Artificial Intelligence

Curious Agents, Reinforcement Learning in Collaborative Exploration of Unknown Environments

Hugo Prata[1 [2014198526]]

FCTUC, Faculty of Sciences and Technology of the University of Coimbra, Polo II, Rua Sílvio Lima, 3030 Coimbra, Portugal {hprata}@student.dei.uc.pt

Abstract. In the study of Curious, Intelligent Agents within A.I., we explored the hypothesis that it's important to simulate our agents within a semi-realistic environment, as a form of stimulating realistic behaviours of curiosity. Our experiment was built around the concept of games not only being viable, but preferable environments for the study of curious behaviours in A.I.

We took into account previous works[1, 5, 6, 12] in the field as guidance on our endeavour to create a world where teams of agents are able to explore diverse environments, with goals such as gathering resources, finding paths, and beating the map.

There are many instances of research on the field of A.I. where one has created an agent that attempts to beat a game. Such an instance would be MarI/O[1], a program made of neural networks and genetic algorithms, that seeks to beat a popular retro game by Nintendo, Super Mario World[2]. In this specific simulation, the agent clearly doesn't know how to play at start, not even knowing how to move, but by trial and error and after numerous simulations, the agent becomes capable of beating all levels proficiently.

We settled on utilizing Reinforcement Learning and Curious Agents while taking inspiration on MarI/O and a second project that created a Snake game[11]. We also took this opportunity to deepen our knowledge on game creation, as we created a game from scratch to serve as the environment for our agents. In this paper, we present our methodology and showcase the results from the study of the behaviour of teams of curious agents as they explore our virtual world.

The other goals of this paper are to further push the development of available A.I. simulation techniques, to enhance the simulation of "curiosity", and to bring more attention into this specific field of study by creating a relatable and interesting interface that will peak the curiosity of any newcomer in the field of Artificial Intelligence.

Keywords: Artificial Intelligence A.I. · Reinforcement Learning · Deep-Q Learning · Deep-Reinforcement Learning · Curious Agents · Collaborative Exploration · Simulation through Games · Game Interface · Game Development · PyGame Library · Interaction and Influence · Behaviour Dynamics

1 Introduction

In the field of Artificial Intelligence, there are many instances where we are faced with the need to generate and analyse large amounts of data as a means to understand the *"why"* and *"how"* of specific agent behaviours. But these algorithms can also learn for themselves if taught to be curious, more so if using a Reinforcement Learning algorithm. This specific branch of A.I. allows the creation of "curious" agents that are totally autonomous in their learning process. This is defined as form of Unsupervised Learning. We believe that the mix of Curious Agents, Environment Exploration, and Reinforcement Learning is synergetic, and so we decided to explore all the previous notions within this report. In this introduction we try to elaborate on them:

Exploration is a theme that has high affinity with the concepts of Curiosity, Reinforcement Learning, as well as Games. On the other hand, a game environment allows for simulations where agents are able to display behaviours very expressive way, capitalizing on the fact that exploration is a widespread, beloved game genre in and of itself. With this, we hope to analyse how different A.I. driven agents will interact with different maps through a game-like interface. We intend to create a diverse and interactive environment from scratch. This environment would then be explored by one or several Agents, each with unique sets of skills which influence how each agent explores and contributes to beat the map.

As previously stated while we took inspiration on MarI/O, we attempted to improve upon the MarI/O experiment by simulating our curious agents within a dynamic RPG-like[3] environment, where resources, biomes, and mobs are randomly present, and where agents need to deal with unexpected situations. In contrast to this, Super Mario is a static and linear Platformer[4], where all enemies and resources spawn on the same locations.

By creating a more elaborate and randomized environment, we enable our agents to explore and beat maps with numerous combinations of agents and strategies.

Finally, our team is passionate about games and game creation, so we took this opportunity to not only deepen our knowledge on A.I., but our game development capabilities. We also have high expectations on contributing to further inclusion and development of A.I. aspects in videogames as a means of evolving both fields.

The remainder of the report is structured as follows: Section 2, State of Art, introduces the common methodologies employed at the time. Section 3, Materials, enumerates our tools and previous research. Section 4, Methodology, describes our procedure and scientific approach, while detailing the decisions made to implement the problem at hand. Section 5, Experimental Results, elaborates on the main results of the experiment, and Section 6 is where we analyse and discuss the results of our simulations. Finally Section 7 gathers our main conclusions.

2 State of Art

Curious Agents and Reinforcement Learning

Reinforcement Learning is a popular branch of artificial intelligence. It involves creating agents capable of independently deciding on an ideal behavior, based on changing requirements[15]. This training method rewards desirable behaviors and/or punishes undesirable ones. In general, a reinforcement learning agent is able to perceive and interpret its environment, take actions and learn through trial and error. All of this allows an agent to handle uncertainty, and adapt to complex environments such as video games. On Curious agents, the idea of Curiosity-Driven learning is to build a reward function that is intrinsic to the agent (generated by the agent itself). It means that the agent will be a self-learner since he will be the student but also the feedback master[16]. Effectively, emulating the behavior of a curious human in an algorithm enhances the machines self-directed learning. All the references used in the making of this project are available at the end of this paper, in the References chapter.

3 Materials

As stated previously, in this section we will list the tools used in our project and on which previous references we based our own project.

3.1 Python Language

We decided to use Python for our project due to the fact its a high level, general-purpose and object-oriented language that is widely used and easy to learn the basics of. Python also has numerous libraries that can be used to assist in A.I. learning and statistical analysis of large amounts of data. In the following subsections we elaborate on the libraries used for our project.

