Technical Guide
Project Title:

An AI-based multiplayer platform for Tetris/Puyo Puyo through reinforcement learning

Gergely Gellert	17379616
Nevan Oman Crowe	17407926

Supervisor: Annalina Caputo

Date Finished:

Abstract:

In this project we created an adaptation of the games Puyo Puyo and Tetris. Tetris is a game where the player clears lines of blocks by filling a line in completely with no holes. On the other hand, Puyo Puyo is a game where the aim is to match four or more of the same coloured “Puyos” with the hope of chaining together multiple matches with the least amount of input. Similar in concept to the popular game Candy Crush.

In our program you are to play both alone in single player, and against another user or an Ai in multiplayer. We created an AI to play both games as efficiently as possible through reinforcement learning and specifically deep Q-learning.

This project was created in Python utilising open source libraries such as Pygame and Tensorflow. The program will be accessible through running an executable file on the user’s computer. Along with this, the user will be able to interact with the interface using their mouse and control the games using their keyboard.

Our hope with this application is that it will provide an enjoyable experience for anyone wanting to play either Tetris or Puyo Puyo.

Table Of Contents

1. Introduction

1.1. Overview

Our project is a Python-based application aimed to recreate the games “Puyo Puyo” and “Tetris” and create an AI to play with it.
Puyo Puyo/Tetris incorporates the classic games of “Tetris” and “Puyo Puyo” in both a single player mode and a multiplayer mode.
In single player mode, a user can play either game until they lose. In the multiplayer mode a user can play either “Tetris” or “Puyo Puyo” against another player also playing either “Tetris” or “Puyo Puyo” or an AI substituted in for the second player. It is also possible to let 2 AIs play against each other in multiplayer.
This technical manual illustrates the entire development cycle this project went through, highlighting all the initial research, preparations and investigations needed to produce this project. Technologies as Pygame and Neural Networks were used as the primary backbones of this project.
Due to the complexity of this project there were many problems and defects that arose during development which were documented and addressed in this document.

1.2. Motivation

Our main motivation for creating this project was a common interest in games and AI. We wondered if we could create one of our favorite games along with an Ai to play against. We were confident and up to the challenge because last year our third year project also involved creating an Ai which we had success with.

1.3. Glossary

Python

Specifically Python3 is an interpreted, high-level and general-purpose programming language. That will be used to develop the project

Pygame

Pygame is a set of Python modules designed for writing video games. It includes computer graphics and sound libraries designed to be used with the Python programming language.

Pytest

Pytest is a set of Python modules designed to test code to ensure they are working as intended.

Graphical User Interface
Also known as GUI, it is the visual part of the program to show the user the board state and the components interacting with each other through graphical icons.

Neural Network
Neural Networks is a class of algorithms designed to mimic the way the human brain operates. These algorithms attempt to recognise patterns in data that they receive and act to ensure the best possible outcome.

TensorFlow
TensorFlow is a free open source library for machine learning primarily used for deep neural networks.

Keras
Keras is an open source library that provides an interface in TensorFlow for neural networks.

Reinforcement Learning
Reinforcement Learning is a type of Artificial Intelligence algorithm that learns to make a sequence of decisions based on information given to it through a sequence of trial and error.

Deep Q-Learning
Deep Q-Learning is an algorithm of Reinforcement Learning that focuses on finding the most optimal decisions the A.I. can make.

Board
The Board is the environment in which the user can see, interact with and manipulate.

Thread
A thread is a piece of code running in parallel and often independently of the main program

Q-Values

Used in the Bellman Equation for Reinforcement Learning to tell how good it is to be in a certain state.

The Bellman Equation
An equation that determines the value of a decision at a certain point in time in terms of payoff/reward. Used in reinforcement learning.

Numpy
Open source Python Library that allows us to use and manipulate multi dimensional arrays and matrices.

Tetromino
A geometric shape composed of four squares, connected orthogonally, one variation of which makes up the Tetris block.

Puyo
A sphere or slime shaped object in the games Puyo Puyo, 2 of which connected make up the Puyo block.

Pyinstaller
A python library that bundles a Python application and all its dependencies into a single package that enables the user to run the packaged app without installing a Python interpreter or any modules

Batch File
A batch file is a script file consisting of a series of commands to be executed by the command-line interpreters, stored in a plain text file.

2. Research

Our first course of action was researching how we build the games Tetris and Puyo Puyo. We started by analysing both game's structures,dynamics and game interactions. We did this by observing real-time games and by doing on hand research of the games ourselves. We did this to understand how the games work and how the end product should feel.

Once we were satisfied with our understanding we then looked into how the inner workings of each game worked, e.g. point scoring, dealing with special cases, how fast pieces fell. We did this primarily through forum posts and wiki pages that dissected how each mechanic functioned. From these posts we got insight on such things as how scores are calculated and how fast each Tetris block falls relative to how long the user played that Tetris game.

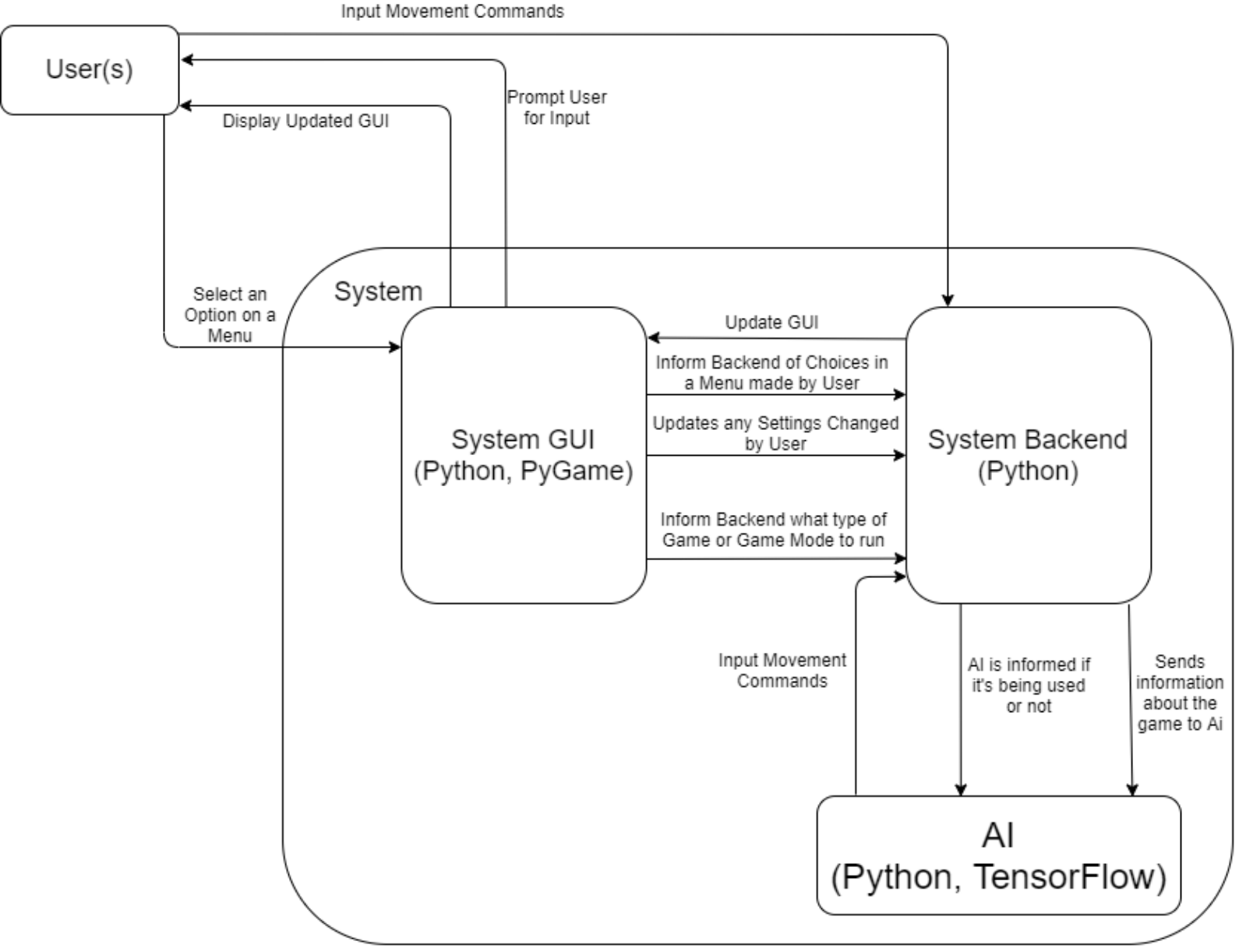
Our next step was researching an open source library called Pygame. Before this project we never used Pygame before, so while we had a lot of experience with Python, learning Pygame was a bit of a challenge for us. This was due to it being focused around producing a GUI which we haven't really done before in Python.

