Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

ROGUEINABOX: A ROGUE ENVIRONMENT FOR AI LEARNING

Framework development and Agents design

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
GIANMARIA PEDRINI

Sessione I
Anno Accademico 2016/2017

To Nives and Sergio

Introduction

When it comes to using a computer to solve problems that where initially targeted at human beings two options come to mind; an hard-coded agent and a Machine learning agent. The main difference between these two approaches is that the writing of an hard-coded agent requires the programmer to tell the agent what to do in every situation, while building a machine learning agent is about giving the program the correct ”brain” structure and telling it what is its target. While an hard-coded agent can be easier to implement and can be able to perfectly solve simpler problems, the harder the problem the harder the challenge for the programmer. He has to come up with a solution for every possible scenario which also requires a deep understanding of the underlying problem. On the other hand while it might be difficult to find the correct setup for a machine learning agent, the ability to automatically learn problem solving from data or experience is extremely valuable and might lead to better automated solutions in harder problem.

This thesis is about a particular branch of Machine learning: Reinforcement learning, and in particular DeepQLearning. Reinforcement learning is an automated learning technique in which an agent performs actions in an environment and for each action receives a reward (positive or negative). The agent ”brain” (Network) then learns from these rewards and improves itself consequently. Using its learned knowledge the agent is then able to predict which action is best in every situation (state) using its QFunction; this function is approximated by the network and gives its name to the learning method. The Deep prefix simply means that the network used is structured in multiple layers and is therefore deep.

Reinforcement learning can be used to tackle different kind of problems but one that comes to mind, where humans also often learn by trial and error, is games. Games are interesting field of study because they are complex, offer a wide array of challenges and require the development of different skills which often can be transferred to real world problems.

When it comes to research and experimentation the flexibility and effectiveness of the frameworks used is crucial. The main choice, given a problem to tackle and an environment for it, is which machine learning framework to use. This choice also limits the choice of programming language that will be used.

For this project Keras [1] was chosen as a machine learning framework, supporting both Theano [2] and Tensorflow [3], it stands out for it simple prototyping and ease of use. Consequently the language this project is written in is Python.

Many game environments suitable for Reinforcement learning already exists, most notable examples are Arcade Learning Environment (ALE [4]), OpenAI Universe [5] [6] and VizDoom [7]. Frameworks for interacting with roguelike games also already exists, such as [8] for Desktop Dungeons and BotHack [9] for NetHack, but none was available for the game of choice of this thesis: Rogue. Rogue is a 1980 roguelike (genre to which it gave name) PC game in which the player controls an adventurer (the rogue) and guides his moves into a randomly generated dungeon. The objective is to reach the bottom of the dungeon, steal the Amulet of Yendor and get back to the surface safely. Rogue is a very interesting challenge for Reinforcement learning mainly for the randomness of each play-through (but also for many more reasons as explained in Section 2.2) and so it was worth building an environment for it from scratch. Rogueinabox (the environment) was built as a general tool for AI research in Rogue, with special focus on modularity and extensibility. Snippets of code taken from Rogueinabox implementations are available in Appendix A.

The work presented is divided in three main sections:

Why Rogue? This section explains the projectural choices the choice of Rogue (a command line interface [cli] game from the '80) as the study case for this thesis.

Building an AI learning environment for Rogue This section explains the reasons behind and the structure of Rogueinabox, the Reinforcement learning environment we built to encapsulate Rogue with a Python interface.

Building and improving an agent for Rogueinabox This section shows the evolution of an agent for Rogueinabox, the steps taken, failures, successes and future objectives.

The aim of this work was to build a new environment for interacting with Rogue and use it for experimenting and improving QLearning techniques. Developing an AI Learning environment for a roguelike game is useful not only for this thesis but also for the machine learning community as a whole; the code for Rogueinabox will be open sourced and released to the public. The research done trying to create a good agent can also be reused on other projects. Even if the final performance is not entirely satisfactory an acceptable result has been reached and lots of experience has been gained.

Contents

List of Figures

Chapter 1

Theory

1.1 Machine Learning

Machine learning is a sub-field of computer science which, as the name suggests, studies algorithms that enable computers to learn from data. Once the algorithm has learned and modelized the patterns of the problem it can then predict the output of a new input using his acquired knowledge. With machine learning, especially in recent years due to the abundance publicly available data, we can solve problems that would otherwise be to hard to solve by hand. This is because with machine learning you don't have to come up with complicated strategies to approach the problem; you only need to decide the right techniques and network model for the program "brain" and the algorithm will try to generalize a strategy by itself. The ability to generalize is key in this kind of techniques and overfitting the given data is a common mistake; the problem chosen for this work embraces this generalization principle at his root, being a procedurally generated game.

1.1.1 Neural Networks

The core element of Machine Learning are Neural Networks [10]; a multi-layer web of many weighted neurons. Each neuron computes a simple function but their outputs are combined, weighted and concatenated to obtain an highly complex artificial brain that can solve difficult problems. Different kind of neu-

rons and layers exists, each with its function. In this work we are particularly interested in vision related tasks; Convolutional layers [11, 12, 13] have proven to be an effective choice. A Neural Network might be able to solve difficult problems but the information learned and encoded inside it are not easily accessible; it is often referred to as a ”black box”. For a closer look regarding Convolutional Networks visualization [14] is an interesting read.

1.1.2 Machine Learning Categories

Machine Learning is usually divided into three different categories based on the way the data is presented to the algorithm:

Supervised learning the program is given labeled data in the form of multiple tuples (x, y) where x is the input and y is the desired output. After the training the program should be able to infer with the highest accuracy possible what Y matches a given X (never seen before).

Unsupervised learning The program is given unlabeled data and has to determine some kind of structure or pattern in it.

Reinforcement learning The program has to perform a certain task and acts in an environment receiving a reward (positive or negative) for his actions, learning and collecting its data little by little from experience.

This thesis will be focused on Reinforcement learning and in particular Deep Reinforcement learning [15] and QLearning.

1.1.3 Reinforcement Learning

In a Reinforcement learning scenario many different entities are in play at the same time:

Agent

The Agent is the part of the program that contains the decision making algorithm. It can decide which action perform on the environment given the data

available to him and his ”brain” (the network model, which is updated regularly)

Environment

The environment is responsible for the simulation of the task to be solved and the interaction with the agent. To every action received from the agent it responds with (at least) the reward gained and the resulting state.

State

The way the state of the environment is returned to the agent determines what the agent sees and how it sees it. Naturally the choice for the state representation deeply impacts the results of the training.

Reward

The reward is the only tool to tell the agent what is its objective. To compute the reward information from the game are used, so inherently giving rewards to an agent gives him more information. The programmer must find a good balance in the amount and complexity of information conveyed into the rewards and avoid cheating. A good reward should lead the agent in the right direction highlighting good transitions and not too sparse or otherwise the agent will scramble in the dark for a long time (if not forever).

1.1.4 QLearning

QLearning is a Reinforcement learning technique in which a QFunction Q(a, s) is learned performing actions on the environment. The QFunction is a function that takes an action and a state as input and outputs the expected utility of performing that action. The QFunction is arbitrarily initialized and then, while the agent acts on the environment, it's updated (after each action) until convergence (within a certain ε of tolerance). The update is done with QValue Iteration, a method similar to Value iteration [16, 17] using a version of the Bellman equation, the update

formula is shown below. More information on this method, QLearning and Reinforcement Learning in general can be found in the awesome AI lessons taught at Berkeley University and available online here [17].

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right)$$

Learning rate A parameter that determines how important is the new information compared to what is already known. Usually decreases during the course of the updates.

Discount factor A parameter that determines how important are future rewards, defining if the agent will be more short sighted or far sighted.

