An application of SARSA temporal difference learning to Super Mario

Lucas Jenß
Hamburg University of Applied Sciences
lucasandreas.jenss@haw-hamburg.de

Abstract

This essay covers the application of the SARSA algorithm to the video game Super Mario. This includes the basic formalizations of the problem at hand, as well as an explanation of the algorithm choice. The focus is set on the modelling approach of the state and action space for the game. The evaluation discusses the resulting implementation.

I. Introduction

In this essay, the applicability of reinforcement learning to the platform game Super Mario is evaluated. In reinforcement learning (Sutton and Barto, 1998), an agent seeks an optimal control policy for a sequential decision problem (van Seijen et al., 2009). In contrast to supervised learning techniques, such as those commonly used for neural networks, the agent is never given any specific examples of the behavior it is supposed to learn. Instead it is given positive and negative rewards, and derives a behavioral policy from that information.

Super Mario is a 2-dimensional game with the goal to reach the end of a level located somewhere to the right of the player. In order to get there, the player must avoid enemies and may collect certain power-ups and items. Formally, this problem can be modeled as a Markov Decision Process (MDP), which consists of a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where

- $\mathcal{S}$ is the set of possible states
- $\mathcal{A}$ is the set of possible actions
- $\mathcal{T} : \mathcal{S}x\mathcal{A} \longmapsto \mathcal{S}$ is the transition function, which maps a certain action taken in a certain state to the state that results from that action
- $\mathcal{R} : \mathcal{S}x\mathcal{A} \longmapsto \mathbb{R}$ is the reward function

The goal is then to learn an optimal policy $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$, which maps every state to the best action that can be taken in it. In other words, the strategy selects the actions $(a_1, a_2, \ldots, a_n)$, starting at the initial state s_0, such that the sum of their return values is maximized (Rodrigues et al., 2008). If the return value function, the state and the action space are properly modeled, this will result in a successfully completed level.

II. Modelling the game

When trying to apply a reinforcement learning algorithm to a specific problem, the first step is to model the state space $\mathcal{S}$, including the actions $\mathcal{A}(s)$ that can be taken in any given state $s \in \mathcal{S}$ (as previously defined in context of the MDP). Each state must represent a situation which the agent might encounter, so that the best action for each state can be determined by the learning algorithm. The result of the learning algorithm is a policy $\pi^*(s, a)$.

The policy $\pi^*(s, a)$ dictates the probability for an action a to be taken in any given state s, such that the outcome is most beneficial to the agent.

In the game “Super Mario”, the game environment is divided into segments of 16x16 pixels, and the area of the environment which the player

sees is 20 segments wide and 15 segments high, as can be observed in Figure 2. In general, Mario can move in six different ways: Jump, Left, Right, Crouch, Left-Jump and Right-Jump. Additionally, all actions except Jump and Crouch can also be taken while running, adding another four actions, resulting in a total of ten actions (as depicted in Figure 1).

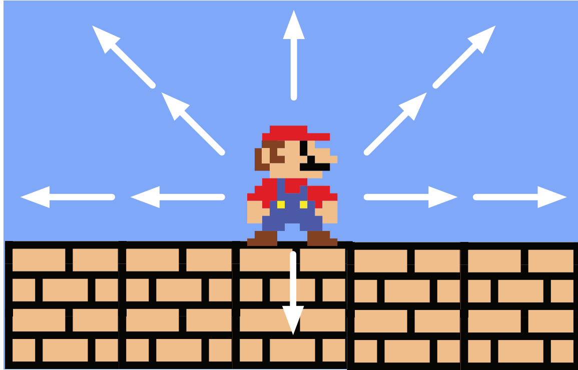


Figure 1: Actions that can be taken by Mario. Two arrows in the same direction indicate an action while running.

The naïve approach to modelling the Super Mario state space is then to define every possible location of Mario inside a level (i.e. the x- and y-coordinate) as a state. Assuming a level width of 211 segments (the first level of the Super Mario Bros. game), this would yield $|S| = 211 * 15 = 3165$. This information, however, will not suffice to beat a level of the game, since enemies are moving targets. Thus standing at a certain coordinate and going one segment to the right might result in Mario's death or not depending on how much time has passed in the game. Another difficulty is the fact that Mario can have different velocities, depending on how long he's been running. He must run eight sequential segments in order to achieve full speed, which is sometimes required, for example, to jump over large gaps. So for each position of the agent inside a level, it has to be in one of eight "speed states". Taking that information into account would result in $|S| = 211 * 15 * 400 * 8 = 10,128,000$ (assuming 400 time units, i.e. seconds, for the first level).

