Reinforcement Learning Project - The Snake Game

David El Malih
david.el-malih@student.ecp.fr
MSc in Artificial Intelligence
CentraleSupélec

Sai Deepesh Pokala
saideepesh.pokala@student-cs.fr
MSc in Artificial Intelligence
CentraleSupélec

October 1, 2020

1 Introduction

In this project (found here), we worked on developing an expert agent that learns to play the popular game of snake from raw pixels. To do this, we used an OpenAI Gym environment and implemented our own methods to facilitate playing of the game. Since the problem at hand involves dealing with pixel values, using Convolutional Neural Networks (CNNs) seemed to be the way to go ahead. The goal of the project is to study the behaviour of our Agent through learning epochs and analyze its performance.

2 The Game

The game is laid out on a square grid where a snake of a variable length is initiated. In each step of the game, the snake has **4 actions** to choose from; giving us the **action space - {up, down, left and right}**. The goal of the snake is to collect as many apples as it can that are generated as the game progresses. There is a **positive reward of +1** when the snake collects an apple and a **negative reward of -1** when the snake runs into its own body or hits the borders. Also, we have a added a **negative reward of -0.01** for each step that is taken within a given episode. This is done to make sure that that the agent not only learns to get an apple but also to get one as fast as possible.

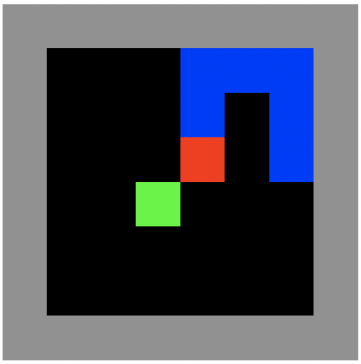

Figure 1: Game visualization

In the Figure 1, the apple is represented by the green pixel, the snake's body by the blue pixels and the snake's head by the red pixel. The grey pixels represent the walls of the grid. An episode ends either when the snake eats itself or runs into a wall. We chose a grid size of 8x8 as we felt it had a good trade-off between game complexity and computation time. Since the borders of the grid constitute of walls, the effective play area is a square grid of dimensions 6x6.

3 The Reinforcement Learning Problem

The task here, as in any Reinforcement Learning (RL) problem, is to take the current state of the game/environment and choose the best action that maximises the future reward. In our game, the transitions from state to state are not stochastic but predominantly deterministic. We will implement the State–action–reward–state–action (SARSA) algorithm in order to learn a Markov Decision Process (MDP) policy. Figure 2 depicts the way in which the agent and the environment interact with each other.

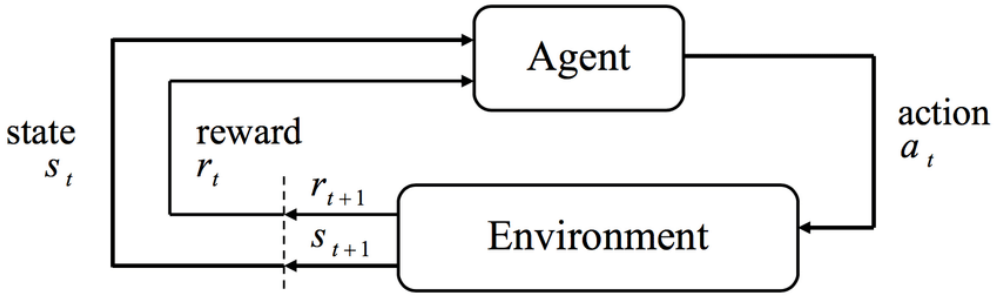

Figure 2: The agent-environment interaction

We would first like to discuss some terminologies before proceeding to discuss the RL problem:

- **Terminal states**: The states that has no available actions. In our case, the terminal states are when the snake runs into a wall or into its own body.
- **Episode**: An episode is a complete play from an initial state to the terminal state. In our case, episodes have the following pattern where s_t, a_t and r_t represent the state, action taken and reward obtained at step t. The number of states visited in an episode is given by n.