The following is the standard QLearning algorithm:

```
1  initialize action-value function Q(s, a) outputs randomly
2  #the QFunction can also be a linear combination of feature functions
3  observe initial state s
4  repeat
5          select an action a
6                  with probability ε select a random action
7                  otherwise select a = argmin_a', Q(s,a')
8          carry out action a
9          observe reward r and new state s'
10         if s' is terminal state the t = r
11         otherwise t = r + γ · max_a' Q(s',a')
12         set $Q(s, a) = t$
13         s = s'
14 until terminated
```

Exploration vs. Exploitation

How should the action to perform each iteration be chosen? If we always chose the action that has the maximum expected value (i.e. the maximum predicted QValue) for the given state we follow a Greedy strategy. Greedy strategy are often not the best strategies because they blindly follow the first (seemingly) good path they are presented. A Greedy strategy prefers Exploitation to Exploration. A common solution to this problem is using an ε-greedy strategy, in which the

algorithm chooses a random action with probability ε, otherwise takes the greedy option. The ε is then slowly decreased during the updates, this way Exploration is prioritized at the beginning (when the QFunction is less accurate) and Exploitation is prioritized in later stages.

1.1.5 DeepQLearning

The QLearning algorithm seen in the last section is great, but is only viable if the state space is small. In a 2015 paper [18] DeepMind demonstrated the effectiveness of using a Deep Neural Network as a substitution for the QFunction, solving the state space size problem. In DeepMind work the network takes game screens as inputs and outputs an array of QValues (expected utility), one for each possible action.

The algorithm proposed in the paper is the standard QLearning one adapted to use a QNetwork and with improvements made to the parameters and settings used.

```
initialize replay memory D
initialize action-value network Q with random weights
initialize target action-value network Q' with the weights of Q
observe initial state s
repeat
        select an action a
                with probability ε select a random action
                otherwise select a = argmin_a', Q(s,a')
        carry out action a
        observe reward r and new state s'
        store experience <s, a, r, s'> in replay memory D

        sample random mini-batch of transitions <ss, aa, rr, ss'> from replay memory D
        calculate target for each mini-batch transition
                if ss' is terminal state the tt = rr
                otherwise tt = rr + γ · max_aa' Q'(ss',aa')
        train the Q network using (tt - Q(ss,aa))^2 as loss
        Every C steps reset Q' = Q

        s = s'
until terminated
```

DeepMind opted to use a simple quadratic loss function.

$$[r + \gamma \cdot max_{a'} Q(s', a') - Q(s, a)]^2$$

Other than replacing the QFunction with a Neural Network two main improvements were introduced to the QLearning algorithm: Experience Replay and Target QNetwork.

Experience Replay With experience replay the agent is trained on batches of episodes, that are picked randomly from a big pool of the agent past experiences. Each time the agent interact with the environment the transition is saved in a tuple such as (old_state, action, reward, new_state, is_terminal). This method of training improves performance by reducing the correlation between states in the same batches because those states come from very different and far moments from the life of the agent. It is also a more general way of training, decoupling the agent interaction with the environment and its training. Training on a pool of past experience has similarities with supervised learning, also the pool can be easily switched from one created by the agent to one created by humans expert in the task.

Target QNetwork To reduce fluctuation in the QValues approximation and improve convergence time DeepMind algorithm uses two QNetworks during the training. One is the working Network that is updated each step and one is the target Network, against which the working network is updated. The target Network is then adjusted to the working network values after a set number of steps.

This algorithm is the starting point of this thesis and even if many things were changed in the learning process its core remains untouched.

1.2 Python and Keras

The implementation language of choice for this project is Python. There are two main reasons for this choice:

- Simplicity and clarity of writing and reading Python code

- Large library support for machine learning and data science in general

Large library support also means many possible choices; we decided to use Keras as our machine learning library. Keras [1] is a high-level neural networks API that provides an easier and cleaner interface to both Theano [2] and Tensorflow [3], two famous machine learning libraries also written in Python. Keras main feature is simple and fast implementation, a key feature to research project that must be able to try many different approaches.

We chose to use Theano as a backend for Keras, this was mainly done because our network structures and hardware setup (training with a GPU) happened to work faster with Theano. Switching to Tensorflow, if needed, is easy with Keras and can be faster if the user would like to train the model using the CPU.

1.2.1 Keras Examples

In this section we show some example of neural networks implementation in Keras, for further information refer to the official Keras documentation [19]

Sequential models

Building a model in Keras is extremely easy, here is an example with the simplest kind of model, the Sequential model. In this example we will build a model like the one used in the DeepMind Atari paper [18]

First initialize a model:

```
from keras.models import Sequential
model = Sequential()
```

then add consecutive layers with .add():

```
from keras.layers import Conv2D, Dense

model.add(Conv2D(32, (8, 8), activation="relu",
        \data_format="channels_first", strides=(4, 4),
        \kernel_initializer="random_normal", padding='same',
        \input_shape=(stacked_frames, rows, columns)))
model.add(Conv2D(64, (4, 4), activation="relu",
        \data_format="channels_first", strides=(2, 2),
         \kernel_initializer="random_normal", padding='same'))
```

```
model.add(Conv2D(64, (3, 3), activation="relu",
        \data_format="channels_first", strides=(1, 1),
        \kernel_initializer="random_normal", padding='same'))
model.add(Flatten())
model.add(Dense(512, activation="relu",
        \kernel_initializer="random_normal"))
model.add(Dense(actions_num, kernel_initializer="random_normal"))
```

Then choose an optimizer and compile the model with .compile():

```
        from keras.optimizers import RMSprop
        rmsprop = RMSprop(lr=0.00025)
        model.compile(loss='mse', optimizer=rmsprop)
```

Functional models

If there is need for a model with a more complex structure Keras offers a functional API that allows to build models with an arbitrary graph structure. The following code is the implementation for a triple tower model that was used in later phases of the agent design. The idea is the same as the Sequential model, but this time we add layers by calling the relative function with the previous layer as the input.

Define the two input layers, one is derived from the other by removing a channel.

```
from keras.layers import Conv2D, MaxPooling2D, Dense, Input, Lambda
#initialize the starting weights at random
initializer = initializers.random_normal(stddev=0.02)

input_img = Input(shape=(5, 22, 80))
input_2 = Lambda(lambda x: x[:, 1:, :, :],
        \output_shape=lambda x: (None, 4, 22, 80))(input_img) # no map channel
```

Define the three towers, each tower is sequential and each layer is passed as an argument to the next one.

```
#Local Vision Tower
tower_1 = Conv2D(64, (3, 3), data_format="channels_first", strides=(1, 1),
        \kernel_initializer=initializer, padding="same")(input_img)
tower_1 = Conv2D(32, (3, 3), data_format="channels_first", strides=(1, 1),
        \kernel_initializer=initializer, padding="same")(tower_1)
tower_1 = MaxPooling2D(pool_size=(22, 80),
        \data_format="channels_first")(tower_1)

```

```
9
10 #Global vision tower 1
11 tower_2 = MaxPooling2D(pool_size=(2, 2),
12         \data_format="channels_first")(input_2)
13 tower_2 = Conv2D(32, (3, 3), data_format="channels_first", strides=(1, 1),
14         \kernel_initializer=initializer,
15     \padding="same", activation='relu')(tower_2)
16 tower_2 = Conv2D(32, (3, 3), data_format="channels_first", strides=(1, 1),
17         \kernel_initializer=initializer,
18     \padding="same", activation='relu')(tower_2)
19 tower_2 = MaxPooling2D(pool_size=(11, 40),
20         \data_format="channels_first")(tower_2)
21
22 #Global vision tower 2
23 tower_3 = MaxPooling2D(pool_size=(3, 6),
24         \data_format="channels_first", padding='same')(input_2)
25 tower_3 = Conv2D(32, (3, 3), data_format="channels_first", strides=(1, 1),
26         \kernel_initializer=initializer,
27     \padding="same", activation='relu')(tower_3)
28 tower_3 = Conv2D(32, (3, 3), data_format="channels_first", strides=(1, 1),
29         \kernel_initializer=initializer,
30     \padding="same", activation='relu')(tower_3)
31 tower_3 = MaxPooling2D(pool_size=(8, 14),
32         \data_format="channels_first", padding='same')(tower_3)
```

Now merge the output of the towers into a single layer and finalize the model

```
1 merged_layers = concatenate([tower_1, tower_2, tower_3], axis=1)
2
3 flat_layer = Flatten()(merged_layers)
4
5 predictions = Dense(5, kernel_initializer=initializer)(flat_layer)
6 model = Model(inputs=input_img, outputs=predictions)
7
8 rmsprop = RMSprop(lr=0.00025)
9 model.compile(loss='mse', optimizer=rmsprop)
```

Chapter 2

Why Rogue?