While such a number of states is certainly not computationally intractable, it can not be considered a particularly efficient representation either. There are two major downsides:

1. Computational inefficiency: the time it takes the agent to learn is proportional to the number of states that need to be explored.
2. The resulting policy is only applicable for a single level, since two levels are usually designed very differently. As such, to store the policies for all 32 levels of the first Super Mario Bros. game, at minimum around 320MiB of storage[1] would be required (for a game that measures 41KB in total).

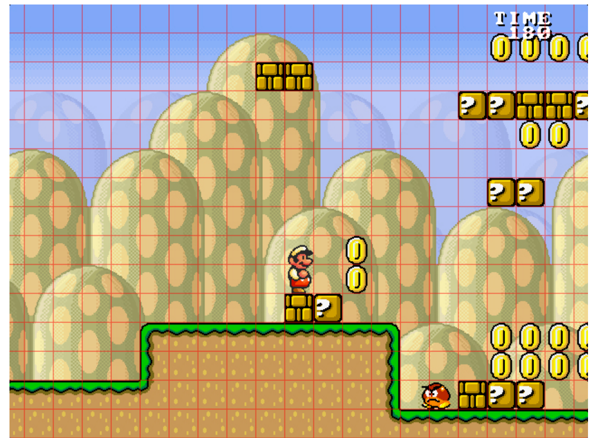


Figure 2: Super Mario, as the player sees it (grid added). 20x15 segments are visible at all time.

As a result, a more abstract notion of state was selected when modelling the state space. Instead of Mario's position inside the level, a number of disjoint situations were extracted that Mario is likely to encounter. As an example, the situation in Figure 3 will be examined. It is composed of three disjoint situations, namely:

1. Mario stands before a ledge
2. Mario stands before a coin
3. An enemy approaches Mario's position

The situations (SI) are disjoint in the sense that

[1]Assuming that we can discard on average 75% of all states as unreachable, 2,532,000 states still need to be stored, as well as the action that should be taken in each. Since there are 10 possible actions, the action can be represented using a 4-bit integer. That results in 10,128,000 bits, or around 10MiB per level, and 320MiB for all levels.

each of them has a set of properties, for example that in (1), which is unique to the situation (2).

$$\text{BeforeCoin} = \{\text{Mario}_x + 1 = Coin\} \quad (1)$$

$$\forall x \in SI : \forall y \in SI : x \neq y : x \cap y = \emptyset \quad (2)$$

$$\mathcal{S} = \mathcal{P}(SI) \quad (3)$$

$$|\mathcal{S}| = 2^{|SI|} \quad (4)$$

Through these properties it is then possible to determine whether or not a situation applies to a given environment. It is also possible for two or more of these situations to occur at the same time (as is the case in Figure 3). Consequently, the state space must contain all possible permutations of the extracted situations, i.e. their power set (3). Since the state space grows expontentially with the number of disjoint situations (4), they have to be selected carefully in order to avoid a large number of unnecessary states.

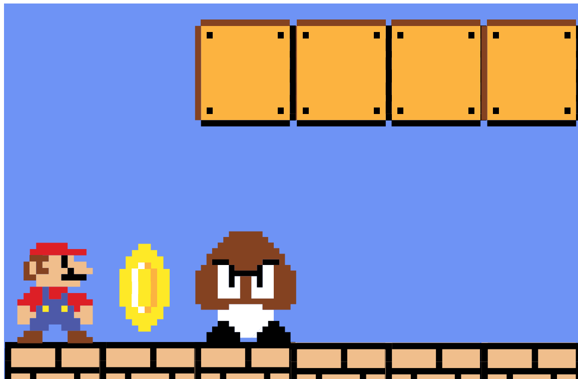


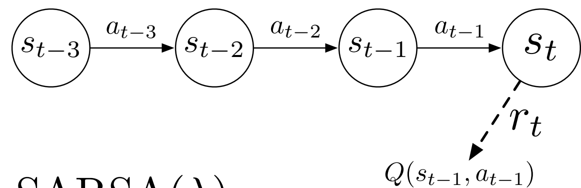
Figure 3: Example of a situation which Mario might encounter. It is composed of three disjoint situations as explained above.

