Deep Q-Network on Acrobot

Mushaidul Islam, Shijon Das, Tanzim Mostafa
Department of Computer Science and Engineering
Brac University - Mohakhali, Dhaka, Bangladesh
Email: mushaidul.islam@g.bracu.ac.bd, shijon.das@g.bracu.ac.bd, tanzim.mostafa@g.bracu.ac.bd

***Abstract*—With the Q-learning algorithm, we need function approximators such as artificial neural networks to memorize the triplets (state, action, Q-value). With Deep-Q learning (DQL), we use neural networks to predict the Q-values for each action given the state. In the Acrobot game, we used Deep-Q Learning algorithm to train the agent. We define the loss function as the difference between the predicted Q-value and the target Q-value. We initialized two identical artificial neural networks, called Target Network and Policy Network. The first will be used to calculate the target values, while the second to determine the prediction.**

***Index Terms*—Reinforcement Learning, Q-Learning, Deep Q-Learning, Acrobot**

I. Introduction

The training of machine learning models to make a series of judgments is known as reinforcement learning. In an uncertain, potentially complex environment, the agent learns to achieve a goal. Artificial intelligence meets a game-like circumstance in reinforcement learning. To find a solution to the problem, the computer uses trial and error. Artificial intelligence is given either rewards or penalties for the acts it takes in order to get it to accomplish what the programmer desires. Its purpose is to increase the total prize as much as possible.

Despite the fact that the designer establishes the reward policy–that is, the game's rules–he provides the model with no tips or ideas for how to solve the game. Starting with completely random trials and progressing to sophisticated tactics and superhuman skills, it's up to the model to find out how to do the task in order to maximize the reward. Reinforcement learning is currently the most effective technique to hint at machine creativity by utilizing the power of search and many trials. Artificial intelligence, unlike humans, can learn from thousands of simultaneous games if a reinforcement learning algorithm is performed on a sufficiently powerful computer.

II. Challenges

The most difficult aspect of reinforcement learning is setting up the simulation environment, which is very dependent on the job at hand. Preparing the simulation environment for the model to go superhuman in Chess, Go, or Atari games is pretty straightforward. When it comes to developing a model capable of driving an autonomous vehicle, creating a realistic simulator is essential before allowing the vehicle to drive on the road. The model must find out how to brake or avoid a collision in a safe environment, where the cost of sacrificing a thousand automobiles are negligible. The challenging part is getting the model out of the training environment and into the actual world.

Another problem is scaling and adjusting the neural network that controls the bot. There is no other method to communicate with the network but through the reward and punishment system. This could result in catastrophic forgetting, in which new knowledge causes some old knowledge to be lost from the network.

Another difficulty is achieving a local optimum, in which the agent completes the task as is, but not in the optimal or needed manner. A "jumper" who jumps like a kangaroo instead of walking is an excellent example. Finally, some agents will optimize the prize without completing the objective for which it was created.

III. Deep Q Learning

Deep Q-Learning is one of the most popular algorithms in Reinforcement Learning. While there exists another algorithm called Q-learning which is similar to this algorithm, deep Q-learning is required when the environment consists of a large number of states. For example, if an environment consists of 10,000 states and there are 1,000 actions per state, this means we would need a matrix with 10 million elements. Hence, it would be impossible to perform this task using our normal method. This also means that most of the interesting problems, such as chess, would not work using the Q-learning method.

To be more precise, this presents two difficulties:

A) As the number of states increases, the amount of memory required to save and update that table would also rise.

B) The total time required to explore each state and create the required Q-table would be unfeasible.

Hence, as a solution to this problem, a new idea was thought of that instead of recording Q-value precisely with matrix, they would be approximated with machine learning models such as a neural network. This is basically known as Deep Q-Learning.