2.1 What is Rogue?

Rogue is a 1980 dungeon exploration game and the father of the roguelike genre (from which the name). In Rogue the player controls a rogue with the objective to reach the bottom of the dungeon, steal the Amulet of Yendor and get back to the surface safely. The game uses simple ASCII graphics to represent the dungeon and features a vast array of monster and items (scrolls, ring, weapons, armors etc..) but its main feature it the random level generation. Every time a new game is started the dungeon is different and the names of the items are switched around; Rogue is the first game to introduce this as a core mechanic and so is considered a milestone in game history. Many other games have been inspired from it, creating a specific genre; notable example are the classic "Net hack" and "Angbad" or the more recent "Spelunky" and "The Binding of Isaac".

2.2 Rogue Features for Machine Learning

In this section we highlight some of the main features of Rogue that makes it an interesting test bench for machine learning and, especially, deep learning.

- **POMPD nature** Rogue is a Partially Observable Markov Decision Process (POMPD), since each level of the dungeon is initially unknown, and

Figure 2.1: A screenshot taken from Rogue

is progressively discovered as the rogue advance in the dungeon. Solving partially observable mazes is a notoriously difficult and challenging task (see [20] for an introduction). They are often solved with the help of a suitable (built-in) searching strategy, as in [21], that is particularly satisfying from a machine learning perspective. A Neural Network based Reinforcement learning technique to learn memory-based policies for deep memory POMDPs (Recurrent Policy Gradients) have been investigated in [22]. The prospected scenarios are similar to those of Rogue: partial knowledge of the model and deep memory requirements, but they considered much simpler test cases.

- **No level-replay** In many video games, when the player get killed, the game restarts at same level, with the same obstacles. Learning in these situations is not particularly hard, but the acquired knowledge is of no use in subsequent levels, and learning must be started anew. As observed in [23], standard CNN-based networks - comprising Deep QNetworks (DQN) - can

be easily trained to solve a given level, but they do not generalize to new tasks.

Rogue has been one of the earlier examples of procedural generated levels, which was one of the main novelty when the game was introduced: every time a game starts or the player dies, a new level gets generated, every time different from the previous ones. Procedural generated content is partially random, maintaining some constrictions (each level will almost always have nine rooms, for example, but the form, the exact position and the connections between them will vary). This means that extensive, level-specific learning techniques could not be deployed, because the player would eventually die, and the dungeon would change. As a consequence, learning must be done at a much higher level of abstraction, requiring the ability to react to a *generic* dungeon, taking sensible actions. Even with a lot of training data, covering all possible configurations, and a rich enough policy representation, learning to map each task to its optimal policy in a reactive way looks extremely difficult. Likely, we need a mechanism that *learns to plan*, similarly to the value-iteration network (VIN) in [23].

- **ASCII graphics** Rogue is meant to be played in a terminal, therefore renders all its graphics with ASCII characters using the ncurses library. This means two things; the simulation is very fast (in comparison to a more modern and complex graphic game) and the information presented on the screen is already coded and differentiated, which makes it easier to parse and reinterpret it.

- **Memory** In many situations, the rogue need a persistent memory of previous game states and of previous choices in order to perform the correct move. A very simple example is when searching for secret passages in a section of the wall or at the end of a corridor. In this cases, the hidden passage my appear after an arbitrary number (usually between 1 and 10) of pressing of the search button (s) and you need to recall the number of attempts already done. You also need memory in mazes, since you need

(at least) to remember the direction you came from to avoid looping (but a more general recollection of past rogue positions would likely improve the behavior and robustness of the agent). Since the discovery of Long-Short Term Memory models (LSTM) [24, 25], the use of memory in neural networks is increasingly popular, providing one of the most active and fascinating frontiers of the current research (see e.g. the recent introduction of Gated Recurrent Units - GRU [26]). LSTM have been already used for in [27] for Atari games, to replace the sequence of states of [18], and are also exploited in [28]. Rogue could provide another interesting test bench for these techniques.

- **Attention** Another hot topic in Machine Learning is attention, that is the ability, so typical of human cognition, to focus on a specific part of a scene of particular interest, ignoring others of lesser relevance, to build a sequential interpretation and understanding of the *whole* scene we are looking at. Clearly, in a game like Rogue, the environment immediately surrounding the rogue is the main focus of attention, and the agent moving the rogue must have a precise knowledge of it, without however loosing the whole picture of the map. Many techniques have been recently introduced for addressing attention, comprising e.g. the recent technique of spatial transformers [29], that looks promising due to the highly geometrical structure of rooms and corridors. We are also currently investigating a different technique, inspired by *convolutionalization* [30], and essentially based on aggressive use of max-pooling mediated by an image-pyramid vision of the map.

- **Complex and diversified behaviors** Dungeon-like games offer an interesting combination of diversified behaviors: moving around, fighting monsters, descending/escaping the dungeon, acquiring loot, exploit the equipment in the inventory. Merging together these activities and their learning is a complex problem. At present, the agent behavior is traditionally divided into two phases, one involving exploring the map, collecting items, finding

enemies, and another one for fighting [31, 32, 28]. Each phase is covered by a specialized network, trained in a specific way. Combining together neural models optimized on different tasks is still an open issue in neural systems.

Chapter 3

Building an AI learning environment for Rogue

Rogueinabox is an environment for Rogue, a layer between the game and an agent which can interact with Rogue using a Python interface. This is accomplished by running Rogue in a virtual terminal, redirecting its output and input streams. Other environments or frameworks that enable interaction with roguelike games already exists, such as [8] for Desktop Dungeons and BotHack [9] for NetHack, but no project was available for Rogue. Rogueinabox was implemented from scratch; this chapter will describe its design principles, implementation and interface.

3.1 Rogueinabox Modules

In this section we explain in detail the different modules we implemented for Rogueinabox. Since we wanted to create an environment for studying and testing deep learning and Reinforcement learning when designing Rogueinabox we aimed for high modularity and configurability. Those are very important features because in research, and especially in these fields there are many unknown variables and being able to tune them individually, precisely and with ease is a priority. For this reason each module is easily configurable to suit the user needs, who can

add his own methods to the already existing library.

- **State representation** This module manages the state representation that will be fed to the agent. The shape and amount of information the user might want to give the agent might vary wildly depending on the objective that has to be achieved. For example we might want to hide some information (such as the inventory or the status bar) and focus on solving a simpler problem like moving and fighting. We might also want to vary the shape and the channels of the states, using multiple channels or cropped views.

- **Reward functions** This module manages the reward function that will give a score to every agent transition. The choice of the reward function defines which objective we are pursuing and in which way, so obviously the ability to change it accordingly with our aims is crucial. The reward module has access to all the raw information that are presented on the screen (before the state conversion) so its easy to manipulate it and extract whatever data or variation in data we find useful. Obviously the programmer has to keep in mind that the reward given to the agent encodes part of the information used to generate it. He must try to avoid giving the agent too much or unwanted information, therefore cheating.

- **Network models** This module manages the structure of the neural network that will form the mind of the agent. The model defines how the agent "thinks" and what he sees and focuses on given a particular state. We used Keras [1] as our deep learning framework of choice because of his simple and researcher friendly structure. Furthermore model construction is abstracted by a model manager, so the user can also use whatever framework he likes to build the model and just encapsulate it in an object with a Keras like model interface.

- **Experience memory** This module manages the agent memory of his past state transitions, which includes actions taken and rewards received. Experience replay has proven to be an extremely valuable tool in Reinforcement

learning [18]; using this technique is possible to reduce correlation between state transitions. Collecting past experiences also allows to train a different model on an already saved history (provided that the state representation is the same) in a time efficient manner. We also provide tools to filter which transition ends up stored into memory, allowing the creation of a more balanced history that better fits the target needs.

