Deep Q-Learning

Begu Maria-Florina(3E2)

Enia Vlad Ieftimie (3E4)

December 2021

Abstract

This paper represents a study on Deep Q-Learning and how it can be used to solve a specific problem, like helping the agent to reach the maximum score in the Waterworld game. In this way, the algorithm's advantages can be observed, like how the agent learns the environment and which course of actions are the best for it. By testing the results, we were able to change the hyper-parameters or the way that the neural network is initialized, in order to obtain the best results on the given problem.

1 Introduction

Reinforcement learning represents an area of machine learning which is based on rewarding desired behaviours and punishing the undesired ones. In this way, the trained agent is able to perceive and interpret its environment, take actions and learn through trial and error.[1]

One of the core concepts in Reinforcement Learning is the **Deep Q-Learning algorithm** which uses neural networks to help the agent adapt to the new environment and act in order to get closer to the goal.

In order to have a better understanding of deep q-learning, the method was used to solve a specific problem, which is WaterWorld. This game consists in the agent, a blue circle, which must navigate around the world capturing green circles while avoiding red ones. The game ends when all the green circles have been captured. [3]

In the following sections, there will be presented the idea of solving this problem, a more precised description of the used algorithm, implementation ideas and the outcomes of the experiment and finally, the conclusions regarding this method and why it is usually used to solve difficult problems.

1.1 Motivation

This game could be solved using an agent that has a random behaviour or other methods that does not involve learning. Even if they may succeed, in some situations, in finishing the game, it is clear that they are not the most optimal options. In order to obtain the best results each game, a trained model obtained by using deep q-learning may be a better option.

1.2 Problem Description

In WaterWorld the agent, a blue circle, must navigate around the world capturing green circles while avoiding red ones.[3]

After capturing a circle it will respawn in a random location as either red or green. The game is over if all the green circles have been captured.[3]

Valid Actions: Up, down, left and right apply thrusters to the agent. It adds velocity to the agent which decays over time.[3]

Terminal states (game_over): The game ends when all the green circles have been captured by the agent.[3]

Rewards: For each green circle captured the agent receives a positive reward of +1; while hitting a red circle causes a negative reward of -1.[3]

2 Method

2.1 Deep Q-Learning(DQN)

DQN is a reinforcement learning algorithm which uses neural networks in order to train the agent, learn more about the environment and predict future actions that will help to achieve the goal. At the beginning, there will be created 2 identical neural networks: one will represent the model that we want to train and the other the target model, which will provide the values wanted for the first model.

When a game starts, for every frame of it, an action will be chosen. The selection process is influenced by a value ϵ, which dictates if the algorithm will "explore", by choosing a random action, or will "exploit", by using the action which our model considers the best. Depending on the taken action, both the score and the state of the game will be updated, followed by retraining the model.

The target model will not be adjusted as often, but after a number of steps. The adjustment will be done by copying the model's weights to the target.

Also, after each game, ϵ will decrease. This happens because "exploration" is more wanted at the beginning, but as we train the model, this will offer better results and as a consequence, "exploitation" is gonna be more desired.

3 Implementation

In order to train the model, there were used the following functions:

1. **def agent(state_shape, action_shape)**: It is used to initialize the neural networks, which have 3 hidden layers, first 2 containing 100 neurons and the third having 50. As activation it was used the Rectified Linear Unit function

($relu(x) = max\{0, x\}$) and as regularization: L2 (weight decay). The function returns the initialized network.

2. **def preprocess_state(state)**: It changes the state given by the game to a state more suitable for solving the problem. From all the information kept by the formal state (position_x, position_y, velocity_x, velocity_y, player distances to each creep, each creep position), it saves only the most useful one(position_x, position_y, velocity_x, velocity_y, player distances to the closest GOOD and BAD creep, the closest GOOD creep and the closest BAD creep). We don't need to keep all the distances or all the coordinates of the creeps, because the ones near to it dictates how the blue particle will act.

3. **def train(replay_memory, model, model_target)**: From the memory, it is selected a random batch (size=100). The batch will contain tuples with the following structure: (state,action,reward,state_next, done). After the selection, there will be computed the future rewards (on the target model) which are used in order to find out the desired output of our model and q_values which are the actual output of the model. With their help, we can find the loss value and apply backprpopagation on our neural network.

For all the episodes, we initialize the score with 0 and find the first state. For each frame, we decide what action should we take with the help of ϵ and compute the reward, the next_state and verify if the game is done. These values together with the state and the action are going to be saved to the memory and at each 4 frames or when the episode is done, we are going to train our model. At each 1000 steps, we copy the weights from the model to our target model. Finally, when the game is done, we change the value of ϵ in the following way: if it's bigger than min_epsilon, we make it smaller, if the number of episode is bigger or equal to 700 we make ϵ to be equal to min_epsilon for the rest of the games (we do this because it can be observed that the model is well trained at that point and we can keep the exploration minimal, but not 0, so in some rare situations the blue particle will still move randomly). There can also been seen an increase of ϵ at episodes 400,500,600. It was implemented this way, in order to explore a little bit more in case that there are some better actions than the ones that the model's. These exploration episodes are short, in order to explore, but not so much that it can affect in a negative way our training.

4 Experiments' Description

4.0.1 Time

4.1 Observation

5 Conclusion

References

[1] https://www.techtarget.com/searchenterpriseai/definition/reinforcement-

learning

[2] https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc

[3] https://pygame-learning-environment.readthedocs.io/en/latest/user/games/waterworld.html