3.2 PyGame Library

We decided to use Pygame to create our game environment due to its ease of use. Pygame is a cross-platform 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. It ensures better code readability with lesser code lines and better design, and it's an easy way of creating a simple game fast and efficiently. Finally, Pygame runs on nearly every platform and operating system.[7]

3.3 Sklearn Library

We decided to use the Sklearn Library as it provides a large variety of learning algorithms that will be necessary in the making of this project.

3.4 PyTorch - Reinforcement Learning DQN

PyTorch is an optimized tensor library primarily used for Deep-Q Learning. It is an open-source machine learning library for Python and is one of the more widely used Machine learning libraries[8], others being TensorFlow and Keras. We chose PyTorch because its fast, flexible, and easy to write for [9, 10]. It's available at https://pytorch.org at the time of the creation of this report.

4 Methodology

In this section we will explain the basic architecture of our game, the classes created and the resources used. We will also go through the different biomes and give an overview on how they affect the players. So as to differentiate between game characters, we will refer to the team of agents that are our main test subjects as the "players", and any other characters in the game as "NPC"'s.

4.1 Game Architecture & Environment

With the help of the python library PyGame, we were able to create a top-down, grid-based map, where each grid is 32x32 pixels. The size of the map itself has no limits, but the basic map has a size of 64x64 grids. All the configurations are set in the config.py file. It's not possible to view the entire game at once, but the size of the game window can be changed in the config.py file. Use the arrow keys to move the window. At the start of the game, a matrix of biomes is loaded into the game alongside all the game resources(textures, sprites, stats, etc) and the map grid is drawn.

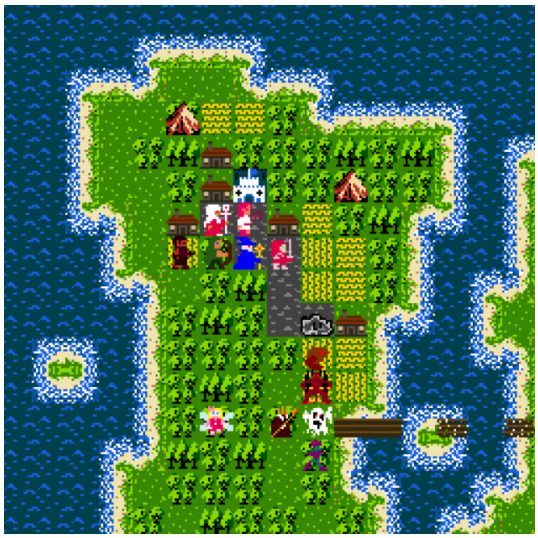

Fig. 1. Preview of the Starter Village area, with some extra NPCs and all player classes

The Biome Classes: A Biome depicts the environment of a certain grid. Certain biomes act as obstacles, with some being impossible to pass through, such as rivers and mountains. Other biomes would grant bonuses to the agent, like roads allowing for faster travel. Some biomes are even hostile to the player, such as deserts and swamps. The map itself is surrounded by water, so the environment has natural borders. No optimal path will be set right from the start. Blocking the player from threading certain tiles is done with an invisible wall class that can slow or stop the player movement.

For easier pseudo-randomization, we divided the map into six biome quadrants. These quadrants then scale in hostility towards the players following an "U" shape. While the green biomes have lots of weak monsters to train on, and villages to heal, the Desert biome will have no villages, and tougher monsters will roam the dunes. When arriving at the swamplands, players will be met with full hostility, there will only be undead waiting for them, within their lairs.

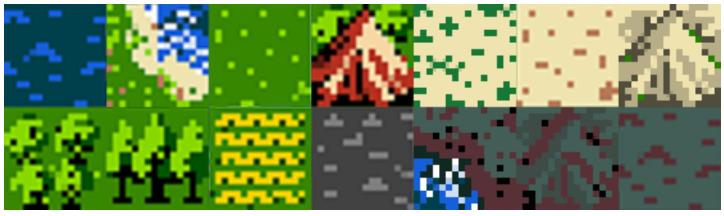

Fig. 2. Preview of our biomes: water, coastline, grasslands, mountain, semi-desert, desert, dune, forest, crops, road, swamp-coastline, bog, swamplands

- water - impassable biome, unless there is a bridge over it;
- coastline - neutral biome (several variants). These shallows can be crossed;
- grasslands - neutral biome;
- mountain - impassable biome;
- semi-desert - neutral biome;
- desert - hostile biome;
- dune - impassable biome;
- forest - neutral biome (several variants);
- crops - neutral biome (several variants);
- road - pathway biome;
- swamp-coastline - hostile biome (several variants) These shallows can be crossed;
- bog - impassable biome;
- swamplands - hostile biome;

The Building Classes: A Building depicts a place of interest on a certain grid. Buildings can be used for things such as crossing a river (bridges), hint at paths or bonus NPC's, or give "treasures" directly to the Player. Reaching certain buildings (such as all Evil Castles) is a requirement to finish a map, and

gathering a building yields the player positive points. In the context of our game, to beat a map the player will need to find the building where the boss resides, and then beat the boss.