- **Agents** This module manages the different implementations of the agent. We provide 3 base agents; an user controlled one, a random agent, and a Qlearner agent that is capable of training and running the model using a deep QLearning strategy as shown in [18]. As with any other module the user can write his own agent that uses the tools provided by Rogueinabox and implements a learning algorithm of choice.

- **Logging** This module manages the logging of the agent actions. Logs can be printed to various streams (std-out, file...) and filtered by verbosity levels. This module also provides a way to trace the execution time of section of code; its most notably use is monitoring the speed and performance of the model updates during training.

- **UI** This module manages the user interface for Rogueinabox. Since the screen updates require time it is recommended to train with UI turned off and just parse the logfile to retrieve information about the current state of a training. Nevertheless sometimes it might be useful to watch what the agent is doing to hunt down bugs or just to see the result of a training in action. We provide two different implementation of the UI, one is a TKInter GUI (for desktop uses) and one is a Curses UI (for remote headless server uses).

3.2 Rogueinabox Interface

Rogueinabox offers a Python interface to interact with Rogue, which can be used to write different kind of agents. Here is a brief description of it.

This methods allow the programmer to fetch information about Rogue

- **get_actions()** Return a list of the actions currently enabled
- **get_legal_actions()** Return a sub-list of the enabled actions including only legal ones (i.e. actions that will make game time pass)
- **print_screen()** Prints the current screen
- **get_screen()** Return the current screen as a list of strings.
- **get_screen_string()** Return the current screen as a single string with n at EOL
- **game_over()** Check if we Rogue is at the game-over screen (tombstone)
- **is_map_view(screen)** Return True if the current screen is the dungeon map, False otherwise
- **is_running()** Check if the rogue process exited
- **compute_state()** Return a numpy array representation of the current state
- **compute_reward(old_screen, new_screen)** Return the reward for a state transition

This methods allow the programmer to interact with Rogue

- **send_command(command)** Send a command to Rogue
- **quit_the_game()** Send the keystroke needed to quit the game
- **reset()** Kill and restart the rogue process

Snippets of code taken from Rogueinabox implementations are available in Appendix A.

Chapter 4

Building and improving an agent for Rogueinabox

Hard-coded agents for roguelike games similar to Rogue are already available; examples are Borg [33] for Angband and BotHack [9] for NetHack. This chapter will explain the steps taken for building and training a QLearning agent for Rogue in the Rogueinabox environment.

4.1 Setting an objective

Rogue is a very challenging game, even for a human; we did not expect to train an agent that is fully capable of beating the game, at least not in the early stages. We set for the agent different milestones with ever increasing difficulty and tried to tackle them one step at a time. As said before Rogue is a kind of game that supports this approach, offering different and gradually harder tasks to accomplish.

4.1.1 Milestones

The milestone for this project were:

- Reaching the door in the first room

- Reaching a second room walking down a corridor
- Exploring most of the first floor
- Exploring as many floors as possible

These objectives focus only on moving around and map interpretation skills; enemies, items, inventory and other Rogue features were intentionally left out. An interesting continuation to this thesis might be extending the agent behavior making him able to solve more difficult tasks considering also the features we didn't use.

4.1.2 Grading agents performance

Rogue doesn't have a proper score system; the end game screen says gold gathered is the score but this number doesn't always reflect how much the player has advanced into the dungeon (especially in our case, since we straight up ignore gold existence for our objectives). Another way of grading the different agents could be by number of tiles explored, rooms explored or floors descended; still all these approaches don't take in consideration the behavior of the agent. Does he get stuck often or is it killed by monsters? Does he loop around and dies from hunger? Is it good or bad if he stays on the same floor raking up experience instead of diving deep in the dungeon? All these problems make rating the agent behavior (mostly to evaluate changes in code) really difficult, especially when two different version of the agent are failing, which one is worst? In conclusion while comparing two agents that work well one could arbitrary decide a scoring method and stick to it, when comparing failing agents (which often was the case during our research) we often had to resort to a subjective evaluation, based on observations of the agent behavior.

4.2 A common ground for training

Since a lot of tries had to be performed we tried our best to keep the training settings and parameters as constant as possible, exception made for the ones we

were experimenting with.

For the tries described below we:

- Used the same QLearning strategy (same algorithm and same agent)
- Had ε anneal from 1 to 0.1 over the course of the training (usually 2M iterations)
- Used the following parameters: learning rate = 0.00025, $\gamma = 0.99$, batch-size = 32
- Started training with a minimum of 50K transitions already in history, with a maximum history size of 200K
- Checked the progress at constant intervals, usually 1M e 2M for the normal training sessions and 5 30 and 80 loops for the static history training sessions

4.3 The evolution of the agent

4.3.1 The first steps

The most known bibliography example that studied QLearning applied to games is the paper ”Human level control through deep Reinforcement learning” [18] published in 2015 by DeepMind. The application of DeepMind research on Atari games to roguelikes, is mostly an unexplored field; the only example currently available is relative to the game Spelunky [34]. Applications to other genres are also rare but exists, most notably [28] applies DQL tecniques to the first person shooter (FPS) game Doom. As a starting point we decided to try to apply DeepMind work to Rogue, also considering as inspiration the other research cited, mostly for state representation and reward function design. This was easily done implementing in Rogueinabox not only the same QLearning algorithm used for the Atari agent but also the network model and learning parameters.

DeepMind model takes as input an image of the screen and outputs the expected utility of every available action. This model is a sequential model, there are several convolutional layers followed by fully connected ones and the output of each level is the input of the following one. The convolutional layers encode the feature of the screen images while the dense layers combine the features to obtain a (hopefully) good prediction of the QValues. A Keras implementation of this model can be seen in Section 1.2.1

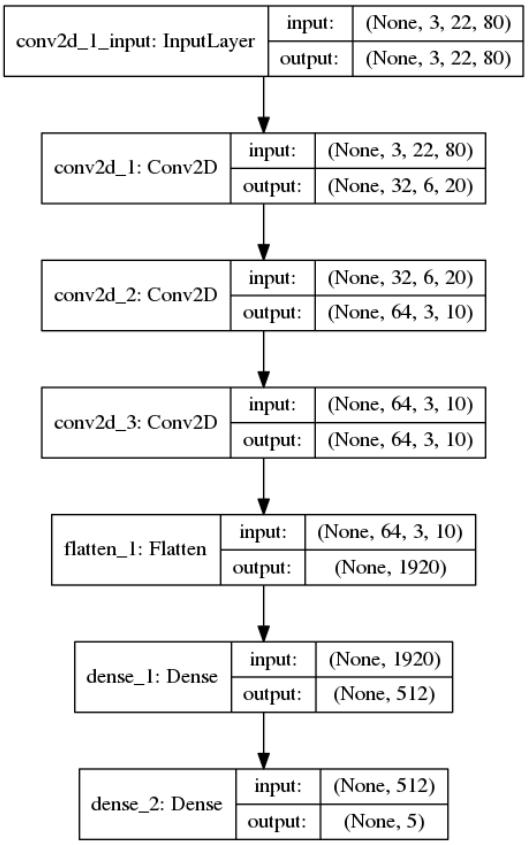

Figure 4.1: The Atari network model

Even if the model was taken mostly ”as is” (exception made for shape differences) other elements had to be adapted.

Adapting the method to Rogue

In DeepMind works [18] the game score is taken as a reward; this is possible because scoring points in Atari games is fairly common so its easier for the agent to experience good transitions. In Rogue, as discussed before, the objective is not really quantifiable by a number and even considering the score (gold count) as a reward, score increases are very sparse and not at all easy to encounter for a training agent. To solve this problem we had to design a custom reward function, described more in detail below.

Regarding the state representation we had to face a different problem, converting the ASCII screen to a simplified image. We decided to divide different points of interest into the different channels of the image. This approach resulted in an increased dimensionality of the states, so we choose to avoid stacking more frame into each state. This is a reasonable modification since in Rogue there is no velocity, each frame is static and can be considered on his own.

