Report Third Project IAJ

João Vítor ist199246 Sebastião Carvalho ist199326 Tiago Antunes ist199331

2023-11-2

Contents

1 Introduction

The goal of the project was to change the game of the second project, and implement a Reinforcement Learning algorithm to play it.
For this, we first started with making the game restart when the player dies or wins the game, so that we can leave the agent to train without having to restart the simulation each time.
We only implemented one Reinforcement Learning Algorithm: Q-Learning.

2 Game Changes and new States

2.1 Game Changes

As said before, to make the game restart when the player dies or wins, we had to change certain aspects of the game.
First, we started by not destroying the objects like chests and enemies when they were collected or killed. Instead, we just disabled them by setting their active boolean to false. When the game ended, we simply changed all the objects' active flag back to true. We also created a restart function for NPCs, so that their base stats would go back to their initial stats. This is extremely useful for Sir Uthgard, to recover his health, level, etc., and for the enemies, so that they would respawn in their initial positions.
We also created a new Script for collectable items like the potions and the chests, because sometimes the player would collide with them, changing their position, and we needed this position to reset back whenever the game restarted.

2.2 New States

Since we decided to use Q-Learning, we also needed to change how we represented the state, since too many different states would make our Q-Table too big and need a lot of memory. So, we decided to create another state representation, with less states, but still enough to distinguish between different situations.
We decided to use macro states, which are states that represent the current game state in a more simplified way, reducing the total number of possible states. The representation we chose varied across the development, so we decided to describe each one in the next section.

2.3 Serialization

To use a Reinforcement Learning algorithm, we needed to save the Q-Table, so that we could load it in the next training session.
For this, we created 2 serializers, one to serialzie the Q-Table, and another to serialize the stats of the run (wins, deaths, timeouts, etc.).
We also added 2 flags to the AutonomousCharacter script, in the section QLearningOptions, that allows us to choose if we want to save the Q-Table in the end of the session, and if we want to load the Q-Table from previous training sessions in the beginning of the game.
We use 2 JSON files to store the data: "qtable.json" and "run.json". If the flag to save the table is selected, the file will be overwritten, so be careful to not lose any data unintentionally.

3 Q-Learning

3.1 Algorithm

Q-Learning is a Reinforcement Learning algorithm, this means that occasionally the agent has rewards or punishments as a result of performing actions.
For this algorithm we use macro states, these represent the current game state in a more simplified way, reducing the total number of possible states for the game. Anything that is not encoded in these states cannot be learned, so these changed as we improved the algorithm. For each state, the algorithm must also know what actions are available to it. This is because the algorithm will keep a Q Table that stores the quality of an action given a certain state, this is called the Q value.

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma * max(Q(s',a'))),\ \alpha \in [0,1],\ \gamma \in [0,1]$$

Where s is the current state, a is the action performed, r is the direct reward for the performed action, s' is the new state and a' is the best action in the new state.
α and γ are the learning parameters. α is the learning rate and represents how much the new Q value will change because

of the new reward. γ is how much the indirect reward will propagate backwards.
We kept α at 0.5 and γ at 0.1 throught all iterations. After every execution of an action in a state, we update it's respective Q value acording to the formula above. For our exploration strategy, we choose ϵ-greedy, this means that for learning we get the action with the highest Q value for the state, most of the time, but we have a low chance of using a random action, ϵ.
We implemented our Q-Table using dictionaries, where the key is the state and the value is another dictionary, where the key is the action. This way, we only keep in memory states that were already seen, instead of having all possible states, and thus saving memory.

3.2 Training process

For the training process, we let the algorithm first run and calculate the Q values. We would then see the amount of victories and determine if we should let it run for longer. If the version seemed to be learning very little we cut the learning short but in other cases we let it run for about 1000 iterations. We would also note what the agent did in the run, to see if there was any any information in the state that he was missing.
After this we would change what we thought was necessary and train the agent again. All enemy behavior was kept on during training.

