Papathanasiou Theodoros 2011030058

Semester Assignment Report

COMP513: Autonomous Agents

Abstract: We present our implementation of a reinforcement learning agent who plays Super Mario World by using Q-Learning and Boltzmann exploration. We apply our algorithm to the first two levels of the game and report on our results.

1 Introduction

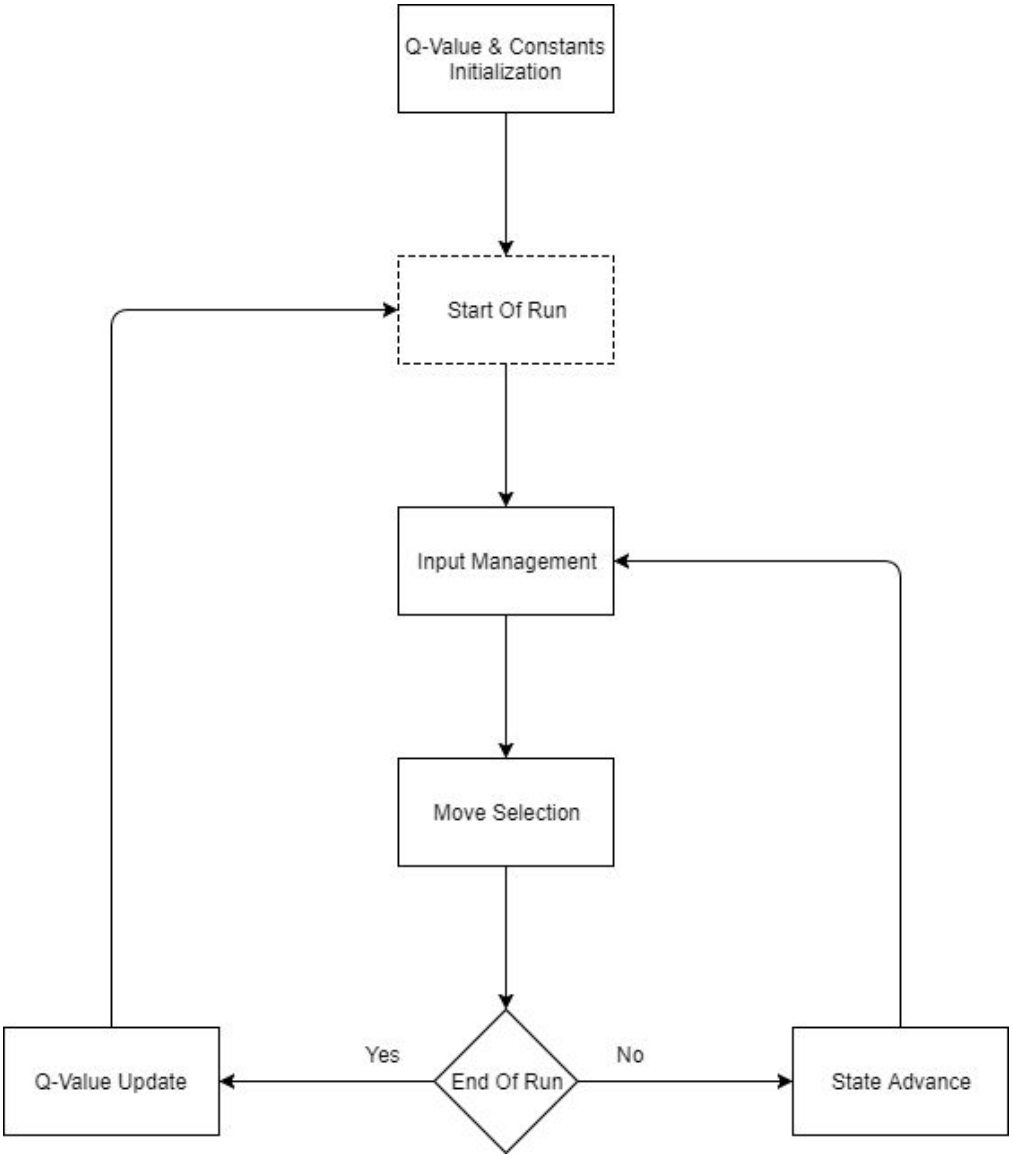
The problem of solving games through reinforcement learning has been extensively covered in the field of machine learning with works such as "TD-Gammon" and "Playing Atari with Deep Reinforcement Learning". In this project we try to implement a simple Q-Learning algorithm based on the works of "MarI/O" and "Playing Mario with Deep Reinforcement Learning"to solve a Super Mario World (SMW) level. To take advantage of pre-existing infrastructure we choose to use the programming language LUA as it is used in most of the SNES emulator APIs.

