

# Reinforcement Learning for a Platformer Game

Ondřej Duba

ČVUT–FIT

dubaond3@fit.cvut.cz

January 4, 2026

The main goal of the semestral project was to create a very simple platformer game and then build an agent that will try to win that game as effectively as possible

## 1 Introduction

There exist many Machine Learning algorithms for solving games these days. From playing Chess and Go to playing more complex 2D/3D games. Inspired by these complex algorithms, I decided to implement my own algorithm, that would be able to solve a primitive 2D game based on player's movement.

## 2 Game implementation

I implemented a very simple physics engine for a 2D platformer game that is used to control the entire game logic. It keeps track of player's position, it can detect collisions with map edges, objects, coins and finish flag. The physics also includes applying gravity to the player and keeping track of player's velocity and adjusting it accordingly each frame.

## 3 Q-learning

The agent is trained using the Q-learning reinforcement learning algorithm, where the main idea is to pick the best move at the current position from what has the agent previously learned.

The game has a finite number of states and we can therefore represent every state that the agent can reach. In my implementation, the game state consists of agent's surrounding area grid ( $n \times n$ ) (keeping track of the tile type, such as air, barrier, etc.), agent's velocity in  $x$  and  $y$  directions simplified to three states (movement to the left, right or stationary), and whether a player is on the ground or in the air. Each state can be evaluated by a reward function (saying how good the state is). If an agent is closer to finish, collects coins or completes level, reward will be higher. On the other hand, if an agent dies or takes too long to reach the finish, the reward will be negative. On top of that, at each state, agent can choose from four actions, move left,

right, jump or remain idle.

Initially, every move is as good as the others (each valued as 0). As training begins, agent first chooses moves randomly, because there aren't any best moves, but after each move, agent updates the moves value at the current state as follows

$$Q^{new}(S_t, A_t) = (1 - \alpha) \cdot Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a))$$

Where  $Q^{new}(S_t, A_t)$  is the value of move  $A_t$  at position  $S_t$ ,  $\alpha$  is learning rate,  $Q(S_t, A_t)$  is the current value of the move,  $R_{t+1}$  is the reward for taking action  $A_t$  at state  $S_t$ ,  $\gamma$  is discount factor (how much we value the future state) and  $\max_a Q(S_{t+1}, a)$  is the estimate of the optimal future value.

By first trying random moves and then updating their values, we eventually, after many iterations, find the best moves that lead the agent to the end.

It's important to mention, that we actually keep trying random moves even if we already have some estimate of what the best moves are. This move selection is controlled by parameter  $\varepsilon$ , that is initially set to  $\varepsilon = 1$  and is slowly decaying to a lower value. This effectively means, that initially when we don't have too much information we tend to choose more random moves than later, when we already know what the best move might be. This step is really important, because it allows the agent to learn not only initially, but even later on and possibly improve the best moves. The decaying of  $\varepsilon$  is done after each generation (agent wins the game or dies).

## 4 Visualisation

I implemented two visualisation modes, first mode, where the user can play the game themselves, this was mainly for debugging the game's physics and second mode, visualisation of the agent learning the game.

Agent visualisation shows the agent playing the game and displays statistics, such as what is the current generation, how many times has the agent

won, the minimum number of steps it took agent to finish the game across all generations and more. User can slow down or fast forward the visualisation.

In current implementation, the training is tied to the visualisation itself and this was intentional, as the visualisation was the primary focus, but it is possible to train the agent without the visualisation with very little code changes as the API is prepared for this.

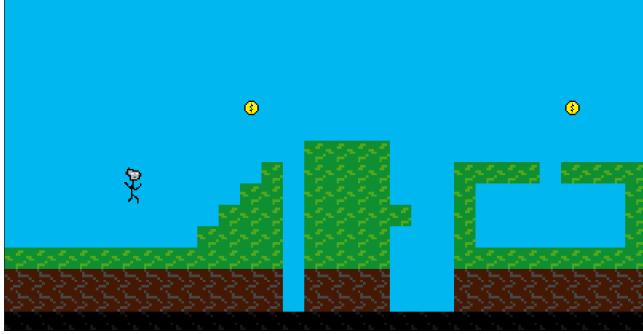


Figure 1: Game visualisation showcase, showing the agent and the map

## 5 Performance

Initially, I was slightly concerned of Python's performance, because the state space that the agent can explore is quite large. For  $3 \times 3 = 9$  surrounding area grid (where there are 5 possible tile types), while also keeping track of  $x$  and  $y$  velocity (3 values each) and if the agent is on ground (two values), there are  $5^9 \times 3 \times 3 \times 2 = 35,156,250$  possible states. However, in reality, only a few percent of those states are ever reached, but still not a small amount.

The main part of the computation is trying so many possible options and computing which one is the best, while also needing to train the agent for a lot of generations in order for the agent to find a path from start to finish.

With that being said, I haven't met any Python related performance issues and the algorithm was fast enough that it could train the agent in 5 to 10 minutes, depending on the map difficulty. Though, it would obviously train much faster in other languages, such as C++.

## 6 Pitfalls of this approach

Due to large amount of possible states, it is not possible to increase the grid dimensions significantly and this might lead to the agent not being able to identify larger obstacles, that are further away. Another thing that I found out were instances, where the agent got stuck behind an obstacle that

was closer to the finish (because the reward function rewards the agent for being closer to the finish), even though the actual way to the finish could only be achieved by going back. This obstacle could technically be avoided by removing the reward for distance from finish, but in my testing, this reward made finding the finish much faster overall. Agent could also sometimes avoid this obstacle completely by randomness.

## 7 Conclusion

I was happy with the results, because the agent was able to solve a large number of levels I had built and some of those levels were quite difficult. After letting the agent train for 20,000 – 30,000 generations, it was able to play the levels almost perfectly.

The next steps for this project could be optimizing the code to make the learning faster, optimizing the parameters of the agent and adding new features, such as new obstacles, enemies, etc.

It would be interesting to compare Q-learning with other Reinforcement Learning or Deep Learning algorithms.

## References

- [1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, second edition, 2014. [Online PDF accessed 2026-01-04] <https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf>.
- [2] The Qt Company. Pyside6 documentation. online, 2026. [cit. 2026-01-04] <https://doc.qt.io/qtforpython/>.