III. Algorithm choice

One-step SARSA was chosen as the reinforcement learning algorithm, since it is an easily implementable on-policy time-difference learning algorithm, making it applicable for non-episodic tasks, which was chosen as the modelling approach in this paper (see section II). This ruled out Monte Carlo based learning algorithms, as these require episodic tasks.

An often employed improvement of SARSA is called SARSA(λ). The difference of SARSA(λ) is that, for every action a_{t-1} taken in a state s_{t-1}, the received reward r_t is not only applied to the state-action-pair $Q(s_{t-1}, a_{t-1})$, but also to all previous steps in a decaying fashion, determined by the λ parameter. This difference is illustrated in Figure 4. In the SARSA(λ) part of the figure, r_t is propagated to the $Q(s_{t-2}, a_{t-2})$ state-action-pair decayed by λ^2, and consequently the state-action-pair $Q(s_{t-n}, a_{t-n})$ would be decayed by λ^n. For many applications, this improves the learning speed of the algorithm (Sutton and Barto, 1998).

SARSA



SARSA(λ)

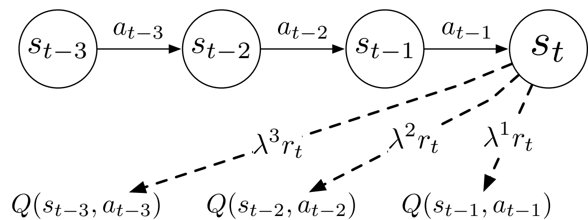


Figure 4: Difference between SARSA and SARSA(λ) learning algorithms. Note that reward propagation is only shown for the state s_t.

In this particular case though, SARSA(λ) is not a good fit. This can be easily shown using a simple example. Let us consider the situation depicted in Figure 5. If SARSA(λ) was used, the negative reward for the death in the last state would propagate through all previous states. This would result in a penalty getting applied to Q("coin in front", "run right") as well as Q("path clear", "run right"), which are both perfectly good choices for the agent to make. As such, SARSA(λ) was not used.

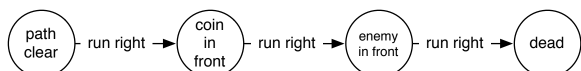


Figure 5: Situation depicting why SARSA(λ) is not a good fit for the chosen state-action model.

IV. Environment

For the evaluation of the implemented algorithm, the "Benchmark" environment from the Mario AI Championship was used (*Mario AI Benchmark*, 2013). The environment, which is implemented in Java, provides a simulation of the Super Mario game world. Instead of using statically designed levels, as is the case in the real Super Mario games, the environment provides randomly generated levels as well as accelerated simulations to reduce learning time.

Even though the environment is not very well documented and contains some quirks, it made the implementation of the algorithm a lot easier, compared to the modification of a game emulator, which is the approach taken by Murphy (2013).

V. Implementation

This section covers the abstract implementation details, that is, disregarding the quirks of the environment previously mentioned. To store the agent's "knowledge", i.e. the $Q(s,a)$ values for all state/action combinations, a two-dimensional array was chosen as the representation (henceforth called "knowledge array"):

$$\begin{array}{c c} & \begin{array}{cccc} Action_1 & Action_2 & \cdots & Action_n \end{array} \\ \begin{array}{c} \text{State}_1 \\ \text{State}_2 \\ \vdots \\ \text{State}_m \end{array} & \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \end{array}$$

As previously explained, our state space is defined by $\mathcal{S} = \mathcal{P}(SI)$, making the number of states $2^{|SI|}$. As a result, it is possible to represent every state in a bit representation of $|SI|$ bits, where each bit represents a single disjoint situation, which is active if the bit is set to one (this requires a pre-defined order for the disjoint states). The bit representation is then used, in the form of an integer, as the index for the State-dimension of the knowledge array[2].

The SARSA algorithm itself was implemented according to Sutton and Barto (1998), using the ϵ-greedy policy. The only difference is that, since our problem is not episodic, the outer (episode) loop of the algorithm was removed.

Algorithm 1 Non-episodic SARSA

```
  Initialize Q(s,a) arbitrarily
1 Choose a from s using ϵ-greedy policy
2 loop
3     Take action a
4     Observe reward r and new state s'
5     Choose a' from s' using ϵ-greedy policy
6     Q(s,a) ← Q(s,a) + α [r + γQ(s',a') − Q(s,a)]
7     s ← s'
8     a ← a'
```

VI. Reward strategy

In order to evaluate the implemented algorithm, it was still necessary to decide what rewards should be given to the Mario agent as it progresses (the reward function $\mathcal{R}$), plus the parameters of the SARSA learning algorithm, namely Alpha and Gamma.

