How to teach AI to play Games: Deep Reinforcement Learning

Mauro Comi Follow
Nov 15, 2018 · 9 min read

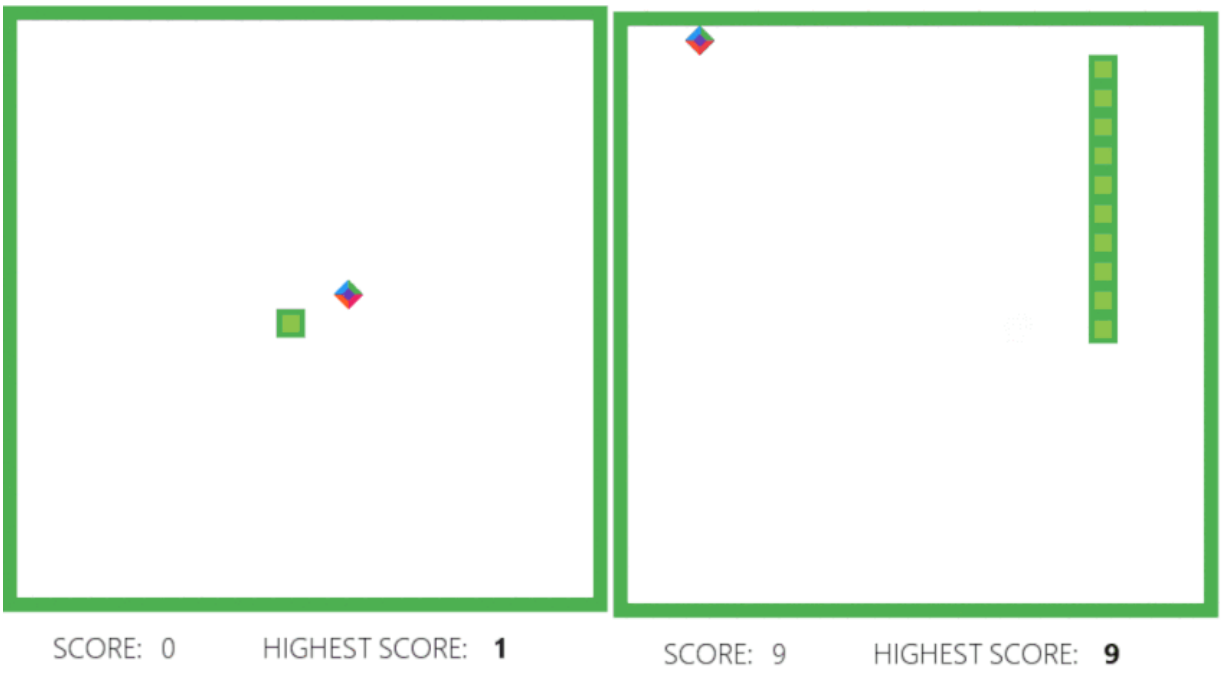
If you are excited about Machine Learning, and you're interested in how it can be applied to Gaming or Optimization, this article is for you. We'll see the basics of Reinforcement Learning, and more specifically Deep Reinforcement Learning (Neural Networks + Q-Learning) applied to the game Snake. Let's dive into it!

Artificial Intelligence and Gaming, contrary to popular belief, do not get along well together. Is this a controversial opinion? Yes, it is, but I'll explain it. There is a difference between Artificial intelligence and Artificial behavior. We do not want the agents in our games to outsmart players. We want them to be as smart as it's necessary to provide fun and engagement. We don't want to push the limit of our ML bot, as we usually do in different Industries. The opponent needs to be imperfect, imitating a human-like behavior.

Games are not only entertainment, though. Training a virtual agent to outperform human players, and to optimize its score, can teach us how to optimize different processes in a variety of different and exciting subfields. It's what Google DeepMind did with its popular AlphaGo, beating the strongest Go player in history and **scoring a goal considered impossible at the time**. In this article, we will see how to develop an **AI agent** able to learn how to play the popular game Snake from scratch. To do it, we implement a **Deep Reinforcement Learning algorithm** using Keras on top of Tensorflow**.** This approach consists in giving the system parameters related to its state and a positive or negative reward based on its actions. No rules about the game are given, and initially, the agent has no information on what it needs to do. The goal for the system is to figure it out and elaborate a strategy to maximize the score — or the reward.
We are going to see how a Deep Q-Learning algorithm learns to play Snake, scoring up to 50 points and showing a solid strategy after only 5 minutes of training.

For the **full code**, please refer to GitHub repository. Below I will show the implementation of the learning module.

The game



On the left, the AI does not know anything about the game. On the right, the AI is trained and learnt how to play.

The game was coded in python with Pygame, a library which allows developing fairly simple games. On the left, the agent was not trained and had no clues on what to do whatsoever. The game on the right refers to the game after 100 iterations (about 5 minutes). The highest score was 83 points, after 200 iterations.

