 1 to the score even when no there are not any line clears and it penalises a lost game. As we also want encourage it to clear more than a line at a time, the number of lines is squared and then multiplied by the number of blocks cleared from the game grid.

- State representation: The information the net will receive is related to a few parameters obtained from the board game in each of the possible moves. Those are:

  - Lines: Meaning the number of lines cleared for.
  - Holes: Total number of holes in the board. Meaning empty cells with a block on them.
  - Total bumpiness: Being the sum of the heights differences between columns
  - Height sum: Height sum of all the columns.

  Others like max height, min height, max bumpiness, next piece and current piece where explored but to no avail apparently.

- Neurons: A [32, 32] chain of dense layers, meaning there are 4 total layers: input layer with 4 neurons (each related to one of the representation parameters), the two 32 neuron layers that will be modifying their values, and finally the output layer.

This project comes with its own Tetris implementation, unfortunately, it is flawed in a few aspects, meaning that piece spawn positions do not match those of Tetris 99 and more importantly it does not feature the SRS, here our custom environment comes into play. A whole new script mimicking the one Nuno Faria used to calculate the states was built to fit our environment, including an action for storing a piece, which was not part of the original project.

After having our initial contact with reinforcement learning and getting an initial version of what our AI could look like, which played similar to the one the original project had, we still needed it to be much better.

Eltetris

Here an improved version of Pierre Dellacherie's Algorithm [18] was tested combined with what we already had. The difference came only with the state representation, where now instead of four metrics we had nine:

- Lines: Same as before.

- Holes: Same as before.

- Landing height: Height where the piece is placed plus half of the pieces height.
- Row transitions: Number of times an empty cell is adjacent to a filled cell on the same row and vice versa.
- Column transition:Number of times an empty cell is adjacent to a filled cell on the same column and vice versa.
- Cumulative wells: Sum of the depth of all wells. A well being consecutive empty cells within the same column with adjacent blocks to the sides.
- Eroded piece cells: This is the number of rows cleared multiplied by the number of blocks eliminated from the placed piece
- Height sum: Same as before.
- Stored piece type: Type of the piece in storage.

This also meant that our input layer now had 9 neurons instead of 4. In testing this new configuration, we did not manage to get the neural net to converge. For some reason it learned how to stack pieces quite well but it rarely chose to perform line clears. After much investigation, we could not pinpoint the reason for the mentioned behavior but other alternatives where found.

5.2.2 Zeroize318 AI

[19] Here is where our AI improved by leaps and bounds. Even though this project did feature a kind of SRS, some of the spins where not implemented and, additionally, no bag system was used. In order to train the wanted agent as closely as possible to Tetris 99, we once again resorted to using our custom environment. This meant scrapping the whole game and redoing the calculations of the board all over again but this time on our environment. Before doing this, the project was tested thoroughly to ensure it worked properly.

Here the board states are separated into two inputs sequences. First is the board configuration, which is basically the game grid (10×20) represented as a bool matrix with true and false meaning if the cell is filled or not. With this, we then create two separate 2d convolutional layers the first one with 64 layers and a size 6 kernel, and the second one with 256 layers and a size 4 kernel. Then for each of them we send the output to two other different layers, one for max pooling and one for average pooling both with the same pool size. For the first 2d convolutional layer, the pool size is 15×5 and for the second one it is 17×7. The output is finally sent to next layer together with the second input that we will subsequently explain. Separating the input like we just did may help learn game features at different scales, otherwise, the neural net can on its own deem one of the two to be unimportant and value it much lower.

The second input looks much closer to the old models we used and its main purpose is to aid the neural net with extra information. This information is comprised of:

- Height Sum: Same as before.
- Total hole depth: Depth of the highest buried hole in each column sum.

- Can store: If a storing move can be performed.
- Type of the held piece: Stored piece.
- Type of the current piece: Piece in play.
- Upcoming pieces: Pieces that are coming next in order.

Each of the piece related metrics is and array of 0's with length seven, for the total number of piece types in Tetris, with a 1 in the position corresponding to its type. If there is not a held piece the whole array contains 0's. The neural net looks like fig 5.1.

Once the inputs are prepared, they are finally connected to two dense neuron layers of sizes [128, 64].