Our algorithm is a Q-Learning multistate implementation. Each state represents half of an in-game timer unit. The number of states is limited to 700, 100 more than the time limit in SMW levels. The available actions are reduced from 8 to 4 so we can minimize the state space and reduce learning time.

We apply our algorithm to the first two levels of SMW with a range of different values for the Q-Value update function and Bolzmann exploration and we observe the changes in the agents actions and its convergence to the optimal action policy.

Finally we draw conclusions about the importance of non zero initial Q-Values on the convergence to the optimal move. We also realize and quantify the importance of hardware and stability in the emulation for a proper learning process.

2 Implementation Model



Q-Values and Constants Initialization

In our implementation the starting Q-Values are not set to 0, but are set to include some prior knowledge about the game. This choice is made to reduce the learning time of our agent and accelerate the overall development process.

After extensive experimental runs, the constants of the Q-Value update function were set to:

$$a = 0.5$$
$$\gamma = 0.8$$

With these values we observe that the agent's learning and subsequent actions are streamlined, resulting in the smoothest observed behavior.

The temperature's starting value is set to T = 4, as to promote randomness and exploration which carries through deep into the learning process. The temperature's value is annealed over time to a rate of 0.001 for each run.

Inputs

As input, our agent detects Mario's movement to the right. The more we move to the right the better the reward.

Move selection

After each state advance, our agent considers what it's next move should be. Firstly it computes a probability for each move through the use of boltzmann exploration. Then, it chooses a move based on their probabilities.

Output

The agent's output is the button to be pressed on the next frame of the game. In our implementation the available buttons to be pressed have been reduced from eight to 4, to reduce learning time.

End of run

When our agent detects that it has been stationary for 80ms it terminates the current run, receiving a reward based on the distance traveled. This choice was made based on the duration of the death animation. As a result a run is terminated when Mario dies or when he is stuck to a vertical obstacle e.g. a wall or a pipe.

Q-Values update

When the run comes to an end, the agent updates it's Q-Values. We iterate from the first state to the last and update the values based on the result we received.

3 Q-Learning

On this section we delve deeper into the mechanics of the learning process we have implemented and examine the functions used to achieve proper knowledge.

Q-Value update function

As a means of learning, all R.L. agents need to update their beliefs about their environment. In our agent this happens after the end of each run, through the following function:

$$Q(s,a) \leftarrow Q(s,a) + a(r + \gamma maxQ(s',a') - Q(s,a))$$

If s' is a terminal state then:

$$Q(s,a) \leftarrow Q(s,a) + a(r - Q(s,a))$$

Learning Rate *a*

The learning rate determines at what rate the agent discards old knowledge in favor of new one. Our agent has a constant learning rate of 0.5. In this way the newly acquired knowledge affects our agents view about the environment but not in an extreme way.

Discount Factor *γ*

The discount factor describes the importance of the expected reward in the next state. When set to a lower value the future reward is taken less into consideration. This makes the agent choose actions based on short term results. In our case the discount factor is set to a constant value of 0.8. This choice is made to maximize the long term reward i.e. reaching the end of the level.

Reward Function

Our agent receives a reward at the end of each run. This reward is directly tied to the distance traveled to the right, specifically for all states except the terminal and the one before it:

$$r \leftarrow Rightmost\ Pixel\ on\ X\ Axis\ /\ 8$$

On terminal states(s) and the ones before them(s-1) we differentiate between two cases.

If mario has finished the level:

$$r_s \leftarrow Rightmost\ Pixel\ on\ X\ Axis\ /\ 8\ +\ 1000$$

If mario has not finished the level:

$$r_s \leftarrow -100 \ \& \ r_{s-1} \leftarrow -50$$

4 Boltzmann Exploration

Classic Q-learning chooses its next action to be the one with the highest Q-Value. To accelerate the learning process we can implement some sort of exploration of the action - state space. We distinguish two types of exploration, exploitive and naive. In naive exploration, the agent tries its moves randomly with uniform probability, therefore not taking into consideration the prior knowledge it has acquired. On the other hand, in exploitive exploration the agent chooses the action with highest Q-Value with probability p and all other actions with 1-p.