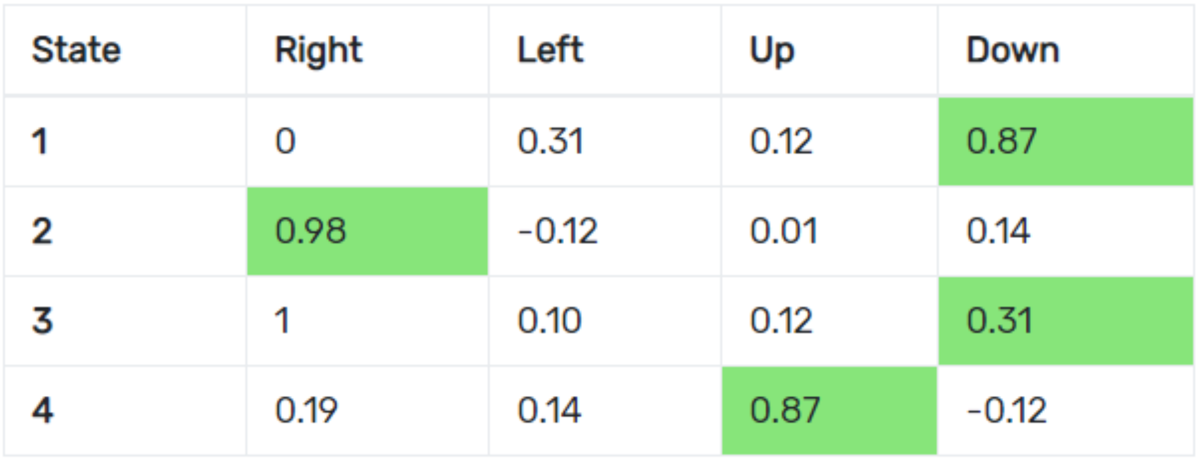
How does it work?

Reinforcement Learning is an approach based on Markov Decision Process to make decisions.

In my implementation, I used **Deep Q-Learning** instead of a traditional supervised Machine Learning approach. What's the difference? Traditional ML algorithms need to be trained with an input and a "correct answer" called target. The system will then try to learn how to predict targets based on unseen inputs. In this example, we don't know what the best action to take at each stage of the game is, so a traditional approach would not be effective.

In Reinforcement Learning, we have two main components: the *environment* (our game) and the *agent* (our Snake.. or to be correct, the Deep Neural Network that drives our Snake's actions). Every time the agent performs an action, the environment gives a **reward** to the agent, which can be positive or negative depending on *how good the action was from that specific state*. The goal of the agent is to learn what actions maximize the reward, given every possible state. S*tates* are *the observations* that the agent receives at each iteration from the environment. A state can be its position, its speed, or whatever array of variables describes the environment. To be more rigorous and to use a Reinforcement Learning notation, the decision-making process that the agent adopts is called **policy**. On a theoretical level, a policy is a mapping from the state space (the space of all the possible observations that the agent can receive) into the action space (the space of all the actions the agent can take, say UP, DOWN, LEFT and RIGHT). The optimal agent is able to generalize over the entire state space to always predict the best possible action.. even for those situations that the agent has never seen before! If this is not clear, worry not. The next example will clarify everything.

To understand how the agent takes decisions, it's important to know what a Q-Table is. A Q-table is a matrix that correlates the *state* of the agent with the possible actions that the agent can adopt. The values in the table are the action's probability of success, based on the rewards it got during the training.



State	Right	Left	Up	Down
1	0	0.31	0.12	0.87
2	0.98	-0.12	0.01	0.14
3	1	0.10	0.12	0.31
4	0.19	0.14	0.87	-0.12

Representation of a Q-Table

In the example, we might want to the RIGHT if we are in State 2, and we might want to go UP if we are in State 4. The values in the Q-Table represent the **expected reward** of taking action *a* from a state *s*. ***This table is the policy of the agent that we mentioned before:*** it determines what actions should be taken from every state in order to maximize the expected reward. What's the problem with this? The policy is a table, hence it can only handle a finite state space. In other words, we cannot have an infinitely large table with infinite states. This might be a problem for those situations where we have a very big number of possible states.

Deep Q-Learning increases the potentiality of Q-Learning, since the policy is not a table but a Deep Neural Network. The Q-values are updated according to the Bellman equation:

On a general level, the algorithm works as follow:

- The game starts, and the Q-value is randomly initialized.
- The system gets the current state **s** (the observation).
- Based on s, it executes an **action**, randomly or based on its neural network. During the first phase of the training, the system often chooses random actions to maximize **exploration.** Later on, the system relies more and more on its neural network.
- When the AI chooses and performs the action, the environment gives a **reward**. Then, the agent reaches the new state **state'** and it updates its Q-value according to the Bellman equation as mentioned above. Also, for each move, it stores the original state, the action, the state reached after performed that action, the reward obtained and whether the game ended or not. This data is later sampled to train the neural network. This operation is called **Replay Memory**.
- These last two operations are repeated until a certain condition is met (example: the game ends).

State

A state is the representation of a situation in which the agent finds itself. The state also represents the input of the Neural network.
In our case, the state is an array containing 11 boolean variables. It takes into account:
- if there’s an immediate danger in the snake’s proximity (right, left and straight).
- if the snake is moving up, down, left or right.
- if the food is above, below, on the left or on the right.

Loss

The Deep neural network optimizes the output (action) to a specific input (state) trying to maximize the expected reward. The value that expresses how good the prediction is compared to the truth is given by the Loss function. The job of a neural network is to minimize the loss, to reduce the difference between the real target and the predicted one. In our case, the loss is expressed as:

Reward

As said, the AI tries to maximize the expected reward. In our case, a positive reward is only given to the agent when it eats the food target (+10). If the snake hits a wall or hits itself, the reward is negative (-10). Additionally, there could be a positive reward for each step the snake takes without dying. In that case, the agent might just decide to run in a circle, since it would get positive rewards for each step. Sometimes, Reinforcement Learning agents outsmart us, presenting flaws in our strategy that we did not anticipate.

Deep Neural Network

The brain of the artificial intelligence uses **Deep learning**. In our case, it consists of 3 hidden layers of 120 neurons. The learning rate is not fixed, it starts at 0.0005 and decreases to 0.000005. Different architectures and different hyper-parameters contribute to a quicker convergence to an optimum, as well as possible highest scores.
The network receives as input the state, and returns as output three values related to the three actions: move left, move right, move straight. The last layer uses the Softmax function.

Implementation of the Learning module

The most important part of the program is the Deep-Q Learning iteration. In the previous section, the high-level steps were explained. Here you can see how it is implemented (to see the whole code, visit the GitHub repository. EDIT: since I am working on the expansion of this project, the actual implementation in the Github repo might be slightly different. The concept is the same as the implementation below).

```
while not game.crash:
            #agent.epsilon is set to give randomness to
actions
            agent.epsilon = 80 - counter_games

            #get old state
            state_old = agent.get_state(game, player1,
food1)

            #perform random actions based on
agent.epsilon, or  choose the action
            if randint(0, 1) < agent.epsilon:
                final_move = to_categorical(randint(0,
2), num_classes=3)
            else:
                # predict action based on the old state
                prediction =
agent.model.predict(state_old.reshape((1,11)))
                final_move =
to_categorical(np.argmax(prediction[0]), num_classes=3)
[0]

            #perform new move and get new state
            player1.do_move(final_move, player1.x,
player1.y, game, food1, agent)
            state_new = agent.get_state(game, player1,
food1)

            #set treward for the new state
            reward = agent.set_reward(player1,
game.crash)

            #train short memory base on the new action
and state
            agent.train_short_memory(state_old,
final_move, reward, state_new, game.crash)

            # store the new data into a long term memory
            agent.remember(state_old, final_move, reward,
state_new, game.crash)
            record = get_record(game.score, record)
```

Final results

At the end of the implementation, the AI scores 40 points on average in a 20x20 game board (each fruit eaten rewards one point). The record is 83 points.
To visualize the learning process and how effective the approach of Deep Reinforcement Learning is, I plot scores along with the # of games played. As we can see in the plot below, during the first 50 games the AI scores poorly: less than 10 points on average. This is expected: in this phase, the agent is often taking random actions to explore the board and store in its memory many different states, actions, and rewards. During the last 50 games, the agent is not taking random actions anymore, but it only chooses what to do based on its neural network (its policy).

> In only 150 games — less than 5 minutes — the system went from 0 points and no clue whatsoever on the rules to 45 points and a solid strategy!

Conclusion

This example shows how a simple agent can learn the mechanism of a process, in this case the game Snake, in a few minutes and with a few lines of code. I strongly suggest to dive into the code and to try to improve the result. An interesting upgrade might be obtained passing screenshots of the current game for each iteration. In that case, the state could be the RGB information for each pixel. The Deep Q-Learning model can be replaced with a Double Deep Q-learning algorithm, for a more precise convergence.

Feel free to leave a message for any questions or suggestions, I'll be more than happy to answer.

If you enjoyed this article, I'd love it if you hit the clap button 👏 so others might stumble upon it. For any comments or suggestions don't hesitate to leave a comment!

. . .

I am a Research engineer in love with Machine learning and its endless applications. You can find more about me and my projects at maurocomi.com. You can also find me on Linkedin, on Twitter, or email me directly. I am always up for a chat, or to collaborate on new awesome projects.

If you are interested in this article, you might like my other articles:

- Artificial Intelligence meets Art: Neural Transfer Style
- Is Artificial Intelligence Racist? (And Other Concerns)
- 3 Challenges of Artificial Intelligence in Computer graphics

Machine Learning Artificial Intelligence Gaming Deep Learning Data Science

[redacted]