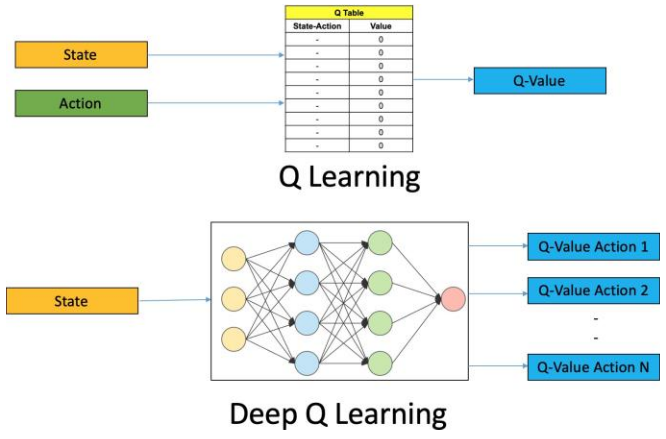


Figure 1. Describes the difference between Q-learning and Deep Q-learning

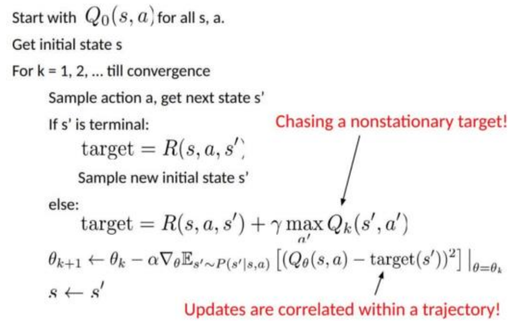
So, in Deep Q-learning, we use a neural network to approximate the Q-values. As an input, the states are given while as output, we get the Q-value of all possible actions. The following are the steps used in reinforcement learning using deep Q-learning networks (DQNs):

1) The user stores all the previous experience in memory.
2) The maximum output of the Q-network determines the next action.
3) The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This means that it is essentially a regression problem. The only difference here is that, as this is a reinforcement learning problem, we do not know the actual value or target value. If we go back to the Q-value update equation derived from the Bellman equation, we have:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\boxed{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)} - Q(S_t, A_t) \right]$$

Figure 2. Updated Q-value

The boxed part of the equation represents the target. Although it may seem that it is predicting its own value, the network will update its gradient using back-propagation to eventually converge because R is the unbiased true reward.



```
Start with $Q_0(s, a)$ for all s, a.
Get initial state s
For k = 1, 2, ... till convergence
    Sample action a, get next state s'
    If s' is terminal:
        $\text{target} = R(s, a, s')$
        Sample new initial state s'
    else:
        $\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$
    $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[ (Q_\theta(s, a) - \text{target}(s'))^2 \right] \big|_{\theta=\theta_k}$
    $s \leftarrow s'$
```

Chasing a nonstationary target!
Updates are correlated within a trajectory!

Figure 3. Pseudo code of Deep Q-learning Algorithm

The above deep Q-Learning algorithm consists of some challenges. The main one being that the target is continuously changing with each iteration. A solution to this is using two neural networks instead of one, to calculate the predicted and target value. Another method that can be used is something known as Experience reply. Here, the system stores the data discovered for [state, action, reward, nextState] in a large table, instead of running Q-learning on state/action pairs as they occur during simulation or in the actual experience. These all combine to help make the deep Q-learning algorithm much better.

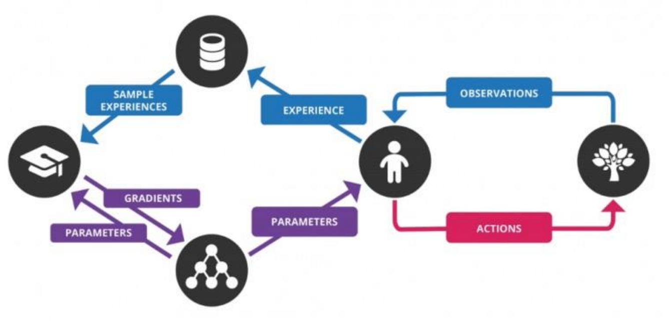


Figure 4. Depicts how the overall algorithm works

IV. MODEL

For our model we have decided to simulate an acrobot system that includes two joints and two links, where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height. In the beginning, the agent does not have any previous knowledge of how its actions affect the environment. By playing the game several times, the agent starts to learn and eventually reaches a point where it acts optimally. The states in this system refer to the condition the robotic arm is currently in. In our case there are 6 states in total.