In our implementation we choose to use Boltzmann exploration, a technique which biases the probability p based on the magnitude of the Q-Values. In it an action a is chosen with probability:

$$P(a) = \frac{e^{Q(a)/T}}{\Sigma_{a'} e^{Q(a')/T}}$$

The temperature T is set to 3 at the start of the learning process and is annealed over time, to a minimum of 1, so that the exploration probability decreases and the exploitation probability increases.

5 Technical Obstacles & Restrictions

In the development of our agent we faced some technical obstacles and restrictions we had to overcome. Some of these are described below.

States and the in-game timer:
In the beginning of the agent's life, we had configured the states of the algorithm to correspond to a real world second. Quickly though we observed that the in game timer did not count in seconds but in an arbitrary unit. This proved very disrupting as some times the actions of the agent did not happen on the same in game time and the obtained knowledge was nulled.

With much trial and error and enough research, we found that the timer was counting based on the frames per second of the original snes machine and the Hz of the monitors it was meant to run on. With this in mind, we synced the algorithm states to the in-game timer by tying them to the frames per second and not to real life seconds.

As a result our agent's actions are always synced to the in game time and the acquired knowledge corresponds to the same game-state and algorithm state. This solution though, comes with some limitations. The fps of the emulation must be kept constant to 60 or else the algorithm states and the in-game timer will be thrown off. One last comment is that even though the two timers are synced, there is still some

small phase between them meaning that there is still a small chance of desyncing to the point of disrupting the algorithm's knowledge.

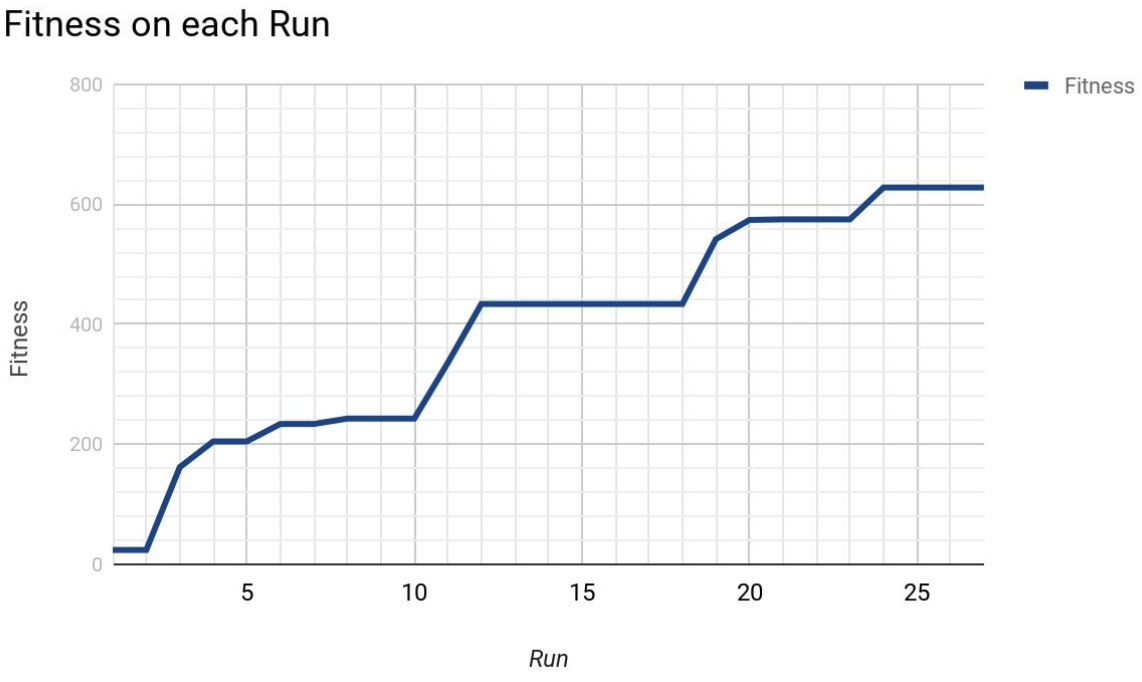
Emulation and system requirements:
Another restriction to our implementation is the emulation process. Due to its high system intensity, our algorithm needs to be run on a machine capable of emulating a SNES with no apparent effect on its performance. This leaves portable devices and microcomputers and their usage to accelerate the learning process out of the question.