$$s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_n$$

- **Cumulative Reward**: It is the discounted sum of reward accumulated throughout an episode. It is given by the following formula where n is the number of states visited in the episode. The discount factor is represented by γ; for which we have chosen a value of 0.9.

$$R = \sum_{t=0}^{n} \gamma^t r_{t+1}$$

- **Policy**: A Policy is the agent's strategy to choose an action at each state. It is denoted by π.
- **Optimal Policy**: The optimal policy is the theoretical policy that maximizes the expectation of cumulative reward. From the definition of expectation and the law of large numbers, this policy has the highest average cumulative rewards given sufficient episode. This policy might be intractable.

The objective of RL is to train an agent such that its policy converges to the theoretical optimal policy.

3.1 Introduction to Q-Learning

As stated above, our goal is to learn the optimal policy π^* that maximizes the expected reward. The value function, which quantifies how good a state is for an agent to be in, is given by:

$$V_\pi(s) = \mathbb{E}\left(\sum_{t\geq 0} \gamma^t r_t\right) \quad \forall s \in \mathbb{S}$$

There exists an optimal value function that has the highest value for all states and it is given by:

$$V^*(s) = \max_\pi V_\pi(s) \quad \forall s \in \mathbb{S}$$

However, we still cannot control which state we end up in as the state we go to will depend on the actions we take. Therefore, we need a new function that takes in the state and action as an input pair and returns the expected total reward. This function is called the **Q-function**. This function quantifies the "quality" of an action in a given state. The term $Q_\pi(s, a)$ gives the expected reward when we take action a, following policy π, when in state s. Just like we defined an optimal value function, we can also define an optimal Q-function $Q^*(s, a)$ that gives the expected total reward for your agent when starting at s and picks action a and has the following relation with $V^*(s)$:

$$V^*(s) = \max_a Q^*(s, a) \quad \forall s \in \mathbb{S}$$

That is, the maximum expected total reward when starting at state s is the maximum of $Q^*(s, a)$ over all possible actions. Using these relations we can extract the optimal policy $*$ by choosing the action a that gives maximum reward $Q^*(s, a)$ for state s. Therefore, we have:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s, a) \quad \forall s \in \mathbb{S}$$

3.2 Learning Steps

The question now is that how do we use this Q-function to learn to play the game? To do this, we first define $Q(s, a)$ by recursive expression as the following:

$$Q(s, a) = r + \gamma \max_{a'} Q\left(s', a'\right)$$

This equation is known as the **Bellman Equation**. This equation tells us that the maximum future reward is equal to the summation of the reward r received for entering the current state s and the maximum future reward for the next state s'. Now, in order to learn, we need to iteratively approximate Q^* using the Bellman Equation. This is given by the following **update rule** in Q-Learning:

$$Q_{t+1}\left(s_t, a_t\right) = Q_t\left(s_t, a_t\right) + \alpha\left(r_{t+1} + \gamma \max_{a'} Q_t\left(s_{t+1}, a\right) - Q_t\left(s_t, a_t\right)\right)$$

where α is the learning rate that controls how much the difference between previous and new Q value is considered. A high value of α would mean updating our Q values in big steps. The idea is that as the agent explores more and more of the environment, the approximated Q values start to converge to Q^*.

3.2.1 Epsilon-Greedy Policy

In the Q-Learning process, while taking the next steps based on the Q-values of all actions in the current state, we can either follow a pure greedy approach ($\epsilon = 0$) or we can follow an ϵ-greedy approach ($0<\epsilon<1$). A pure greedy approach , wherein we always select the highest Q value among all the Q values for a specific state, is not ideal as there is a risk of getting stuck at the local optima. To combat this, we follow an ϵ-greedy approach, where we select an action at random with a probability of ϵ and we select the action with the highest Q-value with a probability of 1-ϵ. As the learning goes on, both the learning rate and ϵ should decay to stabilize and exploit the learned policy which converges to an optimal one.