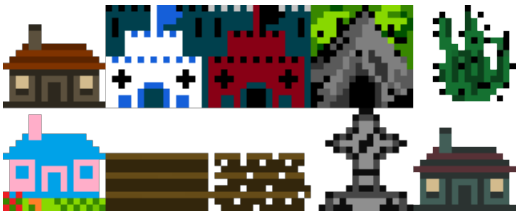

Fig. 3. Preview of our buildings: Village House, Castle, Evil Castle, Cave, Cactus, Faerie House, Bridge, Broken Bridge, Ancient Milestone, Necro House

- Village House - Friendly villagers will slightly heal a player, +health;
- Castle - Friendly knights will slightly train a player, +exp;
- Evil Castle - Summons the boss enemy and its minions, destroy them to win;
- Cave - May have treasure, may spawn enemies;
- Cactus - Found only in the desert, heals players;
- Faerie House - May have a Faerie living within;
- Bridge - allows players to cross rivers;
- Broken Bridge - uncrossable point;
- Ancient Milestone - hints at the right path;
- Necro House - Undead have no need for supplies or shelter;

The Player Classes: We decided to create several different classes of "player" agents, with unique specializations. These then interact with the game environment in their own way. Preferably, these agents will team up to beat the map, although simulations with one agent are possible. Furthermore, we wanted to create "NPC" agents that will either attack or help our "Player" agents.

Fig. 4. Preview of our Player Classes: Thug, Squire, Archer, Knight, Priest, Wizard

- Thug - low level starter class;
- Squire - mid level class, emphasis on tanking;
- Archer - mid level class, emphasis on power;
- Knight - high level class, emphasis on tanking;
- Priest - high level class, emphasis on healing;

– Wizard - high level class, emphasis on power;

The NPC Classes (split between Enemy and Bonus): As previously stated, we wanted other agents to interact with our player agents, so we created NPC agents. These agents will either help or attack our player agents, and are divided between Enemy and Bonus NPC's. Enemy NPC's will simply attack our players, while Bonus NPC's, when found, will give the players a gift, whose scale is correlated to the rarity of that NPC. This means a Villager will be more common than a Faerie, but the gift will also be smaller. Finding these NPC's will always be a net gain, while dealing with enemies will involve several strategies, such as running away when facing stronger enemies or when the player health is low, or fighting weaker enemies to gain exp and grow stronger. Finally, we created a neutral NPC that can either attack or yield gifts when found. We hoped this would add an additional random factor when dealing with NPC's.

Fig. 5. Preview of our Enemy Classes: Slime, Beast, Ghost, Zombie, Necromancer

– Slime - very weak enemy;
– Beast - mid level enemy;
– Ghost - high level enemy;
– Zombie - high level enemy;
– Necromancer - very high level enemy, boss, kill them to win;

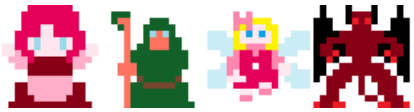

Fig. 6. Preview of our Bonus Classes: Villager, Gnome, Faerie, Demon

– Villager - friendly npc, small bonus when found;
– Gnome - friendly npc, medium bonus when found;
– Faerie - friendly npc, large bonus when found;
– Demon - neutral npc, 50% chance to either gift or attack players;

We are very satisfied with the end result of our game development. We were able to create a large world in a short period of time, with fully functional RPG mechanics. A preview of the world is made available in the following page.

The players start at the north-western village, and need to travel in an "U" shape to reach the Bosses. It's very important to gather resources and train, or else the Bosses will simply be too powerful to beat.