Starting from one of the seminal book on AI (Artificial Intelligence A Modern Approach) we investigated the use of reinforcement learning, in particular Q-Learning as a way to create autonomous agents that are able to learn from experience. We borrowed the book from the DCU library and from it we understood the basics of reinforcement learning
.
During the course of our research we concluded that we would build a Neural Network based AI. We decided to use an open source library called TensorFlow to create and train our Neural Network model so it can play the games we created. We ended up deciding to let the model generate its own data by at first doing random moves in the game it was playing. Over time it would do less and less random moves and train itself from the data it generated.

3. System Design

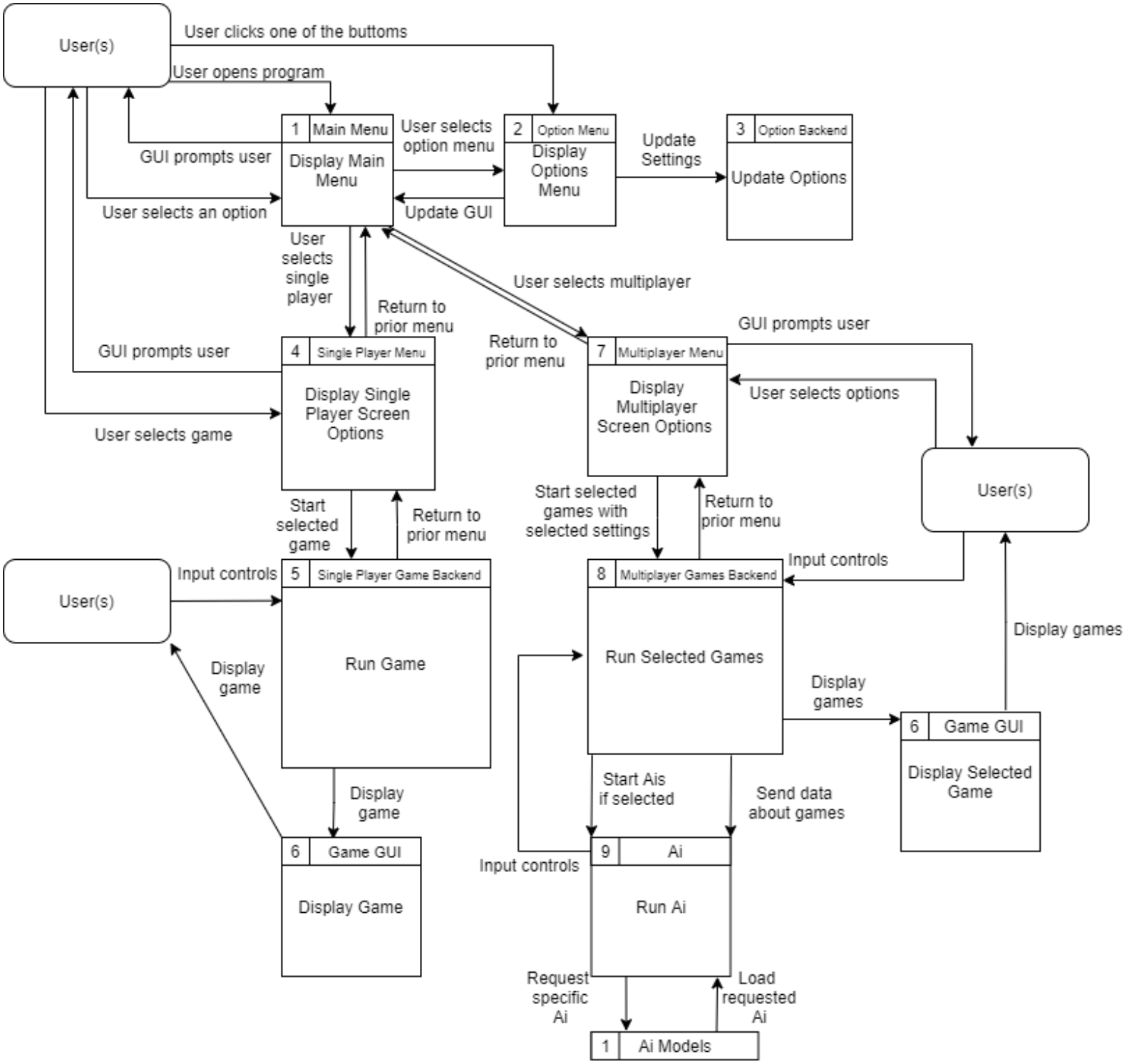
3.1 Context Diagram

The figure below shows how the user(s) interacts with the system. Along with this it shows how our systems interact with each other.