4 Q-Learning Versions

4.1 1st Version

For our states, we started by capturing what we felt was most important while still keeping the total amount of states as low as possible. We captured the characters' health, money, level, position and the time of the game.
These stats were simplified in the following way.

4.1.1 States

Table 1: Simplified states of Health

Status	**Health + Shield (interval)**
VeryLow	[0,6]
Low	[7,12]
OK	[13,18]
Good	>18

Table 2: Simplified states of Time

Status	**Time (interval)**
Early	[0,50[
Mid	[50,100[
Late	[100,150]

Table 3: Simplified states of Money

Status	**Money**
Poor	0 and 5
Mid	10 and 15
Rich	20

Table 4: Simplified states of Level

Status	**Level**
Level2	2
NotLevel2	1 and 3

Table 5: Simplified states of Position

Status	**x Coordinates**	**z Coordinates**
TopLeft	<= 66	>= 48.27
TopRight	> 66	>= 48.27
BottomLeft	<= 66	< 48.27
BottomRight	> 66	< 48.27

This gave us a total amount of 288 possible states.
The reason as to why we choose these divisions was due to our previous experiences with the game.
Health is divided into intervals of 6 because that's the attack damage of the orcs, VeryLow will result in death if the character is attacked by an orc. We found these values to be descriptive enough that we didn't need to do intervals for the skeletons' damage.

In terms of Time we considered 3 intervals. We simply considered that having 3 phases of the game was enough for Sir Uthgard to understand when it was more urgent to get the chests.
For money, we gave it 3 states, we represent each 2 values in a state essencially. We did this in the hopes that the agent wouldn't need these intermediate values to learn that he should pick up the chests.
Level only has 2 states, this is because out of observation we can see that it is possible to win only by being in level 2, and we don't want to waste any more time than we need leveling up or killing enemies.
The last statistic is position, in here we divided the level into rectangles, this was done by analyzing the map and seeing what areas where more connected so that the character would prefer actions in one area.

4.1.2 Reward Function

Our initial reward function only rewarded the agent for reaching the win condition, in which case the reward was 100, or any of the loss conditions, where the reward was -100, as well as +0.1 for any state that was Level2.

4.1.3 Learning Parameters

We used α at 0.5 and γ at 0.1, like specified before. The random action chance, ϵ, was 5%.

4.1.4 Results

In 400 iterations Sir Uthgard hadn't learnt much and won 0 times.
With so few rewards, the algorithm didn't have enough guidance, and our whole approach to rewards was wrong. We had to reward executing an action in a certain state, and not reward the states themselves. To do this, we had to consider the state we were in, and the state the action led to. This also meant death and win states were useless, we simply had to reward the action.

4.2 2nd Version

4.2.1 Improvements

In this second version we kept the same states. We focused mainly on improving the reward function. We rewarded +10 for going up a money state, +10 for actions that made a VeryLow hp state go to Good, +20 for leveling up. Actions that made Uthgard change the quadrant of the map he was on were penalized with -1.

4.2.2 Results

Sir Uthgard displayed a much more intelligent behavior, and we saw him get his first win. After 252 runs, he had 168 deaths, 83 timeouts, and 1 win. But unfortunately, after 2749 iterations, he had only won 15 times. The algorithm was still lacking direction, got stuck trying bad actions for too long and kept goofing around in the late game.

4.3 3rd Version

4.3.1 Improvements

We increased the number of money states from 3 to 5: VeryPoor, Poor, Mid, Rich, and VeryRich.

Table 6: New simplified states of Money

Status	**Money**
VeryPoor	0
Poor	5
Mid	10
Rich	15
VeryRich	20

This way, Uthgard would always be instantly rewarded for picking up a chest. This increased the number of states to 480. For the reward funtion, we added a -5 reward for actions in a Late game state that didn't lead to an increase in money. And we increased epsilon to 0.1, so he'd try new things more often.