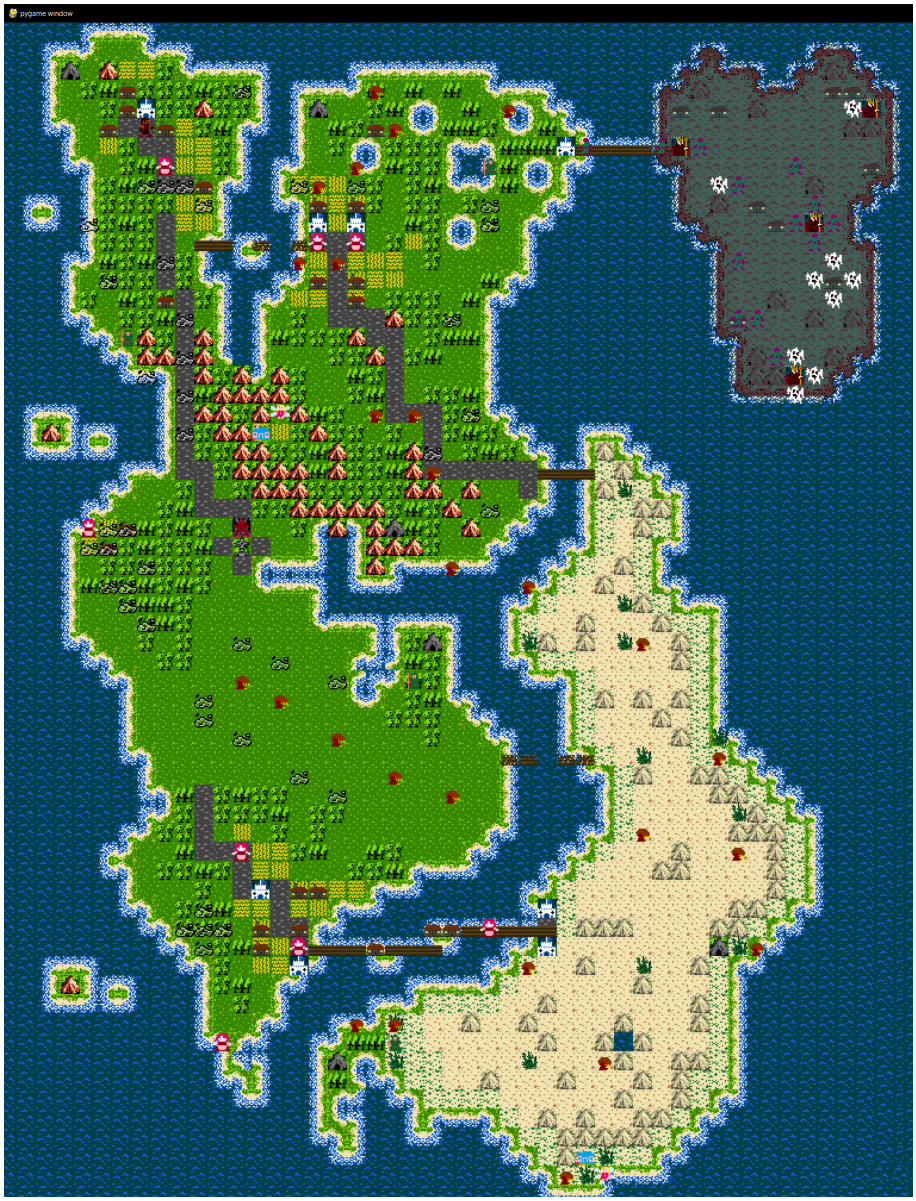

Fig. 7. Preview of the entire map of the game

4.2 Readying the Environment for the A.I.

Our first approach into developing the agent A.I. was to research on how to ready the game environment for the needs of the A.I. Since our game approach was modular, we only had to do very minor alterations to the game code to allow the A.I. to roam our environment through the player sprites.

Our first consideration was on what the A.I. would do: Start, Play, End. We already had a Start and Reset method at this point, so our focus was on when the A.I. would need to use them. When we run the A.I., it automatically starts an infinite loop of start, play, end. Our game already ends when all objectives are met. But the A.I. will not know how to reach these goals at start. So we decided to add a timer and score which will be used to calculate progress. Should this progress stall, the game resets. The score is the total amount of experience gained by all agents. The timer is a rough estimate of playtime in seconds.

Fig. 8. Preview of the game timer and the total player score

Our performance formula makes it so that, after a minimum time tolerance, the A.I. quits on the following cases:

- if the playtime is double the score;
- if the A.I. takes more than the maximum time allowed to beat the game;
- if the A.I. can't keep gathering score at a good rate;
- if the A.I. score goes below zero;

```
def performance(self):
    maximum_playtime = 300  # +/- 5 minutes
    minimum_playtime = 10   #
    # if playtime is over minimum and one of the other conditions is met:
    if self.total_time > minimum_playtime and \
            ((self.total_time > self.total_score) or
             (self.total_time > maximum_playtime) or
             (self.total_time > maximum_playtime / 2 > self.total_score) or
             (self.total_score < -5)):
        self.performance_quit = True
```

Fig. 9. Progress Formula

Note that, should all players die the game will also end, regardless of time tolerance.

Should the A.I. reach a goal, it will get rewarded. And should it reach all the goals and end the game successfully, an additional reward is given. Dying and losing the game results in negative rewards. Since everything in the game rewards a certain amount of experience, this entices the A.I. to explore as much as possible, in an efficient way.

Our second consideration was on the A.I. movement. At this point the player movement was done with the AWSD keys, and all players moved exactly the same. To adapt this to our needs we changed the movement function within the Player Class to receive a direction instead of a keypress event. Secondly, we moved our game loop from within the Game_Session Class into our new Agent Class which we will elaborate in the next chapter. This allowed the Agent to send directions to the Player Sprites within the Game Session and also synchronized the game iterations with the agents. Having done all this, we ran a simple demo with a random direction generator, allowing us to move all three players independently through our Agent Class. We then proceeded to the development of the Agent and A.I., which we will elaborate on in the following chapter.

4.3 A.I., Agent Model, and Reinforcement Learning

We created a total of two files for the AI side of our project. The Agent class holds the several methods required for our simulations. Each Agent is assigned a player, and there can be several agents running at once. The methods within the Agent Class handle the following issues: Agent State, Agent Action, Agent Memory, and Agent Training (short/long term). We will elaborate on these methods shortly.

Within the same file we have a method called Run_AI, which has three main jobs: Running the game sessions, assigning Player characters to our Agents, and calling the several methods within the Agent Class.

Our other AI related file is the Model file. This file holds the Neural Network Class, which in our case is a pytorch linear network, "Linear_QNet". This file also holds a "QTrainer" Class, through which we train our Neural Network.

Agent Class Variables:
Our Agent Class holds the following game-related variables which allow the Agent to play the game:

- An instance of the current Game Session;
- An instance of the assigned Player character;
- A Boolean which tells us whether the Agent is still playing the current session;

Our Agent Class also holds the following A.I. related variables used in our model:

- epsilon, which holds the randomness value, used in the Action method;
- gamma, which holds the discount rate value, used in the Model method;
- memory, which is used to memorize what is learned by the agents;

- model, which creates a linear neural network of several layers, with user-specified input, output and hidden layer sizes, through Linear_QNet();
- trainer, which trains our model, with user-specified learn rate and gamma, through QTrainer();

The final two, model and trainer, are classes stored within our Model file.

Agent State:
The Agents' State is one of the most important pieces of our Model, and is also one of the most complex. It holds all the stimulation's on which the Agent bases its actions. We use several methods to keep the state accurate, as our state has over fifty variables within it. We can divide the variables within our state in three: player status variables, direction variables, and location variables. Each individual variable is of type Boolean up until the state array is to be returned, at which point Numpy is used to efficiently convert the state array into a binary array. Elaborating on each of these variables:

The status variables hold basic player information that is needed to influence how the agent behaves towards itself and other agents:

- is_alive, tells the Agent whether it is still capable of playing or not;
- low_health, warns the Agent that it's characters' health is low;
- full_health, warns the Agent that it's characters' health is full;
- is_fighting, warns the Agent that it's fighting an Enemy;
- is_healer, tells the Agent that it's class is capable of healing other Agents;

We hoped that adding these variables would help the agent deal with surviving the fighting aspect of our RPG game.