Unfortunately this approach didn't succeed and the agent wasn't even capable of moving around and usually would get stuck against the first wall it found. Trying the DeepMind approach was necessary but the problem, despite having similarities (deep QLearning applied to games), is different at his core because in Atari games the level elements were always in the same position on the screen while in Rogue each level has a different room configuration. While keeping game frames as the Network input, the Network structure and other settings had to be adapted.

The adaptation happened slowly, with each step increasingly enhancing the agent performance. Here we will see in more detail the settings that made accomplishing each milestone possible.

4.3.2 Reaching the door in the first room

Reward function

The reward function determines the objective we want to achieve. To reward exploration a positive reward was given when one or more new map pixels became

visible. This can be easily calculated looking at the differences between the old and the new state.

To be more precise the rewards for this step were:

- +1 for discovering a new tile
- -1 for standing still
- -0.1 as a living reward
- 0 for a game over

Since for this step we only wanted to reach a door Rogueinabox would reset (faking a game over) after the first positive reward, that corresponds to the first time the agent discovers a tile (i.e. walks on a door)

The negative rewards are a way to teach the agent to keep moving around and avoid getting stuck on walls, since we notices the agent was really prone to do that.

State representation

Rogue represents its dungeons using only ASCII characters, this view is compact but we decided to simplify it and divide it in channels. Each channel is an 80x22 array of integers and represent a different feature of the Rogue screen. The channel used in this step are as follows:

- **Map channel** Represents the currently visible map, 255 if the position is passable, 0 if impassable
- **Player position channel** Represents the player position, 255 only on the player position, 0 everywhere else
- **Doors positions channel** Represents the doors positions, 255 only on the doors positions, 0 everywhere else

Network model

As said before, the sequential model used for Atari games wasn't good enough for Rogue so we opted for different model structure that uses 3 Towers, merging their output afterwards. Each tower has a different role;

- **Local vision Tower** Takes all the channels as input and after a convolutional step outputs the maximum value for each feature. The maximum value should be relative to a location close to the player position, and we confirmed empirically that this holds true.
- **Global vision Towers** The second and third tower takes as input only the player and doors channels and after the convolutional step an array of maximum values is returned for each feature. This towers should represent a more global vision since the MaxPool step is more loose and return multiple values relative to different parts of the screen. Also the absence of the map layer focuses the attention on the correlation between the player position and the points of interest (in this case the doors).

Experience memory

For this milestone the history for experience replay was collected using a simple FIFO queue, so each step done by the agent ended up inside the history.

4.3.3 Reaching a second room walking down a corridor

Reward function, State representation, Network model

For this second milestone the reward function, the state representation and the Network model remained unchanged, the improvements were due to a different handling of the history used during Experience replay.

Experience memory

The history queue of past transitions used in the first step for experience replay was full of many useless or replicated states. Moreover the ratio of negative reward

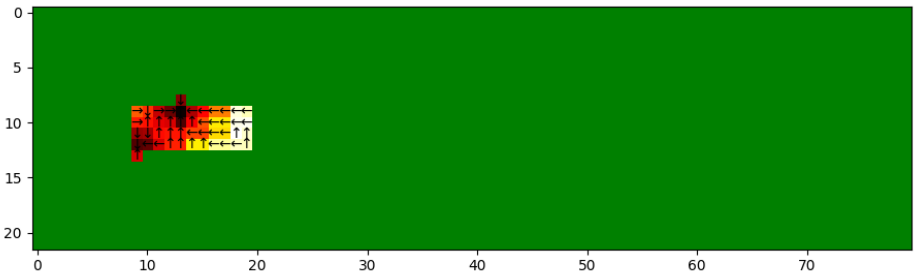

Figure 4.2: A visualization of the agent choices

transition to positive reward ones was very high, this is because especially in the early stages of training when the exploration is totally random, discovering a new part of the map is hard. Usually the useless and replicated states are negative ones, often the ones in which the agent is stuck on a wall, so this two problem can be solved with a single solution. Instead of a FIFO queue we filtered the value that were being inserted into the history taking all positive values but only a percentage of the negative ones.

This new approach to history building greatly increased the agent performance, allowing it to achieve the second milestone.

4.3.4 Exploring most of the first floor

The main problem we needed to solve to achieve this milestone was the agent getting stuck on doors and walls. This problem was solved with a combination of improvement which consist mainly of giving the agent a short term memory allowing him to change behavior when getting stuck. Another important achievement of this milestone is the ability of the agent to come back to a previously visited room if it encounters a dead end.

State representation

A new channel was added to the state ”image”; a channel that encodes the past movements of the agent. At first this layer was an heat-map of past positions, a spot was marked with a higher number (hotter) the more it was walked over by the agent. This approach showed some improvement, but memory representation that provided even better results was found. The added channel is a Snake-like representation of the past positions of the agent; those positions are recorded by Rogueinabox and the corresponding spots are colored with higher numbers the closer the spot is to the rogue.

Reward function

With the Snake-like memory channel the agent had a vision of where it came from, but moving around wasn't incentivised enough by the reward function and the agent kept getting stuck. A new reward, which varies between 1 and 1/n (where n is the length of the agent memory), was added. This reward is computed by first taking the Manhattan distance between the current agent position and its position n moves in the past and then multiplying this value for 1/n.

Network model

For this milestone in each of the model towers was added one more convolutional layer. This was done with the intention of improving the complexity the agent was able to manage, to counterbalance the increase of the state complexity.

4.3.5 Exploring as many floors as possible

We were able to improve the performance of the agent aiming at the completion of this milestone, but the best runs were only able to reach level 4 of the dungeon and with sub-par consistency. Even so, now the agent almost never get stuck against a wall and instead loops around in different rooms. It is also important to consider that the agent is ignoring the presence of monster in the dungeon, monsters which become more dangerous with every level of the dungeon.

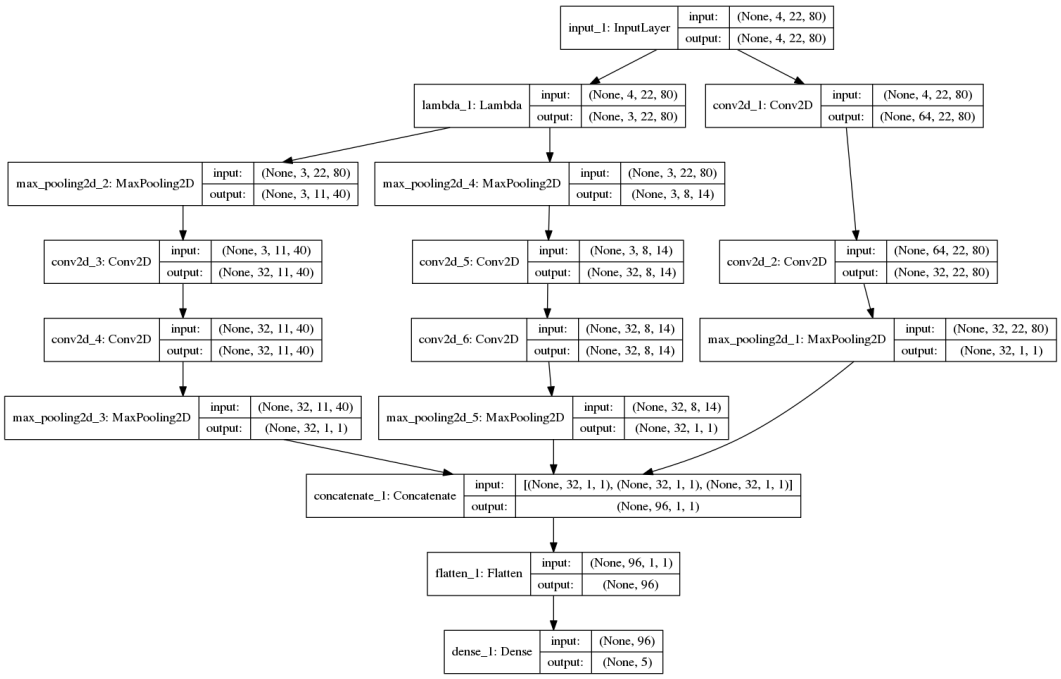

Figure 4.3: The latest iteration of the network model

Reward function

A simple change in the reward function that ended up impacting greatly the performance of the agent is increasing the reward for descending the stairs. As the agent learns to explore more and more, if the difference between exploring few more tiles and descending is small the agent might decide to ignore the stairs (and then never find them again).