Assuming a properly modeled state space and a correctly implemented algorithm, the correct choice of rewards and learning parameters is nonetheless crucial to achieve good learning behavior. In the absence of theoretical guidance, these parameters are often determined experimentally (Kamel, 2004), as is the case for this particular application. Thus at first, the following reward strategy was used (enemies were disabled at this point):

1. For every tick, give a reward of –1
2. If the agent stands still, give a reward of

[2] For example, given three disjoint situations A_1, B_2 and C_3 (the indexes indicating the order), the first dimension of the knowledge array would have $2^3 = 8$ elements. If one would like to know which index to use for the state $B_2 \cup C_3$, the equivalent bits would have to be flipped: $110_2 = 6_{10}$, revealing the index of interest.

"–1 * the number of time units it has not moved"

3. If the agent goes in the wrong direction, give a reward of "–5 * the number of segments it went in the wrong direction"

What could be observed with these rewards was that the agent was able to complete the level most of the time. Also, the initial learning process was quite fast, meaning that the agent knew which was the correct way to go after a few seconds into the simulation. This ability to quickly know which the correct direction is, was a result of the third reward rule. However, the negative sideeffect of this rule is that, if the agent got stuck in a situation where going back a few segments was necessary, it opted to just stand still or crouch indefinitely, since the penalty for going back would've been much higher than for not moving at all. As such, the third rule was eventually removed, resulting in a longer "warm-up" phase where the agent just randomly tries in which direction it should go.

After the agent was able to complete the level successfully without enemies, they were re-enabled. As the Mario agent starts the game being in the "Fire-Mario" state, it can collide with an enemy three times before the agent dies. Since the action of colliding with an enemy (and thus getting smaller) brings the Mario agent a step closer to death, it is given a reward of –50 for the collision with an enemy. For death itself it is also given a reward of –50.

Yet again, this reward strategy led to a satisfying result most of the time, i.e. the agent was often able to complete the level with enemies enabled. However, in some cases, directly after colliding with or dying from an enemy, the agent would simply start running into the wrong direction. This peculiar behavior was caused by a shortcoming in the modelling of the disjoint situations. The environment analysis, while trying to determine whether or not the agent was close to an enemy, only took into account the direction in which Mario was facing (as depicted in Figure 6, yellow square). As such, if an enemy "ran" into Mario from behind or above, and there was nothing dangerous in front of Mario, the penalty was applied to a state such as "Run Right", which made the agent scared of running in the correct direction. This problem was circumvented by adding two more situations ("enemy behind" and "enemy above", magenta and red squares in Figure 6 respectively) to the enemy detection, preventing enemies to "sneak up" on Mario.

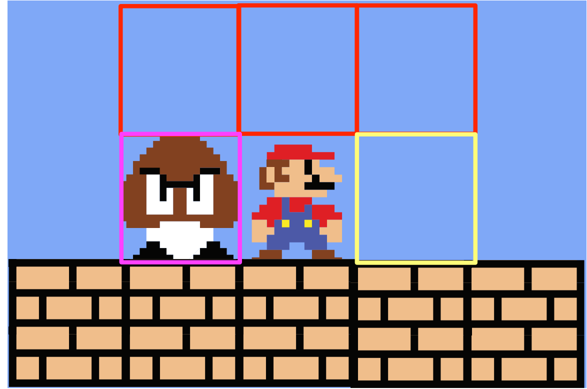


Figure 6: Enemy "sneaking up" on Mario due to faulty enemy detection.

VII. Evaluation

In Figure 7, the Mario agent's learning progression in a trial run is depicted. At the very beginning of the first round, the distance value decreases at some points. This means that the agent is still experimenting a lot, which includes running backward even if there is no obstacle. The inclination of the first rounds's graph is very small, since the agent has not yet learned that running is preferable to walking through the level. Mario dies early by walking into an enemy (the progression graph flatlines around tick 21).