The direction variables simply hold the direction the Agents' player character is facing when the state method is called. The possible directions are left, right, up and down. The agent cannot select more than one direction at once.

Finally, the location variables are how our Agents understand the environment around them. Each Player Class has an inherent range of view, with higher classes having a longer range. This range limits how far an Agent can see around it. There are two types of location variables, interests and blocks. Interests use the Agents' character class range of view, while blocks are a static range of 2 game tiles. Each Agent can have up to 25 interests and blocks at once, for a total of 50. Should there be less than 25 of each, padding is used to assure each tensor is of the same length. Since these variables involve movement, there is a need to consider coordinates and type.

For blocks, we compare the coordinates of the block and the agent. This amounts to a total of four Booleans, which are saved within the state, before the Direction Variables. With these, we hoped to allow the Agent of being capable of avoiding impassable obstacles such as water bodies and mountains. We do not specify the type in these cases.

For interests, we repeat the coordinate process, while also saving the type of interest in scope (such as whether it's a house, a castle, a weak enemy such as slime, or a strong one such as a necromancer, etc). This adds up to a total of 23 Booleans per interest. With these, we hoped to allow the Agent to choose which interests would benefit it the most at the time of the decision, such as choosing an house to heal over fighting an enemy, when at low health. Other players are also interests, because should an Agent be a healer class rather than a fighter class, it is important to know whether other players are in need of help. These Booleans are stored after the Direction variables and are preceded by a single variables that states whether the interest array is empty or not.

```
state = [
    # STATUS
    is_alive,
    low_health,
    full_health,
    is_fighting,
    is_healer,

    # BLOCKS
    for block in known_blocks:
        is_blocked_left,
        is_blocked_right,
        is_blocked_up,
        is_blocked_down,
    PADDING

    # DIRECTION
    dir_r,
    dir_l,
    dir_u,
    dir_d,

    has_interest,
    # INTERESTS
    for interest in known_interests:
        # interest coords
        interest_is_left,
        interest_is_right,
        interest_is_up,
        interest_is_down,
        # interest type
        interest_is_other_player,

        if interest_is_other_player:
            other_player_low_health,
            other_player_full_health,

        interest_is_house,
        interest_is_castle,
        interest_is_slime,
        interest_is_zombie,
        interest_is_demon,
        interest_is_faery,
        # etc all types
    PADDING

]
```

Fig. 10. Pseudo-Code implementation of our Agent State;

The State method is called twice per iteration: Once to acquire the current state of the Agent, and a second time to acquire the new state post Agent Action, which we will elaborate upon next.

Agent Action:
Our Action Method is how our Agents decide on their movement. There are four possible choices: left, right, up and down, which are manifested by a list of integers [0,1,2,3]. The agent chooses one member of this list every iteration. There are two ways of choosing an action:

- random prediction, where the action is purely decided by chance through random.choices() (same probability for all directions);
- model prediction, where the action is decided through the agents' Neural Network, based on what the agent has learned so far. A state is passed through the Neural Network and returns a choice of direction;

Whether the Agent chooses one method over the other is also random. This is decided through a formula where the more games an agent plays, the more likely it is of making a decision based on what it's learned rather than a random decision.

```
self.epsilon = 80 - self.total_games_played
if random.randint(0, 200) < self.epsilon:
    # random move: exploration vs exploitation tradeoff
else:
    # model based move
```

Fig. 11. Formula on whether to choose random or model based decision

The lower the value of epsilon (randomness) becomes, the less likely the value of randint is to be smaller than epsilon, and therefor the more likely the decision is to be based on the model rather than randomness.

This formula is basically a decision between exploration and exploitation, where exploration means learning new actions and their benefits through random choices, and exploitation means utilizing what has been learned so far to make new decisions.

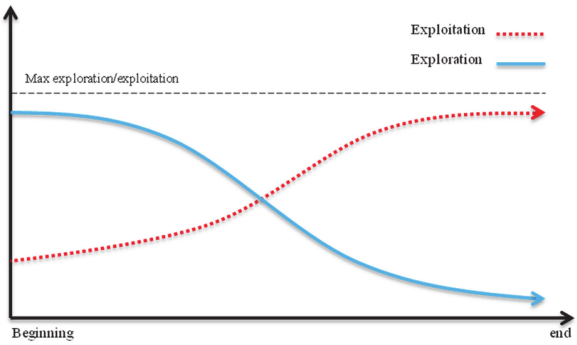

Fig. 12. Exploration vs Exploitation Trade-off[8]

This is known as a trade-off mainly because of efficiency. Since exploration and exploitation infinitely tend towards a set maximum and minimum, there comes a point where it becomes more rewarding to focus more on exploitation rather than exploration [8].

Agent Memory & Model Training:
Our Memory and Training methods are very simple and all work with the same group of variables: a previous state, a new state, an action, a reward, and the player alive state.

The Memorize method simply appends these variables into the Agent memory whose size is defined at the start of the Agent Class file, and is set to 50_000_000 in our case.

On the Training methods, both the long and short term methods call the same training function from the Model class, albeit with a slight difference: the short term feeds it with just one variable of each type, every game iteration, while the long term method feeds it lists of variables of each type, at the end of each game session. In the case of the long term Training method, the Batch Size defines the size of the random sample of memory used for long term training.

Model Linear_QNet():
This is where we initialize our Neural Network with the help of PyTorch. We settled for a linear neural network due to their ease of use and simplicity, and while we settled for a network with a total of ten layers, we will elaborate more on this choice in the Experiments Chapter of this essay. The Network Class receives the size of its input, output, and hidden layers, and has two secondary methods: one to connect each layer and pass a tensor through, and another to save the model to disk.