State representation

The stair position for each floor was added to a separate channel of the state. In some early tries the stair position was represented in the same channels with the stairs and while it was functional, the number of false positive (doors being treated as stairs) was too high to justify the saving in space.

4.4 Training on static memories

Training a DQL agent can take some time, even a day or two. During research having to wait that much time for a result only to find out that the changes were not effective can be frustrating and slow down the research by a lot. Some of this time is take by the actual training and there is no shortcoming for it, but a huge chunk of it is taken by the environment simulation.

If saved the transition history created by the agent can be reused over and over for testing different models without simulating the actions again, saving a lot of time. Obviously the same history can be reused for a new training only if the reward function and state representation remain the same, so when those change a new history must be created.

This method of training was used multiple times during the research presented in this thesis achieving faster and better result in the short run, while degenerating after a while due to overfitting (as explained in the next section).

The following is the code that was used.

Build the model we want to use and load the history from disk:

```
model = build_model()

print("loading history...")
h = pickle.load(open('history.pkl', 'rb'))
print("history loaded!")
```

Define how each iteration must be set up. The method is the same used for creating a QLearning Experience Replay mini-batch but this time the size of the batch is the size of the entire history:

```
def setup_epoch():
    inputs = np.zeros((len(h), 5, 22, 80))
    targets = np.zeros((len(h), 5))

    for i in range(len(h)):
        old_state = h[i][0]
        action_index = h[i][1]
        reward = h[i][2]
        new_state = h[i][3]
        terminal = h[i][4]

        inputs[i] = old_state
```

```
13          targets[i] = model.predict(old_state)
14
15          if terminal:
16              targets[i, action_index] = reward
17          else:
18              Q_new_state = model.predict(new_state)
19              targets[i, action_index] = reward + 0.99 * np.max(Q_new_state)
20      return (inputs, targets)
```

Setup a batch and train on it for a certain number of epochs, in each loop the batch is updated with the better predict function that we just trained:

```
1  iteration = 0
2  while True:
3      iteration += 1
4      print(iteration)
5      inputs, targets = setup_epoch()
6      model.fit(inputs, targets, epochs = 10, batch_size=32)
7      model.save_weights("weights.h5", overwrite=True)
```

4.4.1 Rog-o-matic supervised learning

As seen in the last section an agent can be trained using a pre-built history. This pre-built history doesn't have to be built by the Reinforcement learning agent, it can also come from other sources such as human expert players or hard-coded agents.

For this thesis we also build an agent, called StalkerAgent, that uses a modified version of Rogueinabox Instead of Rogue inside the virtual terminal is run Rog-o-matic [35] an hard-coded agent that has been probed able to win at Rogue.

Even if this tool wasn't used for the purposes of this thesis the agent is provided with Rogueinabox and could easily be used in later works to improve the behavior of a Reinforcement learning agent.

4.5 Training time vs. Improvement

It could seem obvious that increased training time would in turn bring increased agent performance but it is not always the case.

Two different learning techniques were used, one that learned while building an history and one that learned on a pre-built history.

In both approaches the agent behavior keeps improving for a while and after that, if training continues, the agent starts unlearning what it has learned. This inversion happens after a different amount of iteration (and time) which depends on both the training settings and the method used (since the static history methods is faster and iterates on much larger batch sizes).

The reason for this strange behavior is probably due to two factors:

- **The constant increase in predicted QValue**
- **Overfitting**

As we can see in the figures below, relative to two different training that dynamically build the history, the predicted QValues have the tendency to increase over time. This could seem normal, since a performance increase should be reflected in better prediction, but this increase seem to continue indefinitely. This could be due to the fact that the Qleaning algorithm always picks the max of the prediction for choosing his actions and overtime if the choice is not balanced it might lead to worst predictions. In Rogue some moves might be statistically better than others due to map generation biases and the rectangular screen size. This phenomenon has been studied before and is explained in [36]; in the paper the proposed solution is to use another QNetwork to predict the value (and the corresponding action) to chose after each prediction, instead of simply picking the maximum.

The agent trains for a long time on an history of state transition; the map in Rogue is randomized for each life and floor but is essentially generated by the same hidden game rules and this might cause overfitting. The phenomenon is probably negligible when the history is build dynamically, since old transitions are rotated out and new are added each step, but it might be really impactful when training on a pre-built history.

In conclusion, these observations prove that pouring more training time into a model will not result in a linear increase in performance and could instead worsen

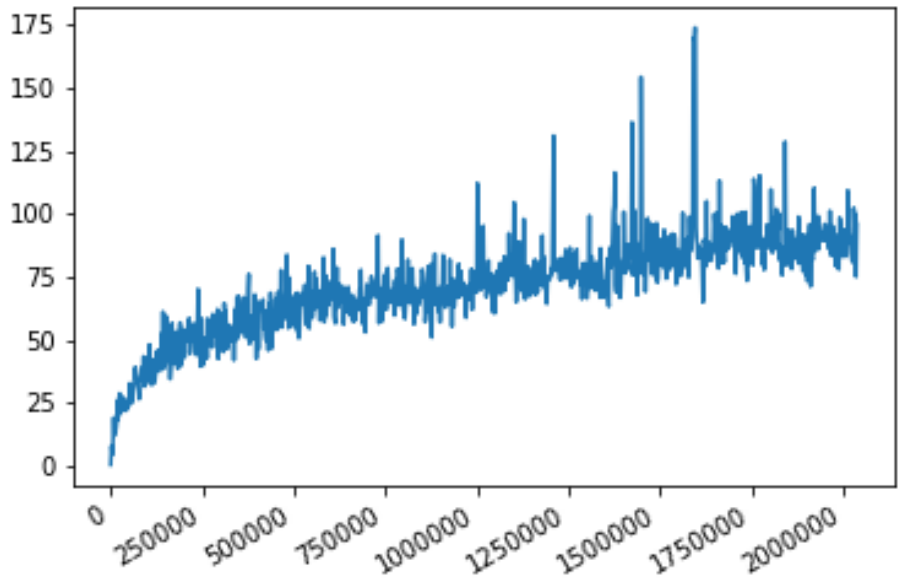

Figure 4.4: The variation of the average predicted QValue over time during training - Tower model - 2M iterations

it. After a while changes in the model or training settings are needed to further improve behavior.

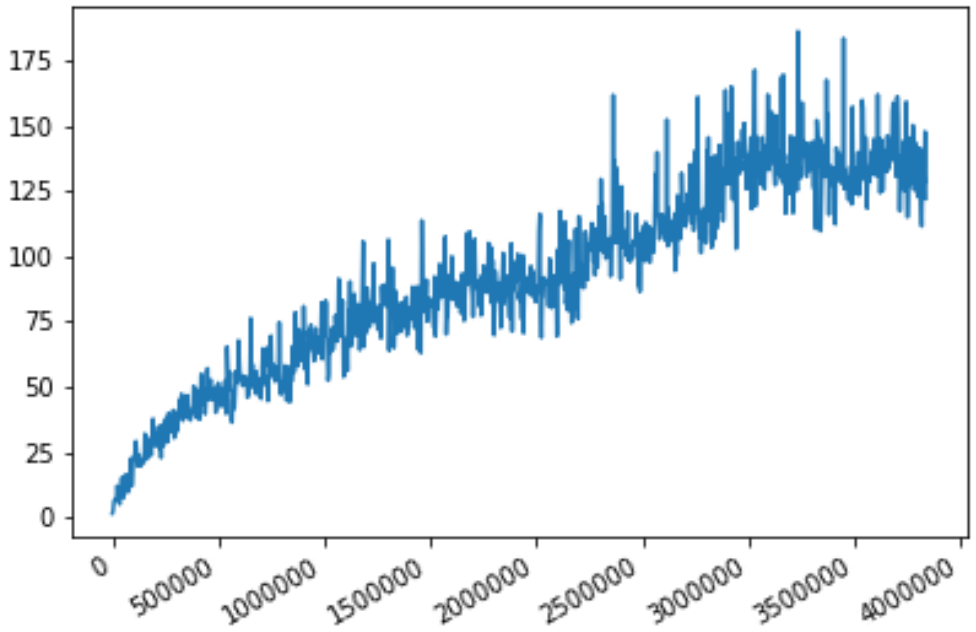

Figure 4.5: The variation of the average predicted QValue over time during training - Tower model - 4M iterations

Chapter 5

Conclusions

5.1 Current results

This thesis had two main objectives:

- Build a Reinforcement learning environment for Rogue
- Build and improve a QLearning agent for Rogue

Both this tasks were completed successfully. Rogueinabox was created and improved alongside with the agent construction; it features not only the main interface for interacting with rogue but also a library of modules which can be either reused or taken as inspiration for newer ones.