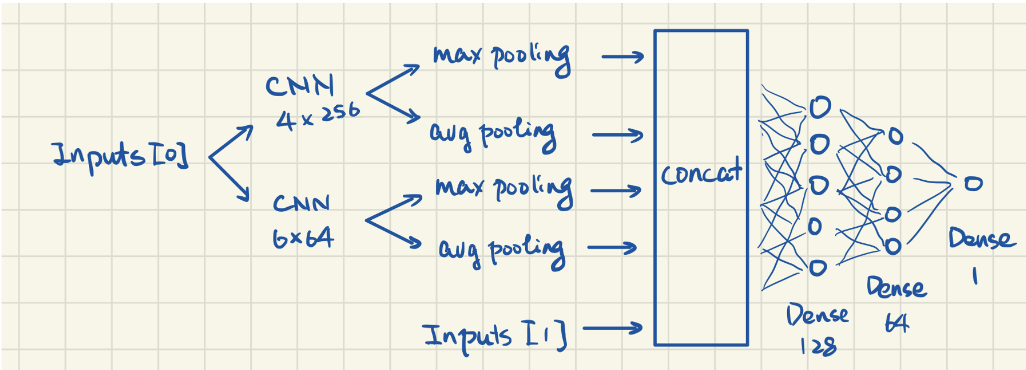

Figure 5.1: Built Model. Figure extracted from [12]

Respecting the learning system, here the task is separated into twenty outer cycles, where an x amount of processes, depending on the number of virtual core the computer has, are spawned to play games until they either fill a memory buffer or play a thousand times, also limiting each game to a maximum of two thousand steps. Each process has a 6% chance of performing a random move except one of them, which will serve us as a reference to see how agent is doing. Once all the processes have finished, the information is dumped in a file for later processing or future usage if we wanted to. Next come five inner training cycles, where the information in the file is sorted randomly and separated into equal parts (if possible) to fit the model, effectively performing mini batch without replacement. After doing this twenty times we are done with the training.

Finally, we have an agent capable of clearing lines four at a time instead of just surviving.

5.3 Experimentation

5.3.1 Parallelization tests

As explained before, the model we used separated sample collection between n number of processes depending on the number of virtual cores the computer has. When doing the first test on our main computer, a laptop with two physical cores and four virtual ones, good results were achieved after about eight outer loops. In order to improve the time we got to those, a different computer with higher CPU and number of cores was used. It contained 6 physical cores and twelve virtual ones but for some reason the net seemed to converge much slower. After some testing and forcing the use of different numbers of processes, it was found that the optimal number for convergence speed was four. This occurs due to only only having one process always fully playing with the net output, which means that depending on the number of processes, the proportion of its effect was bigger or smaller. By replicating the same proportions independently of the number of processors the convergence speed could be maintained stable.

CHAPTER 6

DECISION MAKING

Here we will explain how the main script works i.e., the menu, and the coordination between the image processing module, the neural net and the switch input controller. Also how and when the frames are processed will be mentioned.

6.1 Menu

When executing our main file, we have to specify which COM port we want to connect to (COM5 in our case), then a menu will pop up with four different options:

- Play Tetris: Redirects us to our game implementation model for us to play.
- Train net: Goes to the training module for a new model to be trained with the current parameters in the script.
- Test net: Goes to the training module and loads the specified model. It then starts playing a game until we leave the window or it finishes. The game is shown using the "tetris.py" script we described at the end of subsection 2.6.4.
- Access Switch: We go to the script that will handle the communication between the pc and the Nintendo Switch. This will be detailed in section 6.3

6.2 Frame management

A modified separated module called FileVideoStream [20], that takes care of handling the frames was used. It basically creates a thread that periodically puts the frames received into a non-blocking queue for our main thread to read. This way we can keep executing our code without blocking in case we get ahead of the input. Each time the main process gets a new frame, it sends it to the image processing module.

Figure 6.1: Main menu. Inspired from project [13]

6.3 Flow control

When the script is called, all the resources are immediately loaded. We first connect to the serial port COM5 and try to synchronize with the console, then we load our neural net, and finally we access the capture card. If any of those do not work we exit the program back to the menu.

Once the everything is up and running, we enter the main loop, which will begin displating the images from the console on our computer screen. The commands will be the exact same as when playing our custom Tetris, which means that "a" is the B button on a switch controller and "d" is the A button. Additionally the pause button is letter "p" and the "Esc" key will exit us back to the menu. This way we can play with the console using the keyboard if we wanted to. Finally, when pressing enter the auto play mode will begin. The manual command system is built thanks to Opencv's way of reading user input without blocking the program (cv2.waitKeyEx(1)) and the Switch controller script in conjunction. Whenever we detect any of the mentioned keys, they are sent to a custom method that transforms our input into something the Switch controller script will understand.

When auto play is active, a method called "flow_manager" will be called. The method starts by trying to ask the neural net what the next motion should be based on what was detected on the last frame. This action will only be done if we detect that the information of the last board object and the new one, specifically the bag or the stored pieces, is different as this means that a new piece has spawned. When we get the information regarding the next move, we display it on the game board by drawing the expected position of each piece tile with its respective colour and letter (e.g., "I" in blue), which will help us determine if we then proceeded to place it properly.