Model QTrainer():
This is where our model is trained. The QTrainer Class has instances of the

following values and variables used in Model Learning: Learning Rate, Gamma value, an instance of our Model, our PyTorch optimizer of choice ADAM, and our Loss Criterion.

The QTrainer Class also has the method train_step() which is called for short and long term Training by the respective Agent methods, and receives its inputs through these methods.

The train_step() is an implementation of the following formula:

Q Update Rule Simplified:

$$Q = model\,.\,predict(state_0)$$

$$Q_{new} = R + \gamma \cdot max(Q(state_1))$$

Fig. 13. Basic update rule for Q-Learning Reinforcement Learning

This formula would then be used to predict the next decision based on the values redirected by the Agent training methods, which can be single variables or lists of variables.

Our train_step() method can be divided into two parts: data handling, where we reformat the data to suit our needs, and the implementation of the previous formula using the tools made available by PyTorch. We coded the formula in the following matter:

```
# 1: predict Q values with current state
Q = self.model(curr_state)
```

Fig. 14. First line of the Q-Update Rule;

```
# while not done with session, get Q_new
if alive[idx]:
    Q_new = reward[idx] + self.gamma * torch.max(self.model(new_state[idx]))
```

Fig. 15. Second line of the Q-Update Rule;

```
# predictions[argmax(action)] = Q_new
Q[idx][torch.argmax(agent_action[idx]).item()] = Q_new
```

Fig. 16. The original Q-value is then adjusted using the new Q-value;

Due to the complexity of the train_step() method, we decided to include a snippet of the full method, for better context:

```
# this is called by long and short train(), it will receive either tuples or lists, but not a mix of them.
def train_step(self, curr_state, new_state, agent_action, reward, alive):
    warnings.filterwarnings("ignore")
    # can have multiple values per each input
    curr_state = torch.tensor(curr_state, dtype=torch.float)
    new_state = torch.tensor(new_state, dtype=torch.float)
    agent_action = torch.tensor(agent_action, dtype=torch.long)
    reward = torch.tensor(reward, dtype=torch.float)

    # if there is only 1 value per input, flatten tensor to size 1
    if len(curr_state.shape) == 1:
        curr_state = torch.unsqueeze(curr_state, 0)
        new_state = torch.unsqueeze(new_state, 0)
        agent_action = torch.unsqueeze(agent_action, 0)
        reward = torch.unsqueeze(reward, 0)

        # convert alive boolean into a tuple
        alive = (alive,)

    # AI PREDICTION:    <--------------
    # 1: predict Q values with current state
    prediction = self.model(curr_state)

    # 2: predict Q_new with formula while still playing:
    # Q_new = r + y * max(next_predicted Q value)
    # prediction.clone() to have 4 values
    target = prediction.clone()
    for idx in range(len(alive)):
        # Q_new = reward of current index
        Q_new = reward[idx]
        # while not done with session, apply formula
        if alive[idx]:
            Q_new = reward[idx] + self.gamma * torch.max(self.model(new_state[idx]))

        # predictions[argmax(action)] = Q_new
        target[idx][torch.argmax(agent_action[idx]).item()] = Q_new

    self.optimizer.zero_grad()
    loss = self.criterion(target, prediction)
    loss.backward()
    self.optimizer.step()
```

Fig. 17. Train_Step() method in QTrainer() used by long and short term training Agent methods.

This method was mainly based on a previous work by Patrick Loeber[8] and the following pseudo-code, used in a similar project involving games and Deep Q-Reinforcement Learning[18].

1. Initialize Q-values ($Q(s, a)$) arbitrarily for all state-action pairs.
2. For life or until learning is stopped...
3. Choose an action (a) in the current world state (s) based on current Q-value estimates ($Q(s, \cdot)$).
4. Take the action (a) and observe the the outcome state (s') and reward (r).
5. Update $Q(s, a) := Q(s, a) + \alpha\,[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

Fig. 18. Deep Q-Learning QTrain() train_step() pseudo code

Since Q-learning uses future rewards to influence the current action given a state, it should therefore enable our agent to select the best action for the most optimal reward.

5 Experiments

In this chapter we will go over our experimental setup and methodology. In the experimental setup we will explain how we ran the individual simulations and describe the machine they were done in, whereas in the methodology we will enumerate and explain each individual experiment done within the scope of this project.

5.1 Experimental Setup

Each simulation consists of 150 game sessions. Each game session runs while the rules set by our progress formula (Fig.9) are met. This means each session lasts between 10 seconds and 5 minutes, making the minimum individual simulation time around 30 minutes. Once the first session is started, by running the Agent Class .py file, every following session is automatically started by the agent so no further user input is needed past the first session. Should the user want to keep track of the Agents when they're not visible within the window limits, the arrow keys can be used to move the camera around the map without affecting the simulation.

Every session score is saved in a text file automatically at the end of each session. Each agent saves individual high-scores and game high-scores. An individual high-score is the high-score of a single agent. Should only one agent be used, then there's no difference between individual and standard high-scores. Should more than one agent be used, then the sum of all individual high-scores will be equal to the standard high-score. This allows the user to know which agent could outperform its peers within a group of agents. Once each game session is finished, the scores are saved within a text file, scores.txt.

The machine where these simulations were done has the following specs: Windows 10 Pro OS, 32gb DDR4 RAM, Intel i7-10750H CPU, Nvidia RTX 2070 Super GPU.

5.2 Experimental Methodology

As previously stated, in this section we will explain each individual experiment done. Each subsection is named after the factor we experimented on. Further on, we experimented not only on individual factors such as number of neural layers, but combinations of them.