The second round shows that the agent has already adopted a fairly good strategy. There are very steep segments in the graph, which indicate that the agent has learned to run. At the end of the second round, the agent collides with an enemy again, while running to the right. This has bad consequences for the third round, since a high penalty gets applied to the "run right" action, which propagates through the previous

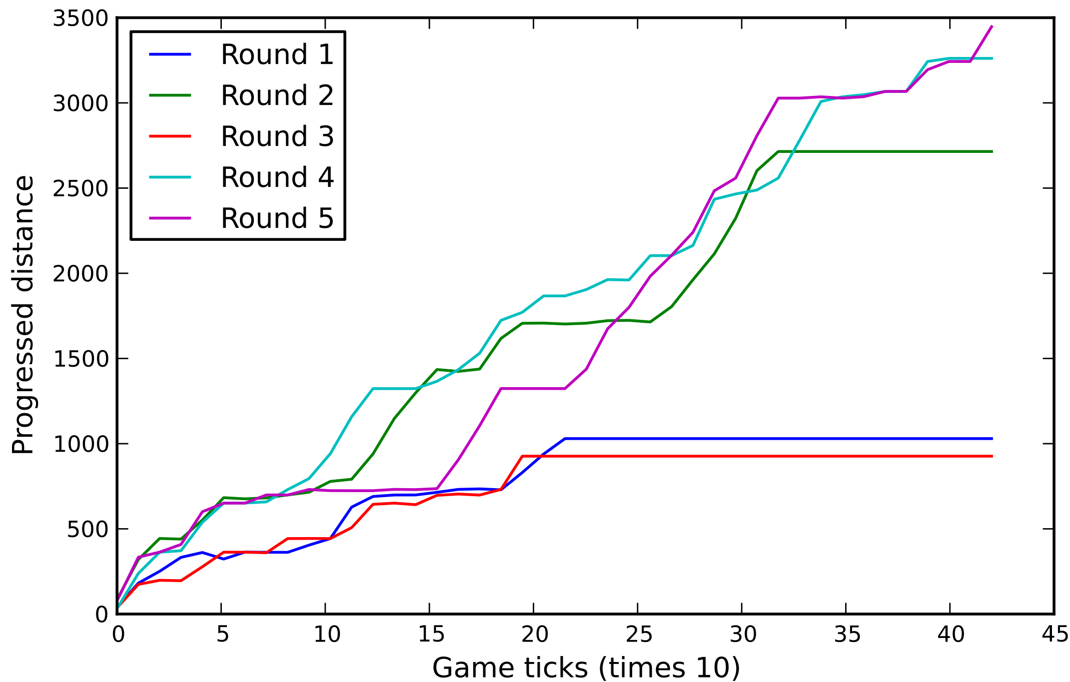


Figure 7: Agent progression in a trial run using the implemented algorithm. Progress is measured from the beginning of the level. For every "Game tick", the agent gets to decide which action to carry out.

actions. As such, the third round looks much worse than the second round.

In the fourth round, the agent has recovered from the results of the second round and almost completes the level, dying closely before the end. The fifth round is the first one where the agent successfully completes the level. After a very flat segment from tick 5 until tick 15, the agent runs almost until the end of the level, as can be seen by the very steep graph progression.

VIII. Future work

In order to further improve the learning rate of the Super Mario agent, Expected SARSA (Sutton and Barto, 1998) could be used. van Seijen et al. (2009) provide a theoretical and empirical analysis of Expected SARSA, and found it to have significant advantages over more commonly used methods like SARSA and Q-learning.

References

Kamel, M. (2004). Reinforcement learning and aggregation, *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)* **2**: 1303–1308.
URL: *http://goo.gl/JMmX9*

Mario AI Benchmark (2013).
URL: *http://goo.gl/qN6y4* (last accessed: 2013-06-08)

Murphy, T. (2013). The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel... after that it gets a little tricky.
URL: *http://www.cs.cmu.edu/~tom7/mario/mario.pdf*

Rodrigues, C., Gérard, P. and Rouveirol, C. (2008). On and Off-Policy Relational Reinforcement Learning, *Technical Report*, Université Paris-Nord, Paris.
URL: *http://goo.gl/9azNE*

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, Massachusetts. ISBN: 978-0262193986.
URL: *http://goo.gl/lrIlA*

van Seijen, H., van Hasselt, H., Whiteson, S. and Wiering, M. (2009). A theoretical and empirical analysis of Expected Sarsa, *2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning* pp. 177–184.
URL: *http://goo.gl/Oo1lu*