In order to make the controls go one by one, a timer method has been implemented. The timer allows us to send a command, wait for the amount of time, then send a clear request, which empties the bit sequence, and then we can proceed to wait again and send the next command. Through trial an error, the smallest time that has been estimated to work reliably is 0.027s. Here, the same method to communicate with the

Switch controller script is used.

CHAPTER 7

RESULTS

Having all of our systems up and running, we can now proceed to test our system capabilities.

7.1 150 line mode

The test have been run in the 150 lines mode, which as the name implies, is completed by clearing said number of rows. The difficulty in this mode lies in that every 10 clears, the fall speed (or gravity) increases.

After 15 games, we achieved the scores seen in figure 7.1 ordered by line count.

120 lines	133.952 points	4:00.23 minutes
121 lines	132.712 points	4:02.55 minutes
123 lines	158.888 points	4:06.38 minutes
124 lines	139.645 points	4:10.34 minutes
125 lines	137.822 points	4:20.48 minutes
125 lines	147.321 points	4:28.20 minutes
128 lines	158.462 points	5:38.28 minutes
129 lines	158.294 points	4:45.12 minutes
131 lines	157.557 points	4:35.51 minutes
131 lines	156.931 points	4:28.68 minutes
131 lines	154.874 points	4:58.12 minutes
134 lines	158.851 points	5:09.33 minutes
137 lines	162.612 points	5:11.76 minutes
140 lines	179.106 points	5:15.01 minutes
146 lines	190.502 points	5:37.71 minutes

Table 7.1: Results from 150 lines mode

Score statistics:

Mean	155168.6
Median	157557
Range	57790
Minimum	132712
Maximum	190502

Table 7.2: Score statistics from 150 lines mode

Mean x	129.66666666667
Median x	129
Mode	131
Range	26
Minimum	120
Maximum	146

Table 7.3: Line statistics from 150 lines mode

As the time is related to the number of lines cleared and the score, calculating the mean does not tell us much.

After seeing the results obtained and the system play, even though the agent gives us almost flawless configurations, the command execution is too big of a barrier to overcome. As we have seen in footage and also tested ourselves, to complete the 150 lines mode it is crucial to rely on the timer that allows us to however over the bottom of the grid for a few seconds or even spin the piece to place it where we want. As the agent inputs the simplest available to get to the position without taking gravity into account, it can collide with other pieces effectively breaking the input chain. The small variability seen in the statistics comes from the agent failing always under high gravity around the same number of line clears.

Even though what we have just said, the time under it reaches said line counts and the scores it gets are better than those of common players, which means that if we could execute more accurate moves we could even retrain our AI to perform T-spins and probably even beat top players. Still, the task at hand would be another whole project probably far too taxing for our current reach.

The results obtained are based on the final version after having tried the variations explained in the next section.

7.1.1 Improvement tests

At first, our objective was for the AI to achieve as many points as possible but, seeing that it could not survive until the when testing it on the actual playing field and realizing that the main problem was execution time, some experiments with intentions to better its survival chance at higher fall speeds have been carried out.

7.1.2 Single clears

The first idea was to train a second neural net that instead of piling rows to clear them at once, only tried to do so one by one. It was meant to work in conjunction to the first agent, using the former up to a certain number of line clears with low fall speed, and the latter once things got difficult. The reasoning behind this was that as rows get higher, the input window gets much tighter, thus, in keeping lines low the agent would fare much better. Unfortunately, this was no the case as the time margin given was still not large enough.

7.1.3 L or R rotation

Due to the following two phenomenons, a situation that might disfavour our agent also happens:

- Up to this moment, our AI only tries to rotate pieces left. This is because rotating once right and rotating three times left will usually give us the same configuration.
- Due to Tetris boards having a width of ten blocks, pieces with and odd number of blocks cannot be placed right in the middle hence placing them one block left is favored.

Pairing the initial piece placement with a left rotation means that on averages, left placements will be performed at a much faster rate so, another idea was to retrain the agent to make it rotate right. Unfortunately, the results did not change much.

7.1.4 Avoiding fake input

As we mentioned, we sometimes have sprites and effects covering what we actually want to detect, specially when we drop a piece, which ends up sometimes covering the dropped piece before we can update its information.