Number of Neural Network Layers We experimented on the effect the number of layers given to our Neural Network had on its performance. We utilized 3, 5, and 10 layers, adding to a total of 3 experiments on the Number of Neural Network Layers.

Size of Neural Network Layers The size of the input and output layers is fixed, since they're dependent on the size of the agent state and number of choices an agent can make, respectively. We could, however, experiment with the size of the hidden layers. We utilized 256, 512, and 1024 as the size of the hidden layers, adding to a total of 3 experiments on the Size of Neural Network Layers.

Batch Size Early on in our studies, we learned that similar projects had had issues with the batch size, having successful experiments only when large batch sizes were used. To experiment on this, we decided to try several batch sizes. We utilized 100, 500, 1000, 10000, and 25000 as batch sizes, adding to a total of 5 experiments on Batch Size.

Learning Rate Learning Rate is one of the most important factors when setting up these kinds of experiments. The learning rate determines the step size at each iteration while moving toward a minimum of a loss function. In setting a learning rate, there is a trade-off between the rate of convergence and overshooting. To analyse the effect learning rate had on our Agents, we decided to test the following values: 0.5, 0.1, 0.01, 0.005, 0.001, and 0.0001, adding to a total of 6 experiments on Learning Rate.

Discount Rate The discount rate, or gamma, was also a target of our experiments, although it was given less attention than the other factors. We tested values of 0.5, 0.7, and 0.9, adding to a total of 3 experiments on Discount Rate.

Agent Action Decision Formula We also decided to test the hypothesis of whether the amount of random actions needed to learn effectively were adequate. For this, we made changes to the formula that chooses between model and random based decisions (fig.11) by changing the initial values of epsilon and the range of values given to randint. By doing this, we were able to make the agent converge to model-based actions earlier or later, by changing the default values that were given at the start of the project.

Agent Rewards Rewards are one of the main emphasis of Reinforcement Learning. We experimented on both how the rewards were given to the Agent, and the values of said rewards.

Our first experiment saw the rewards being the same value as the "experience" stat of the Agents, so when an Agent completed an action that rewarded experience, then that would also count as a RL "Reward". In this case the only negative rewards would come from losing the game, through the performance formula or dying, and by continuously attempting to cross over impassable biomes such as water bodies and mountains.

The second experiment on Agent Rewards saw us removing the negative rewards for impassable biomes.

In the third experiment, the negative rewards for impassable biomes returned, and we also added a negative reward for actions that yielded no reward, such as walking over the same spot continuously.

In the final experiment on Agent Rewards we changed the values of the rewards given to the agent to a fixed amount instead of an amount equal to the "experience" gained by the Agent. We tested this with fixed amounts of 1, 5, and 10.

Agent Number While we focuses mostly on single agent experiments, we designed the game to be played by groups of agents, so we also experiment on the effect of having several agents playing at the same time. We experimented with groups of 3, and 5 agents. This was done to test the behaviour of Agents as a group.

Agent State We made several changes to the variables within the Agent state. The variables that remained unchanged were the Player Status variables (ex. is_alive, is_low_health, etc), the Agent Directions, and the Interest Directions.

We then made experiments on the range of sight, which influences how many interests an Agent can see at once, and on how much information the agent has on said interests. Early on, no information was given, while as the state was further developed, we decided to add variables which tell the agent the sprite type of the interest (ex. if it's an enemy, friend, a building, etc), and eventually we settled on adding all possible sprite types to the state (ex. slime, beast instead of enemy). This was done to test the behaviour of Agents towards their interests.

We also changed the way blocks were dealt with multiple times. At first, the state simply had four variables which were toggled depending on whether the agent had collided with a block, in a certain direction (ex. blocked_right). After completing the Interest function we decided to apply the same method to the blocks, albeit with a fixed range of 2 game tiles. This means the agent would also know where all blocks within 2 game tiles are, instead of toggling a variable when touching a block. This was mainly done to test the behaviour of Agents towards blocks.

Taking into consideration the possibility of using a group of Agents instead of single Agents, we also added other Agents as possible interests, along with their status and a variable "is_player". This was combined with multiple Agents to test the behaviour of one Agent towards the other.

Combination of Previous Experiments

We decided to combine the previous experiments, giving more importance into testing different values of Batch Size and Learning Rate together. We used the more complex Agent State at this stage. We also used all the different Rewarding values and methods we came up with in previous tests. There was a focus on Single Agent simulation in this phase of the project, gamma was fixed at 0.9, and the learning rate used was always equal or under 0.01. All previous batch

sizes were tested again, and the number of layers was fixed at 10. The size of these layers was also fixed at 1024.

6 Experimental Results and Comments

Due to the amount of tests exposed in the previous chapter, we decided to divide the test results between groups based on their effects on the final game scores. We divided them in tests with none, average, and large effects on the score of the simulations.

6.1 Tests with no effect on Results

The experiments with negligible effects on the end score of the simulations were: "Size of Neural Network Layers", "Discount Rate", "Batch Size", "Number of Neural Network Layers", and "Agent State". In these cases, there was no noticeable improvement on the end-score of the simulations compared to the base simulation results when using any of the values or settings enumerated in the previous chapter.

The Agent wasn't able to finish the game, and the average points were always negative, with sparse outliars where the agent was able to sometimes acquire points closer to zero through randomness.

6.2 Tests with average effect on Results

The experiments with average effects on the end score of the simulations were: "Learning Rate", "Agent Action Decision Formula", "Agent Rewards", and "Agent Number". In these cases, there was some improvement on the end-score of the simulations compared to the base simulation results.