4.3.2 Results

These changes were very effective. After 1013 runs, he had 666 deaths, 319 timeouts, and 28 wins. This is when we started to notice the biggest challenge to using Q learning for this problem. Actions were getting mislabeled as bad. If Uthgard got killed or timed out while walking to a chest, picking up the chest would get a negative reward. Sometimes, this was a good thing, for example, when he decided to pick up the chest on the bottom left as his first move. But would be very bad late game, as Uthgard was too scared to pick up the last chests.

4.4 4th Version

4.4.1 Improvements

To avoid Sir Uthgard getting spooked of chests due to the timeouts, we removed the negative reward for timeouts all together. The action itself wasn't responsible for the loss, it was the combination of all actions that led to that point.

4.4.2 Results

The change didn't have the impact we expected. The behavior was looking better, but the results were slightly worse. In 1005 iterations, he died 770 times, timed out 224, and won 11. We were still seeing the same problem.

4.5 5th Version

4.5.1 Improvements

In the late game, since Sir Uthgard was often killed by the bottom left orc or by the dragon while going for those chests. He started doing everything except picking them up, even if those enemies were dead. So, we added a new variable to the state. DragonAndOrc: EitherAlive if at least one was alive, and BothDead. This would help him understand that the chests weren't scary if those enemies weren't there.

Table 7: States for Dragon and bottom left Orc

Status	**Dragon**	**Botton Left Orc**
EitherAlive	Alive	Alive
EitherAlive	Alive	Dead
EitherAlive	Dead	Alive
BothDead	Dead	Dead

Theoretically, this would increase the number of states to 960, but, since most of the time one of them was alive, there wouldn't be that many more states in practice. We also added a +10 reward for transitioning to a state where they were both dead.

4.5.2 Results

Now, in 1001 runs, he had 691 deaths, 264 timeouts, and 46 wins. At this point, putting epsilon at 0, Uthgard won 15/20 runs. And after arduous over-night training with an epsilon of 0.1, and some extra runs with epsilon at 0 to refine his strategy, he had a 20/20 win rate. 4585 runs, 2279 deaths, 1776 timeouts, and 530 wins later, Uthgard had learnt how to clear the dungeon. But we were still not satisfied.

4.6 6th Version

4.6.1 Improvements

For this version, we wanted him to learn faster, in a more stable way and win in less time.
We tried increasing the reward for picking up chest to +15. We decreased the penalty for actions that didn't lead to an increase in money in the late game to -4 and changed how it worked. Previously, if Uthgard was attacked by an orc while going for a chest, picking it up would get a -5 rewarded because the state would be updated, and the action had not led to more money. Now, we made it so picking up chest actions can't get this penalty. We made the transition from Low hp to Good hp also get a +10 reward, instead of only VeryLow to Good. We changed Late to start at 90 seconds. And finally, the win reward is now +100 + [time remaining], to incentivize faster runs.

4.6.2 Results

These changes lead to him learning faster, as evidenced by the graphs bellow. Not only that but, he also seemed a lot more stable as we started to have him run without random actions. Within less runs than previously, he had 100% win rate in 141 runs. And won, on average, 20 seconds faster than before. At the end, the algorithm had run 4781 times, with 2885 deaths, 1441 timeouts and 455 wins. The Q table saw 403 states.

5 Final Version of Q-Learning

To recap all the changes we did, we'll present the final version of the implementation here.

5.1 States

Table 8: Simplified states of Health

Status	Health + Shield (interval)
VeryLow	[0,6]
Low	[7,12]
OK	[13,18]
Good	>18

Table 9: Simplified states of Time

Status	Time (interval)
Early	[0,50[
Mid	[50,90[
Late	[90,150]

Table 10: Simplified states of Money

Status	Money
VeryPoor	0
Poor	5
Mid	10
Rich	15
VeryRich	20

Table 11: Simplified states of Level

Status	Level
Level2	2
NotLevel2	1 and 3

Table 12: Simplified states of Position

Status	x Coordinates	z Coordinates
TopLeft	<= 66	>= 48.27
TopRight	> 66	>= 48.27
BottomLeft	<= 66	< 48.27
BottomRight	> 66	< 48.27

Table 13: States for Dragon and bottom left Orc

Status	Dragon	Botton Left Orc
EitherAlive	Alive	Alive
EitherAlive	Alive	Dead
EitherAlive	Dead	Alive
BothDead	Dead	Dead

We have a total amount of 960 possible states.