In order to minimize the occurrence of this mishappenings, we tried to play the game game with the output game board of the neural net as our next input. Still, if the controller made a wrong move, meaning that a piece could not be placed where it should, we ended up with a discrepancy between our board and the actual one, which cascaded us our way into a lost game. To fix this we tried perform and addition check onto the Switch game board when going to ask for the net input. When matching the next expected board and the detected one, if we differed by a total of more than one block, we stuck with the Switch board, otherwise, it probably meant that a cell got blocked by some effect instead of there being a missed input, so the expected board was chosen.

7.1.5 Combining inputs

To try and lower the input time, we tried sending rotation and displacement commands at the same time, as they occupy different but spaces. Doing so resulted in much faster movements sequences but it also meant many more dropped inputs, making the overall result much worse.

7.2 99 players mode

When testing this mode, we tried it in three different difficulties, 1 3 and 5, them being the easiest, medium, and hardest.

Through some experimentation, we saw that the number of effects was far too great for the AI to overcome, getting pretty low scores on average. As in the previous mode, we ran pretty much the same test but here we discovered something quite interesting. When playing always using the neural net output board as the next state, assuming that we always made a good move, we managed to perform really well and even win come games. This phenomenon is caused because this way, if there is not a single wrong input, we can totally ignore screen effects. In figures 7.4, 7.5, 7.6, 7.7, 7.8 and 7.9 we can see the results obtained.

1
1
1
99
85
30
1
1
96
1
26
1
75
23
95

Table 7.4: Position results from 99 players easy mode

Mean x	35.733
Median x	23
Mode	1
Range	98
Minimum	1
Maximum	99

Table 7.5: Position statistics from 99 players easy mode

Here we can see that our agent, when executing moves properly, can beat easy AIs pretty, sometimes beat normal ones and even score quite high against the hardest ones pretty reliably. When obtaining very bad positions like last place or close to that, it is observed to happen when there is a missed input, which as we do more tests and the compute starts slowing down occur more frequently. Due to there being some really bad games, variability is very high, skewing the results.

45
5
26
23
28
1
6
89
47
96
4
36
6
24
77

Table 7.6: Position results from 99 players normal mode

Mean x	34.2
Median x	26
Mode	6
Range	95
Minimum	1
Maximum	96

Table 7.7: Position statistics from 99 players normal mode

57
69
80
26
43
56
31
57
35
42
44
97
44
36
76

Table 7.8: Position results from 99 players hard mode

Mean x	52.867
Median x	44
Mode	57.44
Range	71
Minimum	26
Maximum	97

Table 7.9: Position statistics from 99 players hard mode

7.3 Final thoughts

After all the trials and errors, having made a very good AI and a system that can almost follow its pace, specially on an outside platform, we can call the experiment a total success.

Having reached this stage means that all our efforts put into so many different problems that constitute it have come to fruition. Great effort has been put into researching into the four main blocks composing the experiment, which very much differ from each other. Ranging from implementing a whole Tetris game, to learning about deep reinforcement learning, arduino and image processing, and using them all accordingly. Implementing, modifying and synchronizing all the modules has been a great ordeal from which much knowledge has been extracted and that will hopefully be but to test again.

APPENDIX A

Appendix

A.1 Opencv

"OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products". [21]

A.2 Tensorflow

Tensorflow is an opensource machine learning and artificial intelligence library focused on training and inference of deep neural networks. [22]

A.3 Keras

Keras is an open-source software library that provides a Python interface to build artificial neural networks. Keras acts as an interface for the TensorFlow library, allowing us to work at a higher abstraction level by controlling the neural net at layer lever instead of at neuron level. [23]

A.4 USB to TTl

A USB-TTL converter is a device that mediates between USB signalling and a simple serial bit stream, using TTL voltage levels and asynchronous communication (ASCII). With the proper drivers and a USB-TTL converter installed, a computer can use any sort of “tty” comm protocol or app to talk to an arbitrary serial device (e,g, an Arduino). [24]

A.5 Deep Learning

Deep learning consists in a set of algorithms that mimic the structure of a brain by having layers of logical units called neurons, that learn through experimentation or analysis of training data.

A.6 Q-Learning

To explain what q-learning is, we first have to define what a Markov decision process (Markov Decision Process (MDP)) is. An MDP is a way to model decision making in discrete, stochastic, sequential environments which change state randomly in response to action chosen by the decision maker. The state of the environment affects the immediate reward obtained by the agent, as well as the probabilities of future state transitions. The objective is for the agent to select actions to maximize a long-term measure of total reward. [25]

Q-Learning takes this one step further by aiding us find a suitable policy to satisfy long term reward through exploration by taking some random actions.

A.7 Genetic Algorithms

Genetic Algorithms are deep learning algorithms inspired by the theory of evolution by Darwin. Said algorithm works by starting with a population of agents which through various iterations constituted by performing the wanted task, evaluating performance, selecting the fittest and mixing the genes (including random mutations), nets us, hopefully, the desired agent.