3.2.2 The need for Deep Reinforcement Learning

In each episode of our game, at each step, there are 4 possible actions that the agent can choose. This means that for a given state, there are 4 possible 'next states'. And these 4 possible next states have have a combined total of 16 possible 'next states'. This number continues to increase exponentially and thus it is clearly evident that the state space is huge.

In the process of reaching the final terminal state, the agent has to go through a lot of states. This means that there are a lot of **(state, action)** pairs possible. When the game gets complicated, the knowledge space can become huge and it no longer becomes feasible to store all **(state, action)** pairs. Even a slightly different state is a distinct state. This requires us to take up a lot of memory to store all the possible states and time to explore them. In order to tackle this problem, and use something that can generalize the knowledge instead of storing and looking up every little distinct state, we introduce the concept of Deep Reinforcement Learning.

4 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL), or to be more specific, Deep Q-Learning in this case is nothing but the use of Deep Neural Networks (NNs) to effectively predict the Q-values for all the possible actions given an input state. Since we are dealing with pixels in this problem, we have used a Convolutional NN (CNN) to model our agent. Figure 3 specifies the input and output of the network.

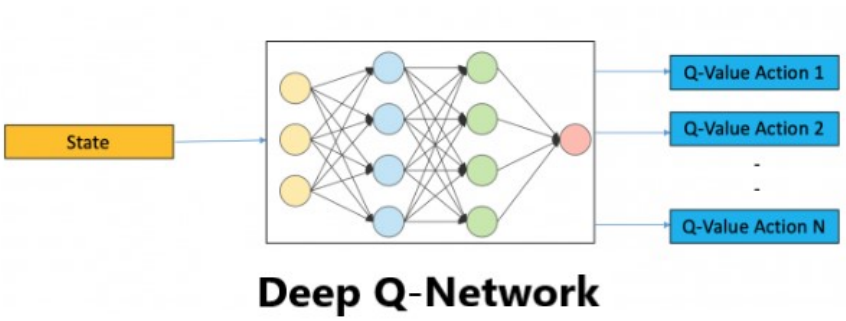

Figure 3: Input and output of a Deep Q-Network

In this approach, all the experience of the agent is stored in memory. The next action is determined by the maximum output of the Q-Network. But, as in the case of any NN, we need a loss function to back-propagate and facilitate learning. To derive the loss function, we work out the following:

By definition, the Q-function is given by,

$$\mathcal{Q}^{\pi}(s,a) = E_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t} r\left(s_t, a_t\right) | s_0 = s, a_0 = a\right]$$

By dropping the first term of the sum and using the linearity of the expectation, we have:

$$\begin{aligned} Q^{\pi}(s,a) &= E_{p^{\pi}}\left[r\left(s_0,a_0\right) + \sum_{t=1}^{\infty}\gamma^{i} r\left(s_t,a_t\right) | s_0 = s, a_0 = a\right] \\ &= E_{p^{\pi}}\left[r\left(s_0,a_0\right)\right] + \gamma E_{p^{\pi}}\left[\sum_{t=0}^{\infty}\gamma^{t} r\left(s_{t+1},a_{t+1}\right) | s_0 = s, a_0 = a\right] \end{aligned}$$

Now, from the Markovian property, we obtain:

$$\begin{aligned} \mathcal{Q}^{\pi}(s,a) &= E_{p^{\pi}}\left[r\left(s_0,a_0\right)\right] + \gamma E_{(s',a')p(.|s,a)}\left[\sum_{t=0}^{\infty}\gamma^{t} r\left(s_{t+1},a_{t+1}|s_1 = s', a_1 = a'\right)\right] \\ &= E_{(s',a')\sim p(.|s,a)}\left[r\left(s_0,a_0\right)\right] + \gamma E_{(s',a')\sim p(.|s,a)}\left[\sum_{t=0}^{\infty}\gamma^{t} r\left(s_t,a_t\right) | s_0 = s', a_0 = a'\right] \\ &= E_{(s',a')\sim p(.|s,a)}\left[r(s,a) + \gamma Q^{\pi}\left(s',a'\right)\right] \end{aligned}$$