Software dependencies:
The whole algorithm is built on the api of a specific third party emulator (BIZHAWK). This leaves it vulnerable and dependent to api and software changes.

6 Results & Future Work

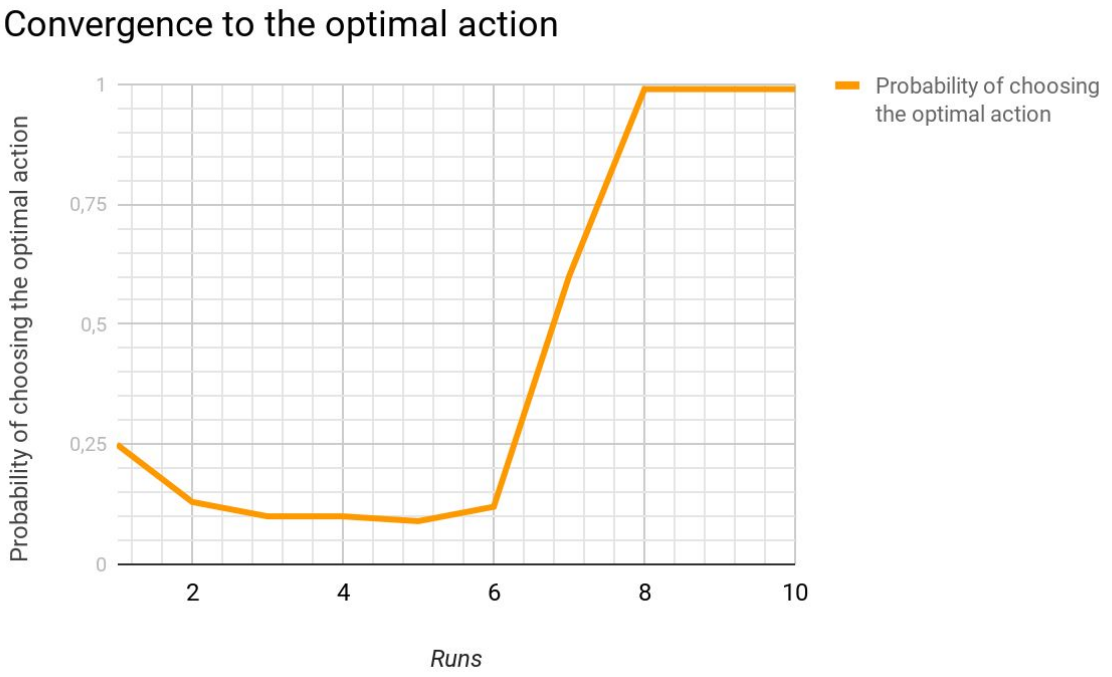
With our current configuration and the prior knowledge added to the agent during initialization we get the following statistics

Avg. runs per obstacle	Avg. runs to finish
1.47	32



Graph. 6.1 The evolution of the agent's fitness from the start to the end of a level.

When we remove the prior knowledge from the initialization of our agent's Q-Values we observe that we still converge to the optimal solution, just with a delay shown in the graph below.



Graph 6.2 The average number of runs required to converge to the optimal action

As of now, our agent is restricted greatly by level design because of our implementation's shortcomings concerning the end and restart of mario's runs. For future work, we suggest the improvement in recognising when a run has ended e.g. through the death animation. Additionally, there is much room for improvement in the design of the state space and its representation either through neural networks or a more efficient lua based architecture. Finally, we believe that the agent's performance could be greatly enhanced through the introduction of machine vision(pattern recognition) to create a more observable game and a knowledge based on causes rather than only the effects of events.

References

1. Seth Bling: MarI/O - Machine Learning for Video Games

2. Alexander Jung: Playing Mario with Deep Reinforcement Learning

3. Caroline Claus, Craig Boutilier: The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems