Karolina Drabent
Patryk Fijałkowski
Artur Haczek
Krzysztof Kamiński
Pamela Krzypkowska

June 6, 2019

Abstract

This short paper will cover the topic of Q-Learning tested on an Agent base input game, which in our scenario is a simple Snake game. We will first explain what Q-Learning is, how and why it works and what is it used for. Later we will present our solution with the usage of Q-Learning methods which goal is to teach an Agent to play our Snake game, maximise its score and amount of moves per game. Finally we would show how our method is working in practise, what changes could be made further on as well as different approaches that we had encountering numerous problems along with their actual outcomes.

1 Background Knowledge

1.1 Q Learning

Q Learning is one of the Reinforcement Learning paradigm methods. This is a method somewhere close to unsupervised learning, which is a Machine Learning method which centres around learning without a teacher. A teacher is usually represented by a data-set or a set of labels and observed outcomes. Q Learning is based on a reward given to an agent after an action the agent has performed in a given environment. The environment is always a model providing positive or negative rewards to agent.

The goal of this learning method is to teach an agent, which decisions to make under which circumstances. Therefore the agent's goal is to create its own model of the environment, not known to the agent at all before the active learning process starts. The agent is changing its future decisions based on the rewards from its actions - building the model previously mentioned.

The classic Q Learning is based on a Q table which is a reward function presented as a matrix. This function returns the value of a Q function having the previous action chosen being A and previous state (before performing a given action A) being Q (The function will be explained later, the main focus is that the system counts the values in Q table based on previous state, action chosen

and reward given from the environment). Firstly the Q table is filled with a neutral element (it can also be any arbitrary value chosen by a programmer), when we are using real numbers the neutral element would be 0. Therefore the Q table has a following function signature:

$$Q : S \times A \to \mathbb{R} \tag{1}$$

The process of updating values in the Q table which tells us what reward can be given to an agent being in state Q, performing action action A is made based on iterations. Before deciding the next actions, agent checks the Q-table and learns which action would be the most profitable based on its previous knowledge of rewards provided. Every iterations gives the system more information because after every action it updates the Q table with the new value of reward-based-value given on action A performed from state Q. The value put inside the Q table is obtained with the following equation 1, presented below:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Figure 1: Q function

The values such as the learning rate and discount factor are there to be tweaked by human agents building the model. Learning rate being a level to which new information overrides old information (how much the Q table is going to be changed after new information is gained) and discount factor is a way of making agent more far-sighted, so it would also value long term rewards that might be gained, not only short term rewards which might be smaller.

The system finished the learning when it reaches the final state. After the final state, the model (meaning the Q table) can be used for next iterations to enable to agent to teach itself better while working with the environment, updating its knowledge with every iteration. It's important to note that there might be no final state, and the Q Learning algorithm can also be used in such situations.

1.2 Deep Q Learning

The method we have chosen in our project is a Deep Q Learning, which is a method of Q Learning using Deep neural networks. In Deep Q Learning there is no Q table, which is a model of environment created by the agent. The model of the environment is contained in the network. This network can be called a Q Network, this network outputs the predicted reward for the agent, given actions it can perform.

1.3 Agent input games

Snake is a very simple game, which originated back in the XXth Century. It is quite popular to use simple games in which a human agent inputs sequences, to showcase popular Machine Learning algorithms, especially neural networks. In this examples a input usually generated by a human is changed so that the input is generated by a computer program (we can also call it an agent).

Having games with points to collect or score like Pacman, Space Impact or Snake it is an interesting case study to implement reinforcement learning methods, trying to make the agent learn to play the game from externally provided rewards. Rewards correlate with point that the player is collecting and the time for which it can stay alive. Many of those games not only enable the player to score points, but also have some actions lead to a death of a player and loosing all points. A learning agent is then facing a problem of optimizing the risks along with potential gains.

It is quite important to note that this is a simplified model of a very low-level human-like learning process based on rewards.

2 Our methods

2.1 Environment

As mentioned before we are training an agent to learn how to play snake. I won't get into details of architecture of the game itself, because it is not too important. The crucial part lies in a model of the environment and what we want to teach our agent. We want to teach our snake to learn how to score as many points as possible and live as long as possible. Points are scored when the agent is moving on a board of fixed size, and is eating apples. The maximum number of points available for the player is bounded by the board size. The game UI is showed on the picture 2 below:

Figure 2: Our snake game UI

The first thing we need for our agent was to implement a way to tell the agent the state of the game, our environment state. The way we structured our state is shown on the picture 3 below:

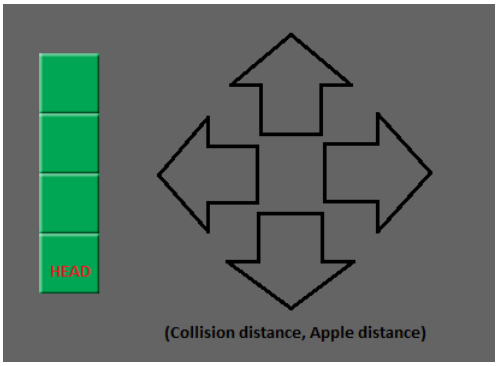

Figure 3: Our snake game UI

As we can see we give the agent information in four directions. It is because the agent can only choose from four possible moves, top, down, left and right. The information is for every direction is the distance to the nearest collision and distance to the apple (the point). The distances were counted in multiple ways, because as it is obvious not all of them were equally good. The collision distances were almost always the same - how many moves in this direction snake needs to make to die, the apple distance though varied. We started from Euclidean distance to the apple from every 'next possible move' and finally based our model on a distance which gave zero when the apple was not in this direction at all and binary 0/1 when apple was in the direction given, with partial points in the range from 0 to 1 on apple being somewhere diagonally. So finally the state is counted in a given way presented with a simple algorithm 1:

Algorithm 1 Count state

```
1:  procedure CountState
2:      applePos ← position of the apple
3:      snakePos ← position of head
4:      for d in directions do
5:          collisionD ← distance to the nearest Wall/Snake in this direction
6:      for d in directions do
7:          if apple is in d then
8:              add 1 to state
9:      if appleD is  empty then
10:         count partial apple distance
        return collisionD  + apple
```

The only problem which we tried to solve and still didn't find the proper answer to is, if our state vector has only positive values (let us make this assumption), should the neural network be able to make out the subtle differences that some cells in the vector, if they are positive, give the model greater chance of getting a bigger reward, on the other hand some other positive values might stand for negative things - like collisions. Are the neural networks sophisticated enough to catch that? We would surely assume that it should but we did have some problems when we tried this method for collision checking.

This was presented the final way of counting the state before putting it into the neural network which is set to provide us with the values with which the neurons corresponding to each action/move is fired. From the neurons activity the agent is choosing the mostly rewarded action and that's how the process is going in an iterative way.

The problem of giving the agent a proper reward is also important. As mentioned before reward is the key factor that is enabling the Agent in Q Learning to teach itself how to perform well (according to what we, humans, tell him is a good way of acting). To give the best outcomes reward is given as a real number from a range $(0, 1)$. Having normalised reward makes some problems more visible, cause the range is small enough for us to notice and count easily the reward percentage.

We tried various rewards from binary to real and normalised and also not normalised at all. As it seems to be, normalised reward do gives the best outcome but sometimes, when it is too precise it gives a feeling of guiding the snake holding a carrot on the stick, instead of actually teaching him. But this is a subject of a rather different dispute.

Our reward we agreed upon can be counted in a such way, presented below on 2:

Algorithm 2 Count reward

```
1: procedure CountReward
2:     if snake is not alive then return deathReward
3:     else
4:         appleDist ← distance from the apple
5:         if appleDist is 0 then return eatReward
6:         elsereturn liveReward + eatReward / appleDist
```

It is crucial to remember that we need to reward for getting the point, but staying alive is also a good feat. In a learning process when the game has more complex goals, reward must be somehow split between those things and we have to decide what is more valuable and how to balance the output given for the agent.

As we have learned it is very hard to train the agent the importance of both living and eating apples, thus getting points. Changing the reward from a lot points for living and less points for eating led to the Agent neglecting the point collecting whatsoever but focusing on getting to the point where the apple

was is also hard because the Agent has to move around in the 'city metrics', having to chose only from four directions. The apple distance itself (counted as a percentage) is also not the best way for counting the reward because the distance which we can get changing our current position to some other is at least as much important as is the direction right. Also how to balance getting closer to the apple with getting closer to the wall and a spike of the negative values?

Complicated environments must have it very hard to present the right reward for the right action, optimizing for gains and losses.

On Game Over the environment is taken back to the starting state which is having the snake of length four, facing the bottom direction with an apple in a random spot of the board. The neural network is preserved with every iteration and the information is later used for training.

2.2 Training

Apart from iterative information collection, it is important to train the agent, based on its previous experiences. Our training at first was done every thousand game, but we figured out that the results were not too good, so the final training is done after every game. And it last the length of the game.

Training is based on a list of samples collected from the agent previous actions and given rewards. One record in a training list is a tuple:

$$QRecord = (PreviousState, Reward, Action, NextState, GameOver) \quad (2)$$

The relation between the *Previous State* and the *Next State* is that between them the agent chose to perform *Action*. *Reward* is given to us by the environment and *Game Over* component is telling us if after the Action the game has ended. In this sense *Game Over* is an additional element helping the agent understand the environment and the rules of the game.