If an optimal policy π^* exists, then the optimal Q-function is defined as:

$$\begin{aligned} Q^{*}(s,a) &= \max_{\pi} Q^{\pi}(s,a) \\ &= \max_{\pi} E_{(s',a')\sim p(.|s,a)}\left[r(s,a) + \gamma Q^{\pi}\left(s',a'\right)\right] \\ &= E_{s'\sim\pi^{*}(.|s,a)}\left[r(s,a) + \gamma Q^{\pi^{*}}\left(s',a'\right)\right] \\ &= E_{s'\sim\pi^{*}(.|s,a)}\left[r(s,a) + \gamma \max_{a'} Q^{*}\left(s',a'\right)\right] \end{aligned}$$

Let $Q^\theta(s,a) = Q(s,a,\theta)$ be our Q-function that needs to be approximated by our nural network. We want $Q^\theta \approx Q^*$.

$$Q^\theta(s,a) = E_{s' \sim \pi^*(.|s,a)} \left[r(s,a) + \gamma \max_{a'} Q^\theta(s',a') \right]$$
$$E_{s' \sim \pi^*(\cdot|s,a)} \left[r(s,a) + \gamma \max_{a'} Q^\theta(s',a') - Q^\theta(s,a) \right] = 0$$
$$\left\| E_{s' \sim \pi^*(.|s,a)} \left[r(s,a) + \gamma \max_{a'} Q^\theta(s',a') - Q^\theta(s,a) \right] \right\|^2 = 0$$

Hence,

$$E_{s' \sim \pi^*(\cdot|s,a)} \left\| r(s,a) + \gamma \max_{a'} Q^\theta(s',a') - Q^\theta(s,a) \right\|^2 = 0$$

From the above equations and due to the deterministic nature of r, the loss function that has to be minimized becomes:

$$\mathcal{L}(\theta) = E_{s' \sim \pi^*(\cdot|s,a)} \left\| r + \gamma \max_{a'} Q(s',a',\theta) - Q(s,a,\theta) \right\|^2$$

So now that we have a loss function that can help facilitate our model learn, let us address the final challenge in Deep Reinforcement Learning - **the challenge of non-stationary targets**. The target is continuously changing with each iteration. In Deep Learning, the target variable does not change and hence the training is stable, which is just not true for DRL. We depend on the policy or value functions to sample actions. However, this is frequently changing as we continuously learn what to explore. As the game progresses, we get to know more about the ground truth values of states and actions and hence, the output is also changing.

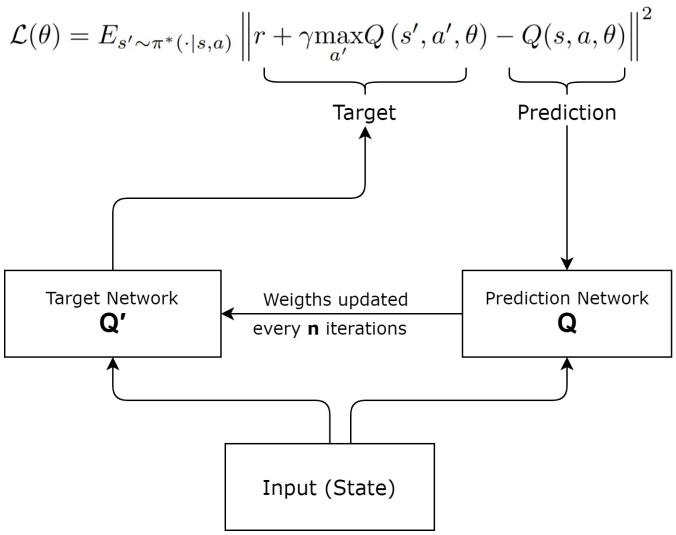

Figure 4: Approach for Deep Q-Learning

To overcome this problem of non-stationary targets, as seen in Figure 4, we use two networks instead of one. This is because if we use one network, the same network will be calculating the predicted value as well as the target value. This is not very optimal as there can be a lot of divergence between the two values. Therefore, we use a separate network to estimate the target. This target network Q' has the same architecture as the prediction network Q, but with frozen parameters. For every n iterations, the parameters of Q' are updated with those of Q. This leads to more stable training because it keeps the target function fixed for a while.

5 Results

After configuring our network, we initiated the agent's training on 10000 episodes. The following is the plot that visualizes the performance of the agent.

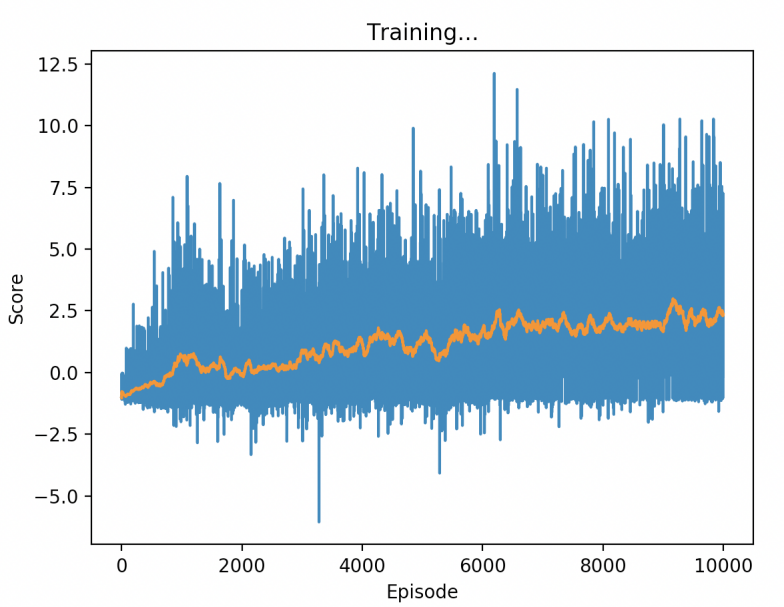

Figure 5: Performance of the agent over episodes; the orange curve indicates the moving average over 100 episodes; the blue curve indicates the score per episode.

From Figure 5, it can be observed that globally, the agent learns to play the game better and better with the increase in the number of episodes played. However, it can also be observed that there are several intermediary highs and lows in the plot. This can be attributed to the exploration v/s exploitation dilemma that is common to any RL problem. As the training progressed, the ϵ value kept decreasing for our ϵ-greedy approach. Therefore the performance of the agent in a particular episode depends on the quality of the knowledge acquired and the number of random states explored.

6 Conclusion

This project gave us a good opportunity to explore the world of Deep Reinforcement Learning. Learning from raw pixels made it even more interesting as it was similiar to how humans learn to play. However, there are some shortcomings/limitations, some of which can be addressed in the future:

- **Huge hyperparameter space:** There are a lot of hyperparameters that can be tuned, some of which correspond to the RL problem whereas the others correspond to the game and the network. Some RL hyperparameters include the ϵ values, the decay rate for the ϵ value and the discount factor γ. On the network side, some hyperparameters include the update frequency of the target network, the batch size for training and the memory capacity of the network.
- **Model Architecture:** Apart from the hyperparameters, the core CNN model which is the main part of our network can also be configured in many ways. So, a better structured CNN-model can give better results.
- **Grid Size:** Our Snake implementation was on an 8 x 8 square grid. Playing and learning from a grid of higher dimensions would require us to adapt the CNN model accordingly. Therefore our implementation is not readily scalable.