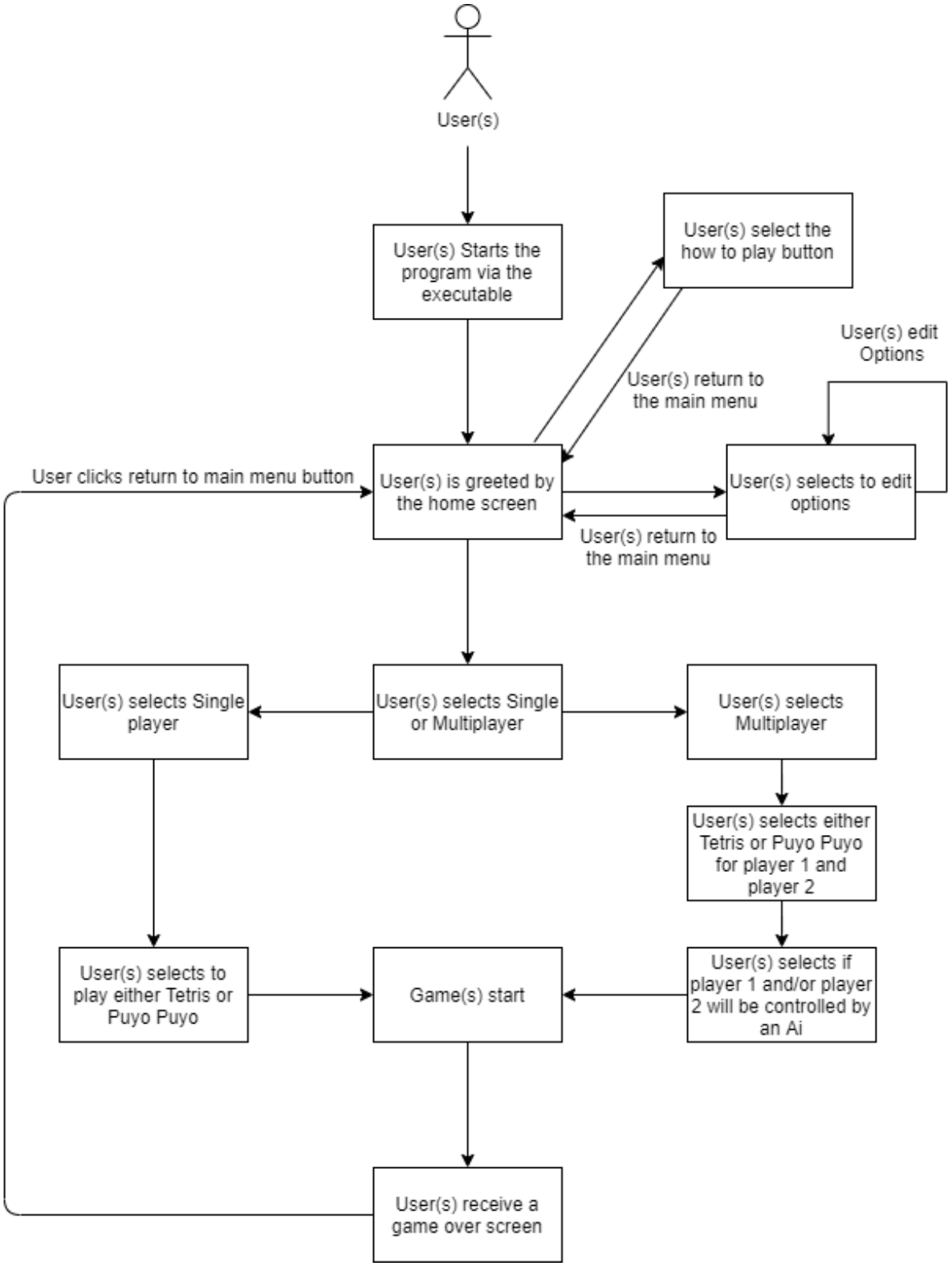
3.2 Data Flow Diagram

Shown below is our Data Flow Diagram, the user begins to interact with our system starting at the top left. Module 6, the Game GUI appears twice for ease of readability.



3.3 Use Case Diagrams

Below is how a user would use the program.



4. Implementation

4.1. Tetris and Puyo Puyo

The first things we implemented were the games Tetris and Puyo Puyo simultaneously. The backbone for our implementations is a 2d array which served as our “board”. This array let us represent the environment the user sees and interacts with while they’re playing either game.

The users interact with this environment by manipulating blocks that they will eventually place on each game’s board. We implemented these blocks by giving them each an x and y coordinate to signify where they were on these boards, a shape to tell both the board what type of shape to put onto the board, and what colour that piece is so the user can easily make out what they’re currently interacting with.

To implement the games we used an open source and widely used library, named Pygame. We used it to build our GUI so the users can see how they’re interacting with our games. Along with this we used Pygame’s inbuilt keyboard mapping to allow our users to interact with our games using their keyboards.

4.2. Single Player Mode

We wanted to implement the games in such a way that allowed for AI development alongside game development. We built a singleplayer screen that gives the user a very simple option, either play Puyo Puyo or play Tetris and upon choosing brings the user straight into the selected game. We decided to develop the single player games as a base for the whole game. Each game is built upon its board, a 6x12 board for puyo and a 10x24 board for Tetris. Each game supplies the player with a block, Tetris blocks consists of a single Tetromino and Puyo Puyo blocks consists of two Puyo of varying or same colour connected together. The player moves these blocks into positions on the board to score points and remove old pieces from the board. This is accomplished by filling up one or more horizontal lines of the board in Tetris or by placing 4 or more of the same colour puyo beside each other.

The games run in a large loop that continues until the board fills up and the game ends bringing the player back to the home screen. This loop waits for the player to interact with the keyboard and then moves and

interacts with the blocks in different ways depending on which key is pressed. This loop also updates the boardstate and draws everything on the screen for the player. This loop provides quick and continuous feedback for the game and the player.

The aim of the single player game is to provide the player with a way to play the games by themselves at their own pace and not have a distraction from another player or AI. It also provides a way for the player to get used to how the games function and to learn how to play the game better. The games are built to provide as close to an infinite game as possible for the player with the game going on as long as the player can play until their board fills up. The single player mode encompasses the base of both games while excluding disruption from other players or AI.

4.3. Neural Network Creation

To create our Ai after researching in depth we decided to go for an Neural Network based Ai. We used the open source python library called Tensorflow. Along with this to run the library we had to use a version of python called Anaconda Python to have greater control over what libraries we used in our environment.

Since Tetris and Puyo Puyo are two very different games, one being filling up the board to clear lines and another a colour matcher, we decided to create and train 2 seperate Ais.

After extensive analysis of multiple solutions and models we decided that the Tetris Ai would have an input layer consisting of 4 nodes. While the Puyo Puyo Ai would have an input layer consisting of 5 nodes. A node is also called a “neuron”, it processes all the information it receives and provides an output. Neural networks take a numpy array as input, which is represented as a list of numbers. Having 4 input nodes means we can give the Ai 4 of these arrays at a time.

Along with this we decided to have 2 hidden layers, each consisting of 32 nodes. These hidden layers are all connected to each neuron in the layers before and after it by a series of weights. These weights are assigned to show the relative importance of inputs from other neurons. The second hidden layer is connected to our output layer.

For our output layer, we decided to go with a layer of size 1. Our output consists of a “confidence interval”. This confidence interval shows to us how confident the Ai is in it’s prediction.
We decided on an output layer of size 1 because of the way we designed the way the AI receives input. With this the AI decides which sequence of moves is best. This is shown with us asking the Ai how confident it is in each sequence of button presses, we then input the sequence the Ai is the most confident in. The sequence of moves possible for our Ai in these scenarios is moving the piece it is currently controlling right a certain amount of times depending on what game it is playing, moving left a certain amount of times again, rotating the current piece 0 to 3 times and finally dropping the piece. Dropping the piece involves placing the piece to the lowest possible point directly below

4.4. Neural Network Training

The training method we decided to use is Reinforcement Learning since training the Ais to always make decisions that would lead to both them not losing the game they’re playing and also obtaining the highest reward possible. The Ai learns to play by repeatedly playing simulations of either Tetris or Puyo Puyo until we deem it has learnt a satisfactory amount. We also implemented Deep Q-Learning, with this it would allow our Ai to determine the right sequence of moves that will allow it to receive the maximum award, thus allowing it to play with the best efficiency.

The way we ran the training is we had either Tetris or Puyo Puyo running separately in a thread and had the AI interact with the game. This would come extremely close to how a real person would be playing the games.

First thing we did was implement a way to extract the necessary information needed from the games. The way we solved this problem through the use of queues. Queues in python are thread safe so it would eliminate any concurrency issues that might happen due to the game being run in a separate thread. When either game generates a new piece that the Ai is required to interact with to progress, it sends this information to this queue which then the Ai processes and determines what to do.

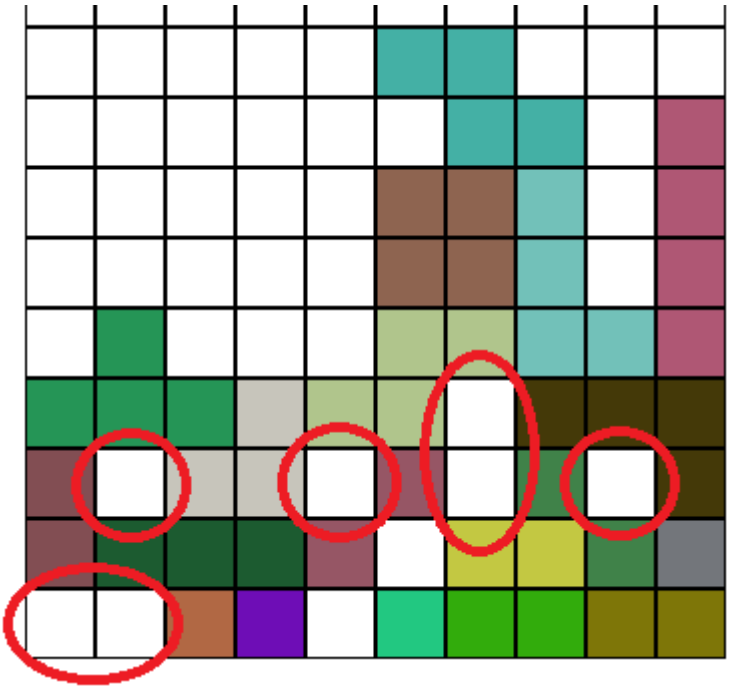
Along with this once the Ai inputs what moves it should do, the game sends over the results of what the Ai did. This includes how many lines got cleared and the current score of the game.