Having the *Game Over* information for a record we do not use the full formula, this is the Q equation to count the training output, because it involves predicting the next state, next state in case of game being over is None or a new game, so we just input the reward given by the network.

During every game information from every move is collected to the list of previous actions, and then the batch is chosen for training, being split into mini batches, so that every training session would be split into separate smaller trainings. Every smaller training is based on inputting the collected mini batch into the neural network.

So our training is flexible in a sense that it can be done in many stages of the learning process. The training still does not break the no-teacher policy, because the agent is only tweaking the model based on previous experiences. It is important for a programmer to change the frequency and locations of the trainings because it can greatly affect the results.

Our training is working in a very simple way which is presented in a algorithm 3 below:

Algorithm 3 Train model

```
procedure TRAIN MODEL
    n ← (0, 10)
    batchSize ← random size in (0, len(ExperienceList)
    miniBatchSize ← batchSize/n
    for miniBatch in batch do
        for elem in miniBatch do
            add elem.PreviousState to trainInput
            if elem.GameOver == True then
                add elem.Reward to trainOutput
            else
                add (1 - gamma) * elem.Reward + gamma *
 model.predict(elem.NextState) to trainOutput
    model.train withtrain.Input, train.Output
```

As we have learned there are numerous training practises and the variation of this one if so far the most common we have encountered. In any case of data-overflow we have easy mechanisms to cut the great explosion of filling up the memory with cutting the maximal length of the Q records list. This is a good safety mechanism, we need to remember that data collected later on is usually way more valuable than data collected on the beginning, when the model is not trained almost at all.

Having that in mind we have also created a way to input and output our model into a *.csv* file. Thanks to this practise we can train our model partially, save data to file, and keep training it later on. Also we can compare models trained with different approaches, with various ways of counting the state (distance to the apple) and various training methods.

As a good practice in Machine Learning we also put something called *Epsilon* which is a percentage with which random moves are chosen instead of predicting rewards from our neural network. This enables the model to explore new paths, and prevents from falling into a local maximum. As the model is getting better in predicting the correct moves, the *Epsilon* is decreasing, in our case we decrease it by every thousand games.

3 Results

3.1 Neural network

Choosing the neural network over a basic Q table was an interesting choice for a couple of reasons. Such networks enable us to work with it with a very low input in building the model itself. Neural network is a sort of a standalone model that only needs the programmer to input the variable vector - information from the

environment and also train the model when needed. Apart from that all the calculations and neuron activity is happening inside the network, we do not interact with that.

One of the problems we had is just that - the sole fact that the network is a very standalone entity and we cannot do much about what is happening inside of it once we input the vector of values. We were tweaking and changing both the reward system, and the distances (impulses getting to the Agent from the environment of the game) - what we called previously a state but the results were always almost accidental. After around 30.000 iterations, whatever the gamma or if there was a third layer in the network at all, the Agent was able not to die, but we could not teach him to pick the apples. And the only action that we could undertake is longer training and trying to expose the agent to such situations in which he failed to do what we wanted him to do.

The hermetic structure of a neural network was sometimes hard to work with cause we were only left with working with just the values vector, leaving all the 'inside job' to the closed model of the network.

3.2 Performance overall

We had a lot of changes and enhancements in our project, all based on the results given by the trained snake. We were able to train a snake that wasn't killing itself-it simple was going in circle, however it wasn't eating any apple. On the other hand we also created a model that was eating few apples but it was killing itself instead.

After checking our snake's performance on longer periods of training, which are time-consuming, we have chosen a different way check our snake. To check which aproach is better, for example whether the network with one hidden layer or two will show better results we prepared diffrent cases to train for 30 000 episodes (games) and see which one has outperformed.

Here is the table with the parameters and the results. Results are based 10 games played by each snake with the epsilon set to 0. All of the trained Snakes had troubles with corners, and mostly were dying in there.

gamma	nr. of layers	dropout rate	nr. of epochs	average lifespan	average points
0.4	2	No dropout	2	32.7	0.2
0.2	2	No dropout	2	18.6	0.2
0.6	2	No dropout	2	25.1	0
0.8	2	No dropout	2	20.5	0.3
0.4	3	0.3	6	15.8	0.1
0.4	3	0.5	6	11.2	0.2
0.4	3	No dropout	4	21	0.1
0.2	3	No dropout	4	39.1	0
0.6	3	No dropout	4	17	0
0.8	3	No dropout	4	19.3	0.1