Bibliography

[1] Wikipedia, the free encyclopedia, "Tetris game board." [Online]. Available: https://an.wikipedia.org/wiki/Imachen:Typical_Tetris_Game.svgs (document), 2.1

[2] Tetris Wiki, "Tetris game board." [Online]. Available: https://tetris.fandom.com/wiki/SRS (document), 2.2, 4.1

[3] S. Mika, "Tetris game board example." [Online]. Available: https://pressover.news/articulos/el-efecto-tetris-y-la-psicologia/ (document), 2.3

[4] davdav1233 and cosin307, "Tetris kicks." [Online]. Available: https://four.lol/srs/j-kicks (document), 2.4, 2.5, 2.6, 2.7

[5] Piotrlogin, ERR0R, and JJBisHere15, "Tetris t-spin single." [Online]. Available: https://harddrop.com/wiki/T-Spin_Guide (document), 2.8

[6] JJBisHere15 and Pineapple, "Tetris t-spin double." [Online]. Available: https://harddrop.com/wiki/T-Spin_Double_Setups (document), 2.9

[7] Piotrlogin, ERR0R, JJBisHere15, and Pineapple, "Tetris t-spin triple." [Online]. Available: https://harddrop.com/wiki/T-Spin_Triple_Setups (document), 2.10

[8] Amazon, "Elegoo arduino." [Online]. Available: https://www.amazon.com/-/es/Elegoo-Placa-ATmega328P-ATMEGA16U2-Arduino/dp/B01EWOE0UU (document), 3.1, 3.2

[9] S. Petite, "Main piece out of grid." [Online]. Available: https://www.digitaltrends.com/gaming/tetris-99-tips-and-tricks/ (document), 4.2

[10] rjeem, "Blocks out of grid." [Online]. Available: https://mag.rjeem.com/tetris-99-review-switch-eshop/ (document), 4.3

[11] M. Valderas, "Game effects." [Online]. Available: https://esports.eldesmarque.com/noticias/tetris-battle-royale-tetris-99-nintendo-switch-62624 (document), 4.4

[12] R. L, "Neural network model," last accessed 11 September 2021. [Online]. Available: https://medium.com/mlearning-ai/reinforcement-learning-on-tetris-707f75716c37 (document), 5.1

[13] C. Dueñas, "Game menu," last accessed 1 May 2021. [Online]. Available: https://github.com/ChristianD37/YoutubeTutorials/tree/master/Menu%20System (document), 6.1

[14] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent interaction," last accessed 15 April 2017. [Online]. Available: https://openai.com/blog/emergent-tool-use/ 1.1

[15] JohnnyTurbo, "Srs system," last accessed 1 May 2021. [Online]. Available: https://github.com/JohnnyTurbo/LD43 2.6

[16] Tim, "Tetris visual interface," last accessed 22 April 2021. [Online]. Available: https://www.techwithtim.net/tutorials/game-development-with-python/tetris-pygame/tutorial-1/ 2.6

[17] N. Faria, "Q-learning nuno faria," last accessed 22 April 2021. [Online]. Available: https://github.com/nuno-faria/tetris-ai 5.2.1

[18] I. El-Ashi, "El-tetris – an improvement on pierre dellacherie's algorithm," last accessed 5 July 2021. [Online]. Available: https://imake.ninja/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/ 5.2.1

[19] R. L, "Qlearning zeroize318," last accessed 11 September 2021. [Online]. Available: https://github.com/zeroize318/tetris_ai 5.2.2

[20] lukasbrchl, jrosebr1, abhiTronix, and drmacdon, "Filevideostream," last accessed 15 Februay 2021. [Online]. Available: https://github.com/jrosebr1/imutils/blob/master/imutils/video/filevideostream.py 6.2

[21] "Opencv about," last accessed 11 September 2021. [Online]. Available: https://opencv.org/about/ A.1

[22] "Tensorflow," last accessed 17 July 2021. [Online]. Available: https://en.wikipedia.org/wiki/TensorFlow A.2

[23] Wikipedia, the free encyclopedia, "Keras," last accessed 17 July 2021. [Online]. Available: https://en.wikipedia.org/wiki/Keras A.3

[24] "Usb-ttl converter," last accessed 2 September 2021. [Online]. Available: https://www.quora.com/What-is-a-TTL-signal-in-communication-and-when-using-a-USB-port-why-do-we-use-a-USB-to-T A.4

[25] M. Littman, "Markov decision processes," last accessed 2 September 2021. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/markov-decision-process A.6