The Agent wasn't able to finish the game, but the average points became positive. Some Agents were able to level their player classes, and reach as much as 150 individual points. Some Agents managed to successfully kill and gather some NPC's.

We believe these results were more thanks to the increase in the agents' gathering potential or number , rather than an improvement in learning capability of the agents.

In the case of Learning Rate, at the lowest value used (0.001), the agents' seemed to show some traits of having learned some actions such as always choosing to go for the castle first before anything else, but were still unable to reliably complete basic actions and would still get lost. There were no changes when using higher learning rate values. This leads us to believe our model is overfitted in its' current form.

In the case of Agent Action Decision Formula, the improvement in results mostly came from having a larger number of game iterations with random movements, which resulted in the Agent covering a wider area and gathering more

points through randomness.

In the case of Agent Rewards, the changes were noticeable, but we believe that these changes were mostly due to randomness. By lowering the Reward values, or removing negative rewards, the behaviour of the agents remained similar, as well as the gathering potential, but due to the changes in the value of the Rewards, the end score changed as well.

We found that the use of different Rewards seemed to affect agent decision slightly, but we are unsure whether this was simply a random occurrence or if the Agents did behave differently and then their memory was flushed with later iterations.

By having negative values on block collisions, the Agents seemed to avoid blocks more often than when there were no negative rewards. This could be seen when the Agents would start walking towards the sea but would always stop one tile short of colliding with it and walk several tiles while avoiding the blocks, up to the point they'd gather a positive reward, which would then cause the Agents to ignore the following blocks and start to lose points through block collisions.

By rewarding the Agent every iteration when colliding against enemies, the Agent also seemed to become very aggressive and fixated, and to die very quickly, thus not gathering many points.

In the case of Agent Number, the improvement in results was mostly due to having more Agents. Once again, a wider area was covered and through a higher gathering potential a higher point average was achieved, but again this was mostly a result of randomness. We were unable to see any sort of cooperation, since no agent managed to evolve to the point of allowing a healing player class to be present in a game session.

The best results were achieved through combining groups of Agents with higher reward factors, and to prolong the use of random decisions while using a very low learning rate. This resulted in two out of three Agents managing to evolve into second tier classes (Squires) and yielded a total score of 319, after 61 game sessions.

6.3 Tests with high effect on Results

Unfortunately, we were unable to produce any high effects on the end results.

6.4 Final Comments

We compared our test results to an initial run so as to try to make any improvement as noticeable as possible.

We can come to the conclusion that the Agents were able to show some minor signs of learning due to the fact that, once the decision formula converged to model based decisions, the Agents would commonly fixate on certain paths and actions.

A common occurrence would be that the Agent fixates on the path where it gathers the starter Castle, runs over houses to heal, and then gathers the Cave on the upper left corner of the game area, just to then be unable to find it's way anywhere else. Another common occurrence would be the Agent fixating on going left after more than 60 sessions, just to lose all the following sessions through gathering negative Rewards by colliding with the sea. A third common occurrence would be when the Agent decides to walk south and fight every monster until it dies for several sessions in a row.

The agents were, however, unable to innovate their decisions. This makes us believe that our model is either overfitted, not trained well enough, or that the model architecture (linear R.L.) might be inadequate for the complex decisions we expect of our Agents. It could also be the case that there is a problem with the Agent memory or the method of updating it. More time would be needed to make sure that the model is in an adequate state and that there is no problem with the Agent methods.

7 Conclusion

During the making of this project, our group was able to experiment on and develop several different sets of skills, from A.I. development to game development, as well as algorithm optimization and graphic design.

Although the results of the experiments were mostly unfavorable, with our Agents being unable to effectively learn how to finish the game, we were able to meet several of our goals, having created a game environment from the the ground-up, to developing a group of working A.I. Agents capable of playing the game we created, and experimenting on learning algorithms.

We hope to further develop this project in the future, on a better time-frame, and by experimenting with other types of learning algorithms network types, such as recurrent and convolutional neural networks. We also want to elaborate on the agent system we created by adding intelligence to our static NPC's, and to add more random factors to our game such as randomized positioning of Agent interests and biomes. We also hope to experiment with other different layer architectures, and on a general note, to find the reasons for the previous unsuccessful results.

We are grateful to have been granted the opportunity to work on this project, and for the liberties in decision-making given to us. We have high hopes of enrolling on and completing future works within the field of Artificial Intelligence. We also hope to apply the knowledge gained from this project on our future endeavours, and to help widen the general use of A.I. throughout Computer Science and Engineering as a whole.

References

1. http://pastebin.com/ZZmSNaHX
2. https://en.wikipedia.org/wiki/Super_Mario_World

3. https://en.wikipedia.org/wiki/Role-playing_game
4. https://en.wikipedia.org/wiki/Platform_game
5. https://www.cisuc.uc.pt/download-file/13470/MS6EczRIvif99eYej3X7
6. https://github.com/SebastianRehfeldt/collaborative-exploration
7. https://www.pygame.org/wiki/about
8. https://www.toptal.com/deep-learning/pytorch-reinforcement-learning-tutorial
9. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
10. https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html
11. https://github.com/python-engineer/snake-ai-pytorch
12. Learn, understand, and develop smart algorithms for addressing AI challenges, Andrea Lonza
13. http://www.dcsc.tudelft.nl/ bdeschutter/pub/rep/10_003.pdf
14. https://arxiv.org/abs/1911.10635.
15. https://openai.com/blog/reinforcement-learning-with-prediction-based-rewards/
16. https://people.idsia.ch// juergen/interest.html
17. https://www.researchgate.net/figure/Change-trend-of-exploration-and-exploitation-from-the-beginning-to-the-end_fig1_324731872
18. https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/