1) sin and cos of the two rotational joint angles (total = 4)
2) the two angular velocities

The way we set it up, in each time step, our agent receives -1 as punishment until the goal is reached. Whenever the robotic arm performs an action, it either receives a reward or does not. Therefore, the task of the agent is to recognize which actions can maximize the total reward. It basically has three possible actions:

1) Apply positive torque (+1)
2) Apply negative torque (-1)
3) Do nothing (0)

Also, each episode has a maximum of 500-time steps, in case the agent is unable to reach its goal before. This means that the worst total reward is -500.

At first, we created an environment using OpenAI Gym. Then we built a deep learning model using Tensorflow Keras API. This model is then passed to Keras reinforcement learning (RL) in order to train our reinforcement learning model using policy-based learning.

So basically, our DL model is going to learn the best action to take in that specific environment in order to maximize our score. In order to create our DL model, we are first instantiating our sequential model then we are passing through flatten nodes and specifically to that, we are going to be passing through a flat node that contains our different states. Then we add two dense nodes to start building our DL model with a Relu activation function. Lastly, our last dense node has our actions. This means we pass through our states at the top and we pass through our actions down the bottom. So, ideally what we should be able to do is train our model based on the states coming through to determine the best actions to maximize our reward or our score that we can see here.

Now we take this DL model and train it using Keras RL. We imported deep Q-network agent. So, basically, we have a bunch of different agents within the Keras RL environment to train the model. We are using DQN here. We also have a specific policy. We got value-based RL as well as policy- based RL. The policy used here is the Boltzmann Q policy. For, our DQN agent we are going to maintain some memory. The sequential memory class allows us to do that. To that function, we passed through our model-the DL model and also the different actions that we can take. Our DQN model now starts to train. We actually instantiate our build agent function to set up a new DQN model and we passed through our model as well as our actions. As a result, we get a better set of scores for each episode. The below graph Figure 5 shows how the scores compare with each episode. Now, we have scores which are less negative, thus showing us that simulated system has improved, and the agent can take actions which are more rewarding.

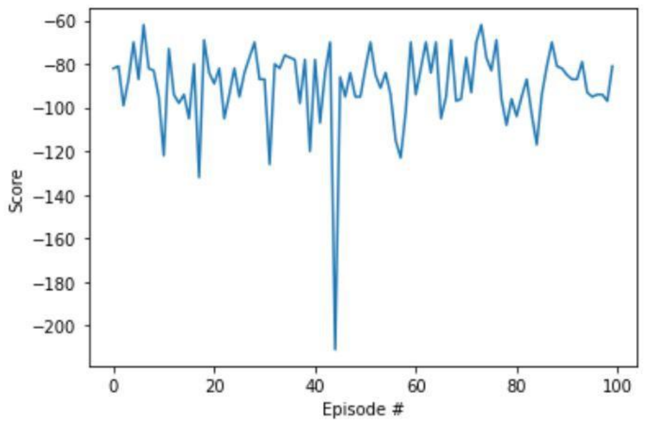


Figure 5. Graph depicting Score against Episodes after training

V. CONCLUSION

To conclude, in this paper we have discussed on what reinforcement learning is and two of its popular algorithms. We have explained the difficulties that arises while using simple Q Learning method to approach a reinforcement learning problem. To further support this claim paper, we have introduced Deep Q-Learning and how it uses neural network to approximate the Q-values. Then, we have used the algorithm to demonstrate its usage on a simulated acrobot system, trained the model to perform better and display improving results. In the future, we will use more difficult environment to run the Deep Q-Learning on, to observe and investigate its behaviour.

REFERENCES

[1] Choudhary, A.(2019, April 18). A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python. Analytics Vidhya. Retrieved September 23, 2021, from https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

[2] Osin´ski, B., Budek, K. (2021, January 5). Deepsense.Ai. https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide

[3] Wang, M. (2020, November 18). Deep Q-learning tutorial: Mindqn. Medium. Retrieved September 23, 2021, from https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc

[4] Deep Q-learning - combining neural networks and reinforcement learning. deeplizard. (n.d.). Retrieved September 23, 2021, from https://deeplizard.com/learn/video/wrBUkpiRvCA