As for the agent, a set of milestone was set and achieved, even if the performance is not always consistent. Working on improving the agent shed light on the importance of certain training decisions, such as state and recent memory representation and experience history build-up and reuse. The success in building an agent also serve as a proof of the working state of Rogueinabox and his fitness for Reinforcement learning research.

5.2 Future work

Even if the results were good there is still much work to do in the future.

Rogueinabox could be improved with more feature and modules. Also since the source code for Rogue and Rogomatic is available, modifications to it could help Rogueinabox. For example certain feature of Rogue could be temporary disabled or modified to help the agent focus more on particular aspects (e.g. disabling monster spawns or changing items/monster spawn rate)

The agent exploration could be improved by increasing consistency; right now the agent is able to explore new area of the dungeon and descend deeper into it but often gets stuck in loops between two rooms, unable to find the stairs, and dies of starvation. The agent might also be extended to perform new tasks, such as fighting monsters or retrieving equipment, both as isolated objectives or in conjunction with what it already knows.

The training speed could also be improved; faster training speed means faster research and faster advances. We showed how training on a pre-built history can speed up learning (but with increased risk of overfitting), in the future the learning algorithm could also be improved using asynchronous methods and CPU training (instead of GPU), as shown here [37].

Appendix A

Code examples

In this section will be presented some snippets taken from the current Rogueinabox implementation.

A.1 Rogue Interface

Here is the main function of Rogueinabox, the one responsible of sending commands to the game and updating the internal state afterwards. The keypresses and the corresponding game responses are read/written through a pipe that communicates with the game process, running in a virtual terminal. A delay is added between each action to give Rogue the time to process each keypress. Once the game update is received the internal variables are updated and the reward and state are calculated based on the the chosen functions. In the meantime other janitorial tasks need be made, such as refreshing Rogue screen to avoid glitches and dismissing game messages to avoid duplicate states.

```
1 send_command(command):
2     """send a command to Rogue"""
3     old_screen = screen[:]
4     #send the command through the pipe
5     pipe.write(command.encode())
6     #send the refresh screen command to avoid graphic glitches
7     if command in get_actions():
8         pipe.write('\x12'.encode())
9     #wait for the screen to update
```

```
10      time.sleep(0.01)
11      #read the updated screen
12      _update_screen()
13      #ignore messages
14      if _need_to_dismiss():
15          # will dismiss all upcoming messages,
16          # because dismiss_message() calls send_command() again
17          _dismiss_message()
18      new_screen = screen[:]
19      #keep internal variables updated
20      _update_stair_pos(old_screen, new_screen)
21      _update_player_pos()
22      _update_past_positions(old_screen, new_screen)
23      reward = compute_reward(old_screen, new_screen)
24      new_state = compute_state()
25      terminal = game_over()
26      #if a different condition other than game over is reached
27      #set terminal accordingly
28      if reward_generator.objective_achieved or state_generator.need_reset:
29          terminal = True
30      #return transition info
31      return reward, new_state, terminal
```

A.2 State Representations

This is an example of a state function, a function that determines what and how the agent sees using internal Rogueinabox information. This particular state function returns a state made of 5 layers, each one representing different information about the current state. These layers are stored in a numpy array for better manipulation; the size of the array is 5x22x80 and each layer is represented as a 22x80 (Rogue map size) image. Each image (or channel of a 5 channel image) has white pixel on the coordinates where a particular is present and black pixels elsewhere.

```
1  compute_state():
2      """return a 3x22x80 numpy array filled with a numeric state"""
3      #the most common case, the screen is a view of the dungeon
4      if roguebox.is_map_view(roguebox.screen):
5          player_pos = roguebox.player_pos
6          stair_pos = roguebox.stair_pos
7          passable_pos = []
8          doors = []
```

```
9           #declare the state
10          state = np.zeros([5, 22, 80])
11          #find the coordinates that are passable and the position
12          #of doors
13          for i, j in itertools.product(range(1, 23), range(80)):
14              pixel = roguebox.screen[i][j]
15              if pixel not in '|- ':
16                  passable_pos.append((i, j))
17              if pixel == '+':
18                  doors.append((i, j))
19
20          #start filling the state array
21
22          # layer 0: the map
23          # 0 if passable, 1 if impassable
24          for i, j in passable_pos:
25              state[0][i - 1][j] = 255
26
27          # layer 1: the player position
28          # 1 only on player position
29          if player_pos:
30              #the player pos has to be adjusted because unprocessed screen
31              #has two more lines, one top and one bottom
32              state[1][player_pos[0] - 1][player_pos[1]] = 255
33
34          # layer 2: the doors positions
35          # 1 only on doors positions
36          for i, j in doors:
37              state[2][i - 1][j] = 255
38
39          # layer 3: the position
40          # 1 only on stair position
41          if stair_pos:
42              state[3][stair_pos[0] - 1][stair_pos[1]] = 255
43
44          # layer 4: snake-like of past positions, with fading
45          unit = 255/10
46          past_positions = roguebox.past_positions
47          for i, pos in enumerate(past_positions):
48              state[4][pos[0]-1][pos[1]] = (i+1)*unit
49
50      elif roguebox.game_over():
51          # the screen is the tombstone game over screen
52          # return an array of 0 to differentiate
53          state = np.zeros([5, 22, 80])
54      else:
55          # the screen is inventory, option or a transition screen
```

```
56          # return an array of 1 to differentiate
57          # the agents should not get to this case
58          state = np.ones([5, 22, 80])
59      return state
```

A.3 Reward Functions

This is an example of a reward function, a function that determines what should be the behavior of the agent. This particular reward function tries to make the agent explore and move around, descending to the next floor as soon as possible. To achieve this much greater rewards are given to descending as opposed to exploring, also continuous small rewards are given if the agent has been moving away from where is was before (hence probably exploring) The following are the rewards implemented in this function:

- +100 for discovering a new tile
- +1 for discovering a new tile
- +0.1 x d, where d is the Manhattan distance of the current position and the position 10 moves before (up to +1)
- -1 for standing still
- -0.1 as a living reward
- -0.1 for a game over
- -1 for error states

```
1       compute_reward(old_screen, new_screen):
2           """return the reward the last action yield"""
3           def get_player_pos(screen):
4             for i, j in itertools.product(range(1, 23), range(80)):
5               pixel = screen[i][j]
6               if pixel == "@":
7                 return (i, j)
8           def manhattan_distance(a, b):
9               return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

```
10          if not roguebox.game_over() and roguebox.is_map_view(old_screen)
11                  \and roguebox.is_map_view(new_screen):
12              # parse the screen for infos
13              infos = get_infos(old_screen, new_screen)
14
15              # compute reward
16              reward = 0
17              #reward for descending
18              if infos["dungeon_level"]["old"] < infos["dungeon_level"]["new"]:
19                  reward = 100
20              else:
21                  #punishment for standing still
22                  if get_player_pos(old_screen) == get_player_pos(new_screen):
23                    reward = -1
24                  #reward for exploring and discovering new parts of the map
25                  elif infos["explored_tiles"]["new"] >
26                          \infos["explored_tiles"]["old"]:
27                      reward = 1
28                  else:
29                      #living reward
30                      reward = -0.1
31                  #add movement bonus, to incentivise wandering
32                  a = roguebox.past_positions[0]
33                  b = roguebox.past_positions[-1]
34                  reward += manhattan_distance(a, b) * 0.1
35
36          elif roguebox.game_over():
37              # game over
38              reward = -0.1
39          else:
40              # we are in some other view, probably a sub-menu like
41              # inventory or options
42              # return a dummy reward of -1 to avoid crashes
43              reward = -1
44
45          return reward
```

A.4 Agents

This an example of the implementations of the main methods of an agent. This particular agent is a Qlearner agent, it interacts with Rogueinabox and tries to learn how to reach its objectives using DeepQLearning (DQL). There are three main functions: predict, act and observe which implement the principles of Rein-

forcement learning. The agent predicts which is the best action to make, performs it acting in the environment and then learns from its experiences. These three steps combine into the function train_step, which is called every iteration.