5.2 Reward Function

Our rewards are awarded acording to what the action changed given the current state.
Rewards for money changes:

- If money is 25 (win condition), the reward is +100 + [time remaining].
- If we picked up a chest the reward is +15.
- If we are in the Late time state and the action wasn't picking up a chest we reward -4.

Rewards for Health changes:

- If we reach a value smaller or equal to zero (loss condition), we reward -100 for the action.
- When we pass from health VeryLow or Low to Good, we reward the action with +10.

Rewards for level changes:

- We reward +20 to an action that takes us to Level2.

Rewards for position changes:

- We reward -1 to actions that makes us change the quadrant we are on.

Rewards for the Dragon and Orc state changes:

- If we executed an action that changed this state to BothDead and we are in Late time state we reward +10 for the action.

Time state changes are not rewarded directly.

5.3 Hyperparameters

In the end, our learning parameters were α at 0.5, γ at 0.1 and ϵ at 0.05. After 4200 iterations we changed ϵ to 0.

5.4 Learning (Data collected)

These charts shows a training session of our agent with the 5th and with the final version of Q-Learning, where 4200 iterations were done.

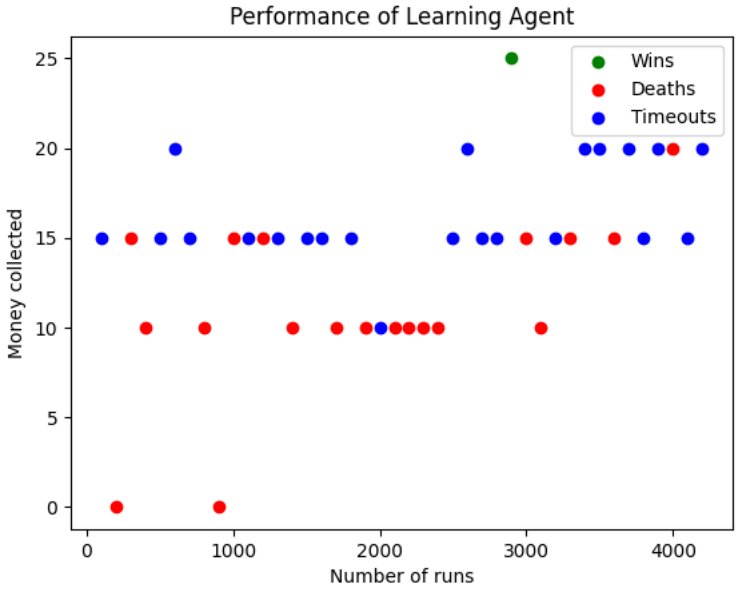

Figure 1: Performance over time for 5th version of Q-Learning

Figure 2: Performance over time for 6th and final version of Q-Learning

6 Conclusions

With our final version of Q-Learning, we can see that the agent learns fast not to die, dying by timeout on the 100th iteration. But still in some iterations, the player gets worse results. This might be due to the ϵ value being too high, and the agent choosing a random action that is not optimal.

One solution we could use to solve this problem, would be to have a variable ϵ, that starts at 1 and decreases over time, preventing random actions from happening too late in the training. We could do the same with the learning rate α, so that the agent would learn faster in the beginning, and slower as it got better.

Overall, we think our reward function is good, since it quickly makes our agent learn to pick up chests, avoid enemies and heal itself. Even though we have an iteration when the player dies with 5 money, we consider it an outlier, since there isn't any repetition of this behavior.

Q-Learning is a very simple algorithm, so other RL algorithms using Neural Networks might show better results, but we think that Q-Learning is a good algorithm since with the right reward function and states, it can learn very fast.