Before we give the Ai information and ask it to decide how it should manipulate the game it's currently playing we parse the game's board state into a numpy array. First we simulate each possible end scenario that would arise from a sequence of movements, we then take this end scenario and parse it into pieces of information that the Ai will process.

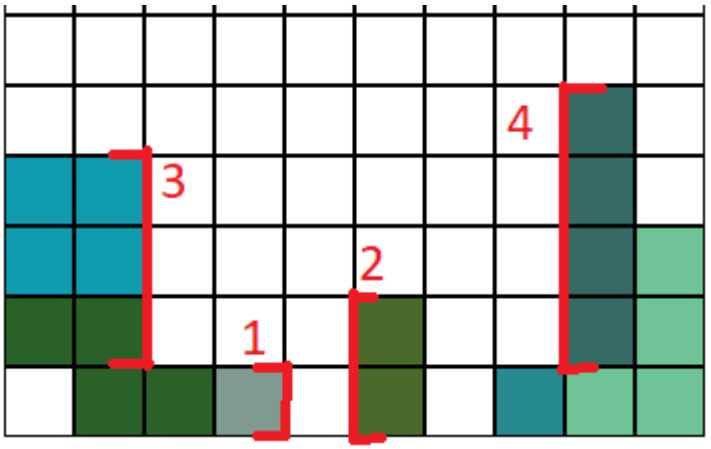
The reason we need to parse the board is because the size and shape of the board along with it's pieces, all change frequently depending on the state of the game. For example, a board with 2 columns filled would be different size and shape than a board with only the bottom 2 rows filled. Due to our Neural Network only accepting a static sized input because of the size of our input layer, we have to parse each game state to give a consistently sized representation that the Neural Network can use as input.

For Tetris, once we receive For Tetris we parse the board into 4 different pieces of information. These include the number of holes present , bumpiness, the sum height of all columns present and the amount of lines cleared with a sequence of moves. Below is a visual representation of the concepts just described

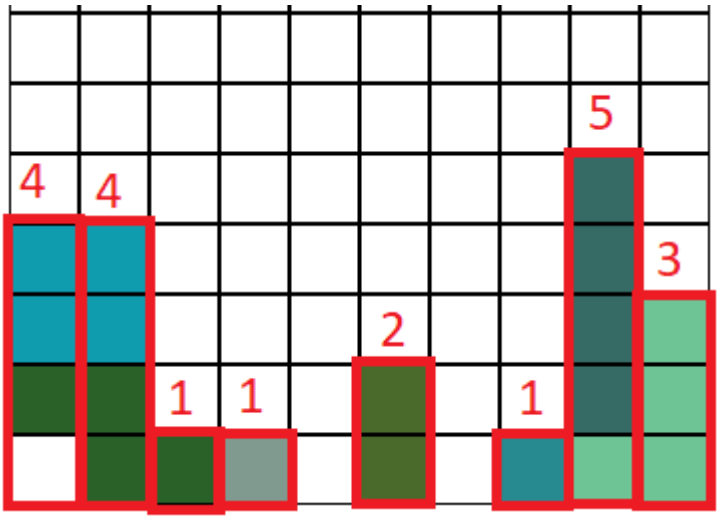
A variety of holes in the board



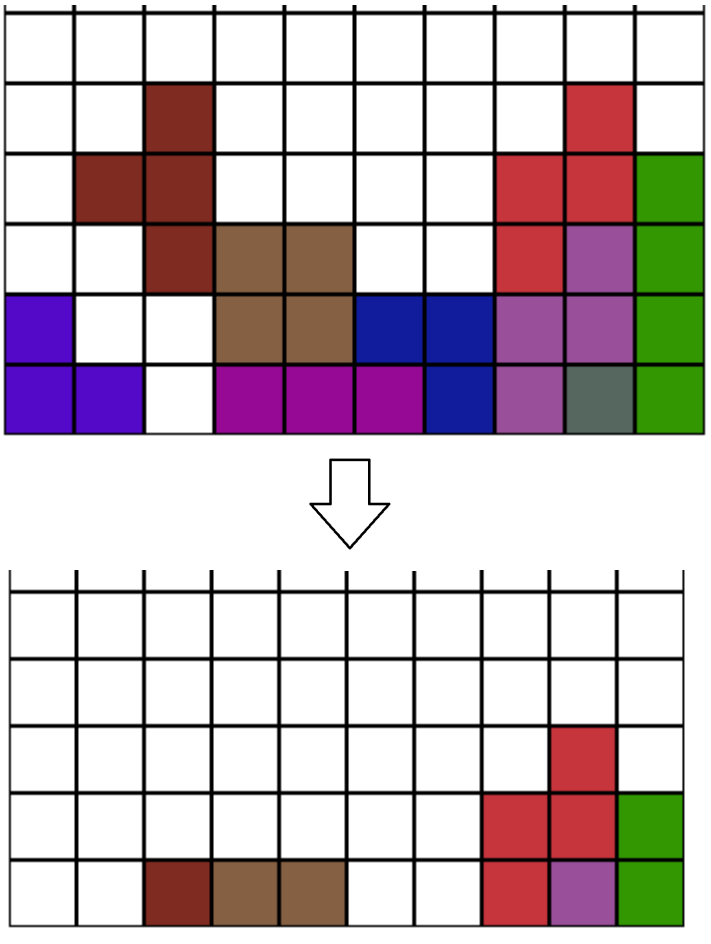
Bumpiness (Difference in height between columns)



Sum Height of all the Columns (In this case it would be 21)

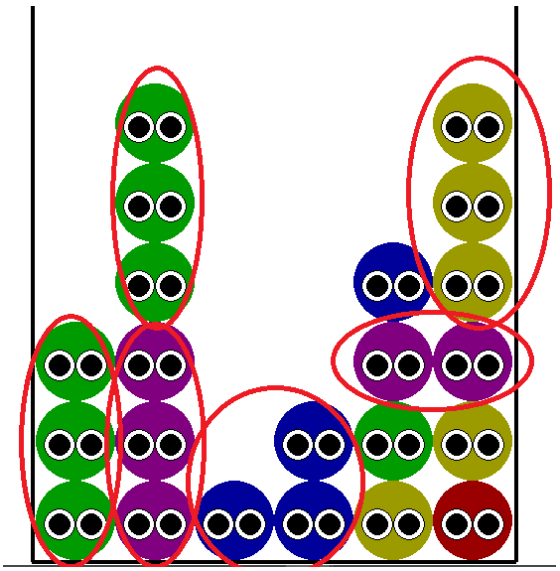


Lines Cleared with a sequence of moves (2 lines cleared)

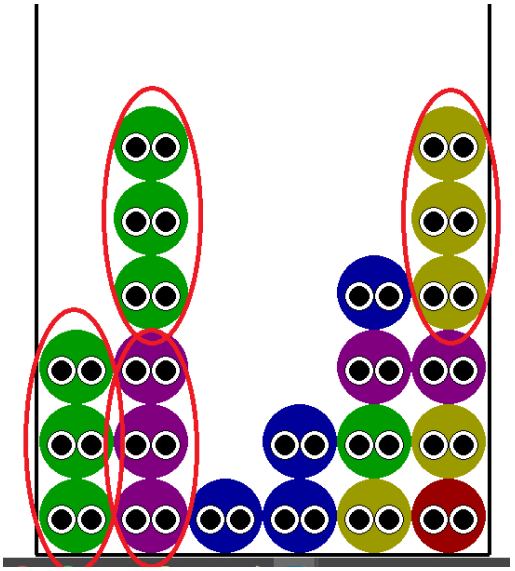


For Puyo Puyo we instead parse the board into 5 different pieces of information. These include puyos of the same colour right near each other, puyos of the same colour in a pillar of 3, the sum height of all columns on the board, the height of the highest column on the board and the number of puyo chains cleared with the corresponding sequence of actions. Below are visual representations of the concepts just discussed.

Puyos Near each other (17 puyos are near each other)



Puyos in a pillar of 3 (4 Pillars)



Once all of the above data has been parsed we pass all of it to the Ai. From this data we receive a confidence interval for all possible combinations of movement in the current game. We input the combination of moves with the highest confidence interval to the game. Once all the moves have been inputted the game then gives us the results of the Ai's input through another queue system.

To train the Ai we need data to train it. We decided to have the AI gather data by playing the games and recording what it sees. We stored these pieces of data in a deque object. To ensure we don't use outdated data we limited the the deque object to only hold 10000 data points. Once this limit has been reached it deletes the oldest data point and replaces it. To first build up some data to use to train we made the AI have a percentage chance to do a random sequence of moves, with the chance

of doing a random sequence of moves instead of a calculated move decreasing with each game played.

Now once we receive the results of the Ai's action, we calculate the reward it receives from performing those sequence of moves. It receives 1 point for successfully placing an object onto the board, and a multiple of 2 points, depending on how many lines or puyos it cleared from the last sequence. It will also receive a punishment of negative 5 points if the sequence of moves caused the Ai to lose the current game. We then create a data point using these results and put it in our deque object. This data point includes the parsed data describing the original state, the now updated state, the reward attributed with this particular transition, and if this transition caused the Ai to lose or not. All of the above data describes the transition from one board state to another.

After each game we choose a certain amount of these transition points to train with. Using these data points we used the Bellman equation to update the q values for this specific transition.

Eventually after repeating this sequence enough we end up with a functioning Ais for both games.

4.5. Multiplayer

We wanted to expand the single player versions of the games to have more depth and replayability as the players ability at the game increased. So we decided to take the base game built for single player and build upon it. We build a Multiplayer screen to give the player(s) an easy way to select all the aspects of the game they would like to play.

The options are split up into deciding the game going to be played e.g. Puyo Puyo or Tetris along with the decision of making one or both of the games played by an AI. This opens up many different options to the player(s). It gives them an opportunity to play with another player, play against an AI to have an interactive single player experience or to have two AI to play the games and have a way to learn from them playing.

This time the games contain base version from single player mode but also contains the multiplayer disruption ability. Should the player score enough points by clearing multiple lines or connecting and chaining enough puyo. This creates disruption for the other player and is added

to the other players queue that removes disruption every 4 seconds and adds it to the plates board. The disruption for Tetris comes in the form of extra lines placed on the board under the player made lines. The disruption for Puyo Puyo comes in the form of grey puyo placed randomly on top of the player constructed board. The only way to remove these grey puyo are to connect 4 or more of the same colour beside it.

Once the options are selected and the games have been started their loops look very similar to the single player version but have been deconstructed to work together. The game loop runs two deconstructed loops, one for either game. It goes through the actions for each game simultaneously, it controls the actions from both players/AI, it controls the disruption queues for both boards and also draws the two boards to the screen for the players at the same time. This allows the games to play side by side without having any concurrency issues and allowing the players to play easily together on one computer.

4.6. Integrating Ai Into Multiplayer

The final step was integrating our Ais that we cultivated into the multiplayer aspect of our game. To simulate the Ai being a user we decided to have the Ais run separately in a thread. ThisTo extract the necessary information about either game to feed our Ai we went with the queue based approach again. This means that when for example a game generates a new block for the user to interact with it would send this information to a queue object which our Ai would retrieve and use in it's calculations.

Again like the training method we would first simulate each possible sequence of movements possible and parse the resulting board state into pieces of information into a numpy array.
We then query the Ai with all of these arrays and it inputs into the game what sequence of movements it should do.

4.7. Creating Game Executable

We wanted people to be able to play the game easily and not have to install python and all the dependencies needed to make the project work, just to play the game. So we made use of a python library named Pyinstaller to turn the all of games files into one executable that is easily

runnable on any computer. We have built one of these files and placed it as a part of the repository for anyone to use who doesn't want to install python and all the dependencies.

We also wanted to supply users with the ability to build and create their own game executable files, to do this we wrote up a batch file that runs the commands for the user and can easily run on their own computer to build and make the executable.

5. Sample Code

5.1 Queue System to give Ai information about the game

One of the first things an Ai needs to come to a decision of how it should play it's game is information about that game. The way we give this information about the game is through a queue system. We have a shared queue object between the Ai and the game. The game then appends information onto this queue when a new object to control is created or when an object that was controlled has been placed. From there the Ai can use this information to perceive the game's surroundings and make a decision of what to do. Figure 1 is Tetris putting information about the game onto the Queue system while Figure 2 shows the Ai retrieving this information.

Fig 1.

```
while not gameOverQueue.empty():
    if tetrisInst.currentBlock is None and not tetrisInst.gameOver:
        tetrisInst.newBlock()
        #New board state, put the new information onto the queue system
        if not tetrisInst.gameOver:
            currentPieceQueue.put(tetrisInst.currentBlock)
            nextPieceQueue.put(tetrisInst.nextShapes)
            currentBoardQueue.put(tetrisInst.field)
            currentReservedQueue.put(tetrisInst.reserved)
```

Fig 2.

```
if not currentBoardQueue.empty() and not currentPieceQueue.empty() and not nextPieceQueue.empty():
    #Get all the information about the board from the queue system
    tempBoard= currentBoardQueue.get()
    tempBlock = currentPieceQueue.get()
    tempNextBlocks = nextPieceQueue.get()
    tempReserved = currentReservedQueue.get()
```

5.2 Board Parsing

Once the Ai receives the information that it requires we begin the process of parsing this information we can provide to the Ai. First we simulate every single possible outcome and then parse each outcome. Taking Tetris as an example, we parse the board into 4 distinct pieces, how many holes the board has, the bumpiness of the board, the sum height of all the columns and how many lines the resulting board will clear.

In the next 4 figures we use examples how we parse boards from the game Tetris.

In figure 3, we simulate every possible future board position with the current board and pieces.

In figure 4, we give the resulting board to our penalityCalc function which gives that board a score.

In figure 5 we calculate the bumpiness and the sum height of the board. Using the zip command in python this allows us to check column by column. We determine the sum height of the board by summing together the total height of each column. We also calculate the bumpiness of the board by subtracting an adjacent column's height from the current column's height and sum all of our findings together. Once this is all done return this information.

In figure 6 we calculate how many holes are present in this particular board. We do this by checking if there's any blocks directly above any blank spaces.

Fig 3.

```
def simulateBoard(simulation,tempBlock,tempBoard,tempReserved,listItem,tempNextBlocks,score,level):
    #Do a deepcopy to ensure we always start off with the same objects
    simulation.currentBlock=deepcopy(tempBlock)
    simulation.field=deepcopy(tempBoard)
    simulation.reserved = deepcopy(tempReserved)
    simulation.nextShapes = deepcopy(tempNextBlocks)
    simulation.score=score
    simulation.level = level
    for item in listItem:
        if item!="":
            #The last command will always be a "drop" which returns how many lines were cleared and if it caused the Ai to lose
            gameOverAndLinesCleared = keysDictAI[item]()

    return penalityCalc(simulation.field,gameOverAndLinesCleared[0]), gameOverAndLinesCleared[1], simulation.score
```

Fig 4.

```
def penalityCalc(board,linesCleared):
    # Here we calculate how good this particular end result is
    temp = bumpinessAndTotalHeight(board)
    return [numberOfHoles(board),temp[0], temp[1],linesCleared]
```

Fig 5.

```
def bumpinessAndTotalHeight(board):
    columnHeights=[]
    bumpiness=0
    totalHeight=0
    #We calculate the bumpiness(difference in height of adjacent columns) and the sum height of all the columns
    for column in zip(*board):#the zip(*board) allows us to go by column by column instead of row by row
        i = 0
        while i < len(board) and column[i] == 0:
            i += 1
        totalHeight+=len(board)-i
        columnHeights.append(len(board)-i)
    for i in range(len(columnHeights)-1):
        bumpiness+=(abs(columnHeights[i]-columnHeights[i+1]))
    return [totalHeight,bumpiness]
```

Fig 6.

```
def numberOfHoles(board):
    #Here we calculate how many holes there are in a board
    #A hole is defined by a blank space with a filled space directly above it
    holes = 0
    for i in range(len(board)):
        for j in range(len(board[i])):
            if i!=0 and board[i][j]==0 and board[i-1][j]!=0:
                holes+=1
    return holes
```

5.3 How the Ai controls the games

When the Ai decides on what sequence of moves to input it provides this input by putting each command onto a separate queue object that is again shared by both the Ai and the game, which is shown in figure 7. The game checks if this queue is empty and if it isn't it performs whatever command is on that queue as seen in figure 8.

Fig 7.

```
def doMoves(movements,movementsQueue):
    #This function here signals to the game what inputs to do that the Ai decided it should do
    for move in movements:
        movementsQueue.put(move)
```

Fig 8.

```
#Here the game checks if there is any input from the Ai to do, if there is do it
if aiFlagP1 and not movementQueueP1.empty():
    tempMovement= movementQueueP1.get()
    #These dictionaries are linked to commands to manipulate the games such as move left or rotate
    keysDictP1[tempMovement]()
if aiFlagP2 and not movementQueueP2.empty():
    tempMovement = movementQueueP2.get()
    keysDictP2[tempMovement]()
```

5.4 How the Ai trains

Everytime the Ai decides on a sequence of moves to make, it calculates the reward it receives from these series of moves. This is shown in figure 9. Once the Ai loses its current game, it selects a random batch of moves to train on and trains itself using them. Through enough iterations it eventually trains itself to become the best version of itself it can be. The function used to train the Ai is shown in figure 10

Fig 9.

```
#It gathers the results of it's input through another queue system
tempNewResults = resultsQueue.get()

linesCleared = tempNewResults[0]
#Here it calculates the reward it wil receive from this result
#It receives 1 point for placing an object, and a multiplicative for how many lines/puyos it cleared
#This is to incentivise it clear more lines/puyos in one go
overAllScore = 1 + linesCleared*2
#If this resulted in the Ai losing the game then punish it to avoid these moves later on
if simulatedGameOverList[bestMoveIndex][0]:
    overAllScore-= 5
print(episode, ": ", counter, overAllScore, linesCleared)
#And once this is done, add all of this to its memory bank
tetrisAgent.addToMovesMemory(currentBoardScores, bestScore, overAllScore, simulatedGameOverList[bestMoveIndex][0])
```

Fig 10.

```
def train(self, batchSize = 32, epochs=3):
    if len(self.memory)>batchSize and len(self.memory)>= 5000:
        #Here it choose a certain number of random states to train off of
        batchToTrainOn = random.sample(self.memory, batchSize)
        nextStates = np.array([x[1] for x in batchToTrainOn])
        nextQs = [x[0] for x in self.model.predict(nextStates)]
        x = []
        y = []
        #Here we update the Q-values for each state choosen and we train our model using it
        for i,(state,futureState, score, gameOver) in enumerate(batchToTrainOn):
            if not gameOver:
                newQ = score + 0.95 * nextQs[i]
            else:
                newQ = score
            x.append(state)
            y.append(newQ)
        self.model.fit(np.array(x),np.array(y),batch_size=batchSize, epochs=epochs,verbose=0)

        if self.epsilon>self.epsilonMin:
            self.epsilon-=self.epsilonDecay
```

5.5 Checking for connecting Puyo recursively

Whenever a puyo block is placed on the board the game has to check if either of the puyo in that block matches the other puyo around it and whether or not there are enough Puyo of the same colour to have them pop. This is achieved recursively for each of the puyo. The function checks each of the positions around it (up,down,left,right) and if a matching colour is found it does the same for that puyo along with a list of puyo already checked to stop backtracking. This goes through all Puyo of the same colour and returns their positions.

Fig 11.

```
def checklocalrecur(self,y,x,colour,poschecked =[]):
    values = [] #up,down,left,right

    if y-1 >=0 and (y-1,x) not in poschecked :#up
        poschecked += [(y-1,x)]
        if self.board[y-1][x] == colour:
            values += [(y-1,x)]
            values += self.checklocalrecur(y-1,x,colour,poschecked)

    if y+1 <=11 and (y+1,x) not in poschecked:#down
        poschecked += [(y+1,x)]
        if self.board[y+1][x] == colour:
            values += [(y+1,x)]
            values += self.checklocalrecur(y+1,x,colour,poschecked)

    if x-1 >=0 and (y,x-1) not in poschecked:#left
        poschecked += [(y,x-1)]
        if self.board[y][x-1] == colour:
            values += [(y,x-1)]
            values += self.checklocalrecur(y,x-1,colour,poschecked)


    if x+1 <=5 and (y,x+1) not in poschecked:#right
        poschecked += [(y,x+1)]
        if self.board[y][x+1] == colour:
            values += [(y,x+1)]
            values += self.checklocalrecur(y,x+1,colour,poschecked)

    return values
```

5.6 Moving Down Floating Puyo Blocks

This function completes an essential part of the Puyo Puyo game as any Puyo that is not on top of another or on the bottom of the board needs to fall down the board to one of those two positions. This function goes through the board and moves down any Puyo “floating” due to placement or from other Puyo being removed and then adds it’s new final position an array to return at the end as each of these new puyo needs to be checked for any consecutive connections or “chains”.

Fig 12.

```
def move_down_floating(self):
    positions = []
    y = 10
    x = 5
    while y >=0:
        while x >=0:
            if self.board[y+1][x] == self.boardblank and self.board[y][x] != self.boardblank:
                oldy, oldx = y,x
                newy, newx = y,x
                while (newy+1) <= 11 and self.board[newy+1][newx] == self.boardblank:
                    newy = newy+1
                positions += [(newy,newx)]
                self.board[newy][newx] = self.board[oldy][oldx]
                self.board[oldy][oldx] = self.boardblank
            x -= 1
        y -= 1
        x = 5
    return positions
```

6. Problems Solved

6.1 Providing our Neural Network with suitable input

One of the first problems we came across was providing suitable input for our Neural Network. We first attempted to input both the current board, the current object the Ai is controlling and the object that will come after the current object to the Ai. The problem with this approach was that our Neural Network required a Numpy array of a pre-specified size and dimension for it to be accepted. Due to the difference in sizes of the board, the current object and future object turning everything into a singular Numpy array would yield an array that doesn't have a consistent size which our Neural Network wouldn't accept as valid input.

Our solution to this problem was parsing the board into different pieces of information depending on the game being played.
For Tetris we parsed the board into 4 distinct pieces of information. How many holes are present in the board, what is the sum height of all the columns, the sum differences of height for adjacent columns (bumpiness), and how many lines did the last sequence of moves cleared.
For Puyo Puyo we parsed the board into 5 distinct pieces of information. These included the amount of same coloured puyos directly beside each other, how many same coloured puyos are in a tower of 3, the sum height of all columns, the height of the tallest column and how many puyos it cleared from the last sequence of moves.
This allowed us to create a streamlined series of information that we can then turn into a numpy table of a specific size for our input layer in our Neural Network.

6.2 DeSync issues

Another one of our problems we encountered was the Ai causing desync issues when it interacted with any of the games during training. This was caused by the Ai directly inputted movement commands into the game. One of the main features of the games is that after a certain amount of time the object the user is currently interacting with moves down the screen to add a sense of urgency and difficulty. The issue stemmed from when the Ai inputted the command “drop” which immediately places the object it is interacting with to its lowest possible position and adds it to the game board.

During the process of adding the object to the board when the command “drop” is issued, the variable that stores this object is set to Null. Since the Ai and the game it’s currently playing are running independently of each other (the game was run in a thread by itself during training) when the game tried to move down the object, or draw the object on screen it would give us a Null Pointer Exception and crash the game.

The way we solved this problem is by implementing a queue system. Instead of directly inputting movements at the speed of light into the game it would append them onto a queue object that can be accessed by both systems. The game itself will then execute any commands inside this queue one by one instead of the Ai manipulating the object, and thus won’t try to draw or interact with an object that doesn’t exist.

This solution also allowed us to limit how fast the Ai could input movement to make it fairer if it were playing against a human opponent for example, since a human wouldn’t be able to keep up with an Ai inputting 15 movement commands a second.

6.3 Ai not functioning properly

At various times we weren't satisfied with the end product of the Ai's training. For example, one of the earlier versions of the Puyo Puyo Ai was really good at doing everything but scoring and clearing Puyos. Besides fixing bugs in the way we parsed and simulated possible transitions, we experimented with parsing extra information. For example for Tetris originally we weren't giving the Ai the information of the amount of lines it cleared per sequence of movements. Once we started giving the Ai this piece of information it ended recognising that to not lose games, it has to clear lines and started doing so.

6.4 Multiplayer Integration

A large problem that arose for the games was the integration of both games and enabling them to work together as a cohesive part running together. Our original idea was to have the two base games running side by side but in separate threads, they would be handled by a Multiplayer game system which would handle the games and interactions between them. This idea ran into some problems and issues. For example It would make it difficult to have the games interact with each other in separate threads and depending on how fast the games are played they could run into concurrency issues.

This led us to work the our design that we have now, with dissecting each game's loop and having one large multiplayer loop handle both, this enabled much easier interaction between the two games and enabled a much smoother integration process.

6.5 AI Integration

For the AI Integration we had to plan out how we wanted the users to interact with AI and how we could bring the two sides together easier and we ran into some problems. We needed to make sure that the AI slotted comfortably into the game system and didn't feel out of place. So we decided to put it in as a part of the multiplayer system and have it act as another player, all that was needed from the user's side was to click a button to select the AI. We restructured the Multiplayer loop and made a similar but separate loop function to play the games when an AI was chosen. This loop would handle the threading of the AI and it's interactions with the Multiplayer game as whole and if it needed to interact with the players board. This separated the Multiplayer out into two parts, with and without AI.

6.6 AI playing too fast

We ran into the problem that the AI can play the games at a much faster rate than any human player could ever hope to achieve. This also led to some serious concurrency issues where the AI would try to do actions before the loop had finished. We solved this by adding in a form of action counter so that the AI could only act on an action it had decided after a certain amount of loops of the game. This enabled the AI to play much closer to that of a human player.

6.7 UI changes for screen size

As we wanted players to be able to have varying sizes of the screen to play the games on we had to change how we originally designed the UI placement and how it worked overall. Due to this varying size, what worked one one board might be completely broken for another. This led us to develop our UI to work based on the size of the screen at any given time. Which led to a lot of different calculations needed for placement and drawing of objects and information.

7. Results of Training Ai and Testing

7.1 Ai Training Progression

In this section we have graphs depicting the progress of the Ais learning how to play both Tetris and Puyo Puyo

Fig 11. Average amount of moves made every 10 games vs games played for Puyo Puyo

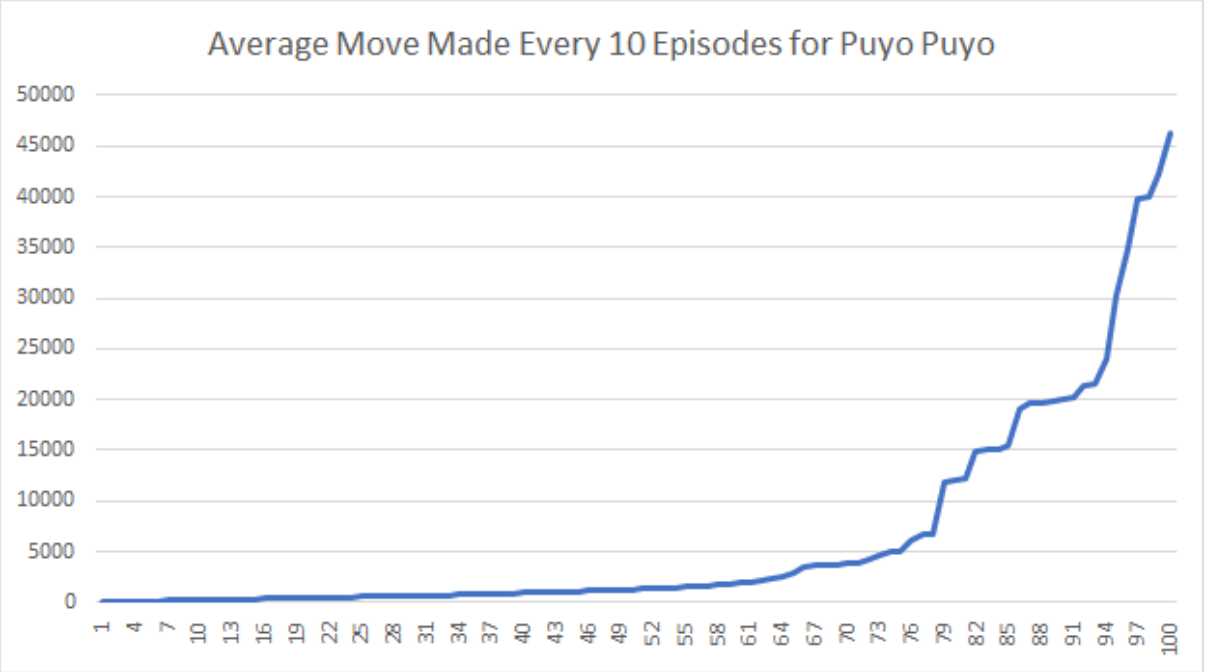


Fig 12. Average score achieved every 10 games vs games played for Puyo Puyo

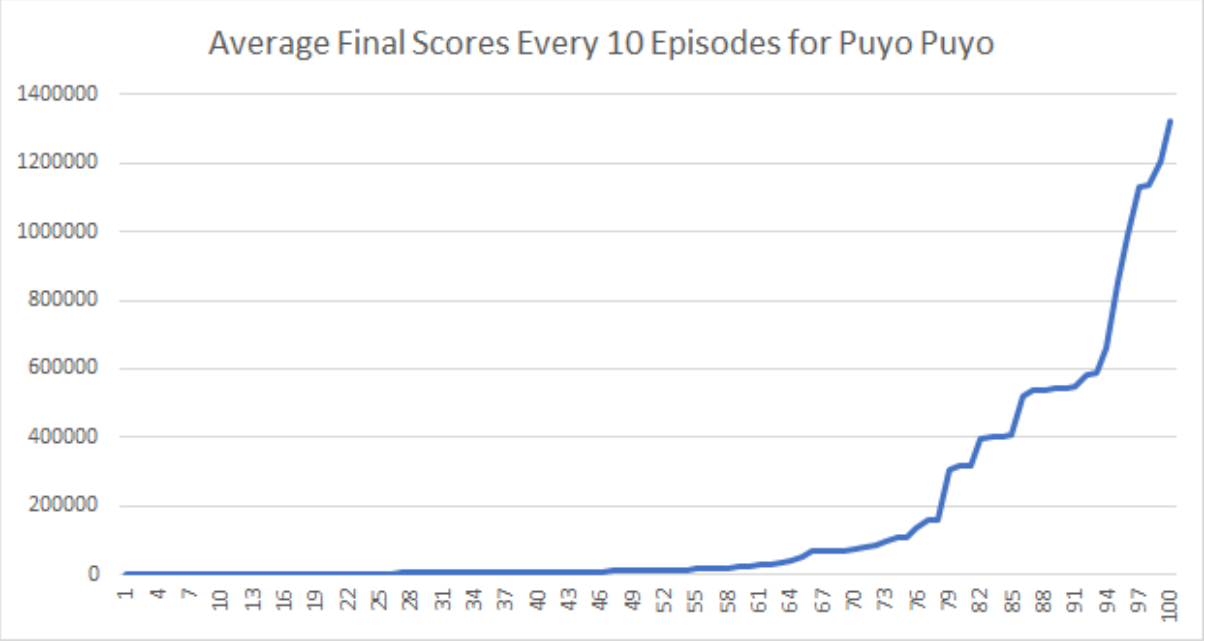


Fig 13. Average amount of moves made every 10 games vs games played for Tetris

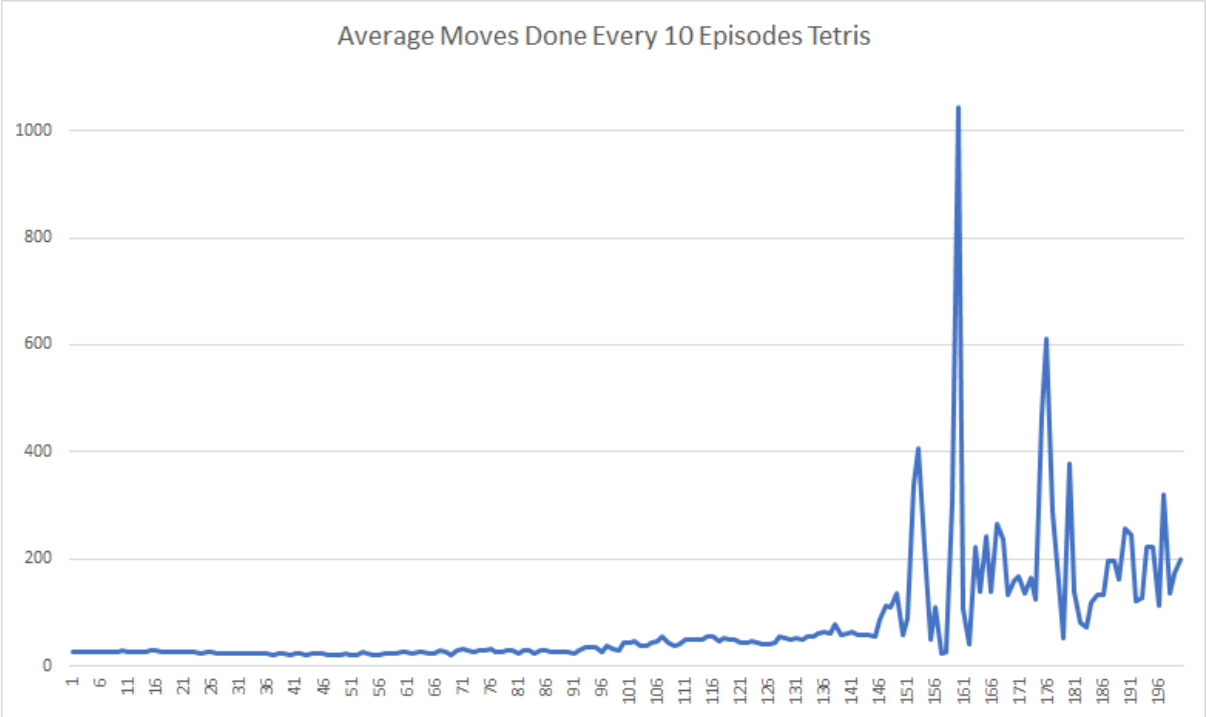
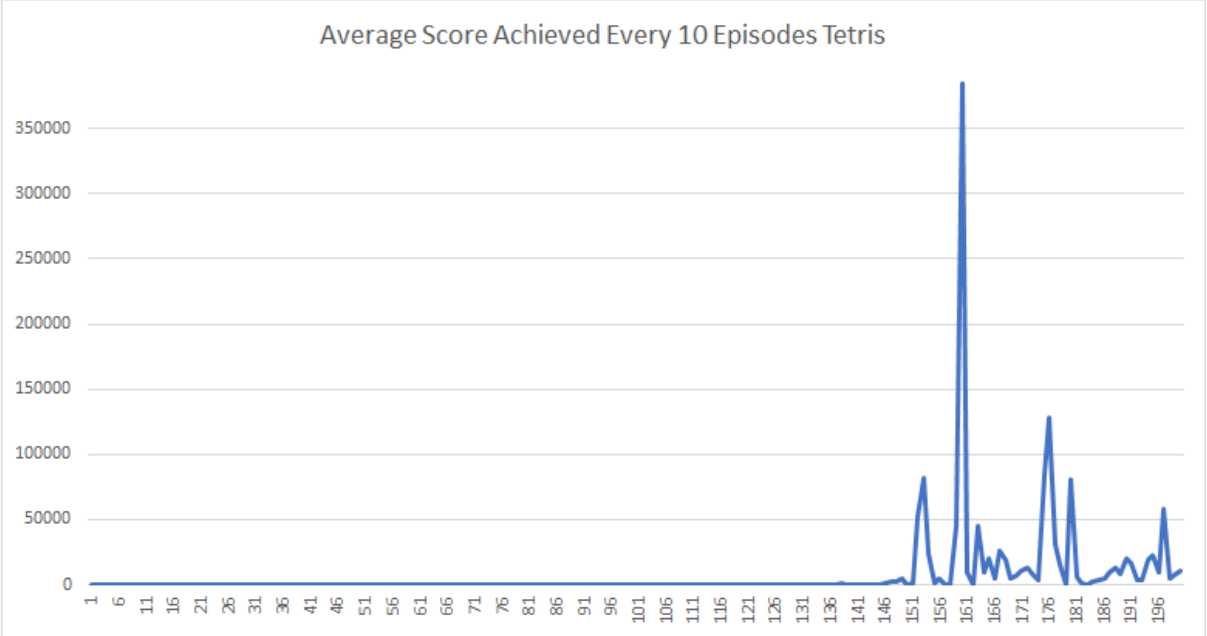


Fig 14. Average score achieved every 10 games vs games played for Tetris



7.2 Testing Results

We have written up 29 tests for the program as a whole. We've split it between the Game side and the AI side. For the games side of the testing we wanted to focus on the game's functionality and if the parts worked as intended and less on the game's GUI side.

We have 25 tests spread over the two games going over the main functionality of each. Each test has multiple assertions to test different states that might occur for that specific function. All of our tests pass and show that the games we have built function how we have designed them during our original development process.

We wrote 4 tests to test how the functions we wrote to parse each game's board. Each test also has multiple assertions to test varying information that we will feed it. All of these tests pass and show that we parse every board correctly and we're feeding the Ai correct information.

8. Future Work

One of the first things we would like to improve if we had more time would be the look of the GUI. We would like to improve upon the functional version of the GUI we currently have.

We would like to give the program as a whole a proper background for each different screen. We also want to improve on the Overall look of the GUI making it more consistent to give it a more up to date look along with having the buttons move when interacted with by players.

Another part to the GUI improvement would be to have better visual queues for the player in game with pop ups and also better sound design to enable sound response to scoring in the game and such.

Another part of the game we would like to improve would be to create and enable an online multiplayer for the game to enable users to play with one another on separate machines. We would want to do this to allow a wider range of replayability and better usability for players as they can use their own computer.

Another thing we would like to optimise if we had more time is the loading times for various pieces of our software. Currently due to our program needing to load the various pieces of the Tensorflow library it causes the loading times whenever an Ai is needed to be longer than we would like it to be. We would like to research how we could optimise these load times when it comes to this library.

Along with this we would like to make improvements to our Ai. Currently our Ai is far from perfect, sometimes it makes moves that a human would deem incorrect and we would like to improve on these flaws. Especially when it comes to the Puyo Puyo Ai, there are various improvements we can make to it that will hopefully give it more functionality such as giving it more data points to judge movements off of. Another thing we can improve on is optimising how fast it can make decisions. Mainly in training there was time where the Ai was failing due to how long it took to make a decision. Optimisations such as optimising how we parse all the information the Ai receives and feeding the Ai each boardstate at the same time instead of one by one are just a few examples.

Another improvement we would like to do would be do a lot more scouring and testing for bugs and to remove/fix as many as is possible. We know that it's highly unlikely that a large amount of code has no bugs in them but with more time would want to find and fix as many as we could.

The last thing we would like to work on in the future is implementing a thread safe queue system that isn't Python's inbuilt Queue system. Over the course of this project we discovered that Python's inbuilt Queue system can be quite slow, and in a project based around getting information too and from our Ai and various pieces of our systems as fast as possible is quite a big detriment. This lack of speed is mostly noticeable when an Ai is playing Tetris in our program. We have tried various solutions such as an open source library called Quick-Queue but unfortunately it wasn't compatible with our system. In conclusion we would like to develop our own thread safe queue system that would allow us to fix various speed issues in our program.

Bibliography

1. “Artificial Intelligence A Modern Approach” by Stuart Russell and Peter Norvig