```
    predict():
        """return a numpy array of length actions_num all set to 0
        except for the index of the action to take which is set to 1"""
        # chose an action $\varepsilon$ greedy
        actions_array = np.zeros(parameters["actions_num"])
        #chooses an action in an $\varepsilon$-greedy way
        if random.random() <= parameters["epsilon"]:
            action_index = random.randrange(parameters["actions_num"])
        else:
            q = model.predict(state)
            actions = parameters["actions"]
            if parameters["only_legal_actions"]:
                legal_actions = roguebox.get_legal_actions()
                for action in actions:
                    if action not in legal_actions:
                        q[(0, actions.index(action))] = -np.inf

    act(action_index):
        action = parameters["actions"][action_index]
        reward, new_state, terminal = roguebox.send_command(action)
        old_state = state
        state = model_manager.reshape_new_state(old_state, new_state)
        return reward, terminal

    observe():
        #sample a mini-batch from past experiences
        minibatch = history_manager.pick_batch(parameters["batchsize"])
        inputs = np.zeros((parameters["batchsize"],) + state.shape[1:])
        targets = np.zeros((parameters["batchsize"],
                \parameters["actions_num"]))

        # Now we do the experience replay
        for i in range(parameters["batchsize"]):
            old_state = minibatch[i][0]
            action_index = minibatch[i][1]
            reward = minibatch[i][2]
            new_state = minibatch[i][3]
            terminal = minibatch[i][4]

            inputs[i] = old_state
            targets[i] = model.predict(old_state)

            if terminal:
```

```
44                 targets[i, action_index] = reward
45             else:
46                 Q_new_state = target_model.predict(new_state)
47                 targets[i, action_index] = reward +
48                         \parameters["gamma"] * np.max(Q_new_state)
49
50         loss = model.train_on_batch(inputs, targets)
51         return loss
52
53     train_step( iteration):
54         action_index = predict()
55         reward, terminal = act(action_index)
56         history_manager.update_history(action_index, reward, terminal)
57         # Begin training only when we have enough history
58         if history_manager.hist_len() >= parameters["minhist"]:
59             observe()
60             # anneal $\varepsilon$
61             if parameters["epsilon"] > parameters["final_epsilon"]:
62                 parameters["epsilon"] -= (parameters["initial_epsilon"]
63                         \- parameters["final_epsilon"]) / \
64             #save progress
65             if iteration \% 100000 == 0:
66                 _save_progress()
67                 #plot(state[0])
68             #update target QNetwork
69             if iteration \% 10000 == 0:
70                 target_model.set_weights(model.get_weights())
71         if terminal or parameters["iteration"]:
72             roguebox.reset()
73             _reinit()
74
75     run_step():
76         action_index = predict()
77         reward, terminal = act(action_index)
```

Bibliography

[1] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[2] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[4] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *J. Artif. Intell. Res. (JAIR)*, vol. 47, pp. 253–279, 2013. [Online]. Available: http://dx.doi.org/10.1613/jair.3912

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai universe," https://github.com/openai/universe, 2016.

[6] ——, "Openai gym," 2016.

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

[8] V. Cerny and F. Dechterenko, "Rogue-like games as a playground for artificial intelligence–evolutionary approach," in *International Conference on Entertainment Computing*. Springer, 2015, pp. 261–271.

[9] krajj7, "Bothack," https://github.com/krajj7/BotHack, 2015.

[10] U. of Wisconsin. A basic introduction to neural networks. [Online]. Available: http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

[11] A. Deshpande. A beginner's guide to understanding convolutional neural networks. [Online]. Available: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

[12] ——. A beginner's guide to understanding convolutional neural networks part 2. [Online]. Available: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

[13] karpathy@cs.stanford.edu. Convolutional neural networks for visual recognition. [Online]. Available: https://cs231n.github.io/convolutional-networks/

[14] F. Chollet. How convolutional neural networks see the world. [Online]. Available: https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html

[15] Y. Li, "Deep reinforcement learning: An overview," *CoRR*, vol. abs/1701.07274, 2017. [Online]. Available: http://arxiv.org/abs/1701.07274

[16] L. P. Kaelbling. Value iteration. [Online]. Available: https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node19.html

[17] N. H. Dan Klein, Pieter Abbeel. Berkeley ai course. [Online]. Available: http://ai.berkeley.edu/lecture_videos.html

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[19] F. Chollet. Keras documentation. [Online]. Available: https://keras.io/

[20] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[21] M. Wiering and J. Schmidhuber, "Solving pomdps with levin search and EIRA," in *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, L. Saitta, Ed. Morgan Kaufmann, 1996, pp. 534–542.

[22] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Solving deep memory pomdps with recurrent policy gradients," in *Artificial Neural Networks - ICANN 2007, 17th International Conference, Porto, Portugal, September 9-13, 2007, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. M. de Sá, L. A. Alexandre, W. Duch, and D. P. Mandic, Eds., vol. 4668. Springer, 2007, pp. 697–706.

[23] A. Tamar, S. Levine, and P. Abbeel, "Value iteration networks," *CoRR*, vol. abs/1602.02867, 2016. [Online]. Available: http://arxiv.org/abs/1602.02867

[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[25] F. A. Gers, J. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, 2000. [Online]. Available: https://doi.org/10.1162/089976600300015015

[26] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Gated feedback recurrent neural networks," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 2067–2075. [Online]. Available: http://jmlr.org/proceedings/papers/v37/chung15.html

[27] M. J. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *CoRR*, vol. abs/1507.06527, 2015. [Online]. Available: http://arxiv.org/abs/1507.06527

[28] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, S. P. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 2140–2146. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14456

[29] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 2017–2025. [Online]. Available: http://papers.nips.cc/paper/5854-spatial-transformer-networks

[30] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 640–651, 2017. [Online]. Available: https://doi.org/10.1109/TPAMI.2016.2572683

[31] M. McPartland and M. Gallagher, "Creating a multi-purpose first person shooter bot with reinforcement learning," in *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Perth, Australia, 15-18 December, 2008*, P. Hingston and L. Barone, Eds. IEEE, 2008, pp. 143–150. [Online]. Available: https://doi.org/10.1109/CIG.2008.5035633

[32] B. Tastan, Y. Chang, and G. Sukthankar, "Learning to intercept opponents in first person shooter games," in *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012, Granada, Spain, September 11-14, 2012*. IEEE, 2012, pp. 100–107. [Online]. Available: https://doi.org/10.1109/CIG.2012.6374144

[33] B. Harrison. Angband borg. [Online]. Available: http://www.thangorodrim.net/borg.html

[34] A. Coggeshall. Playing spelunky with deep q learning – adam coggeshall. Youtube. [Online]. Available: https://www.youtube.com/watch?v=atyscEQRaSc

[35] M. L. Mauldin, G. Jacobson, A. Appel, and L. Hamey, "Rog-o-matic: A belligerent expert system," in *Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence, London Ontario, May 16, 1984.*, 1984.

[36] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[37] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

Special thanks

Andrea Asperti and Carlo De Pieri, for the incredible support and collaboration which made this internship and thesis project awesome.

All my friends, without whom my days would be dull and this thesis would've been completed a month earlier.