Final Project Report

Stepan Tytarenko

Data - Computer Science, Carolina University

DCS 520 50 : Statistics for Data Science

Professor Nalin Fonseka

February 22, 2023

Q-Learning and epsilon-greedy algorithm

Problem statement

Q-learning - is a type of unsupervised learning, which is also known as reinforcement learning. This is the machine learning technique that utilizes the approaches of trial and error to learn for a specific task, receive a reward for making some progress, and be penalized for mistakes. This is pretty much how children learn about the world. However, a child, when choosing whether to use some previous experience or try something new, usually relies on some consciousness. While this is not possible in Machine learning (yet), the algorithm may rely on some technique. In our case, the epsilon-greedy technique. Every time the algorithm needs to decide if it is better to exploit some previous experience or explore new options, we would use the technique.

At each step of the training phase, the algorithm would choose what to do: exploit/explore. The results of the step (no matter what the choice was) are stored in the so-called Q-table. This table is then used when we exploit, thus reusing some previous experience. The rows of this Q-table represent states that the environment can be in, while columns represent which action was performed in the specific state, while the value is the reward for this action.

This is a basic idea of the Q-learning technique (worth noting that epsilon-greedy is only one of the possible techniques for decision-making, its main drawback is the greediness itself, which may not always lead to global optimality).

Solution proposal

The game with basic level editing. One can add obstacles, enemies, and bonuses on the level by putting those in the JSON file. That way creating and editing test environments is as easy as adding a couple of rows to the JSON file. After the level is built, one should initialize some basic parameters and see the model performance. Based on that, one may consider updating or keeping the parameters. The game itself is just a set of primitives. The objects can either be static or moving. The "player" has to get from the starting point to the finish line without "dying" because of obstacles. The agent can perform 4 types of actions, stay, move left, move right, and jump. It is also possible to move in some direction while in the jump. The gravity rules are applied to the player, meaning without solid ground, it would fall. Double jump is not possible.

The formula for the reward would be used as follows [source]:

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha(reward + \gamma \max_{a} Q(nextstate, allactions))$$

Where α - is the learning rate (how many values are being updated each iteration)

γ - discount factor, how important long-term rewards are, rather than short-term greedy ones.

To go further, we should consider not only the greedy decisions made at each step but also the consequences of such decisions. For that, we need to introduce the concepts of “path reward” and “backpropagation of path reward."

In machine learning in general and in supervised deep learning specifically, the concept of error backpropagation is very common. Each time during training, after the Neural Network predicts the output and the loss is calculated, the process of backpropagation starts. The method calculates the gradient of the error with respect to the networks’ weights. What we try to implement in this project is a similar concept, however, for unsupervised reinforced learning.

For that, suppose we have reached some point of the “end game." This point might be the finishing line, might be due to collision with a killing obstacle, or might be due to falling off the platform. It might also be on the 5th step or on the 255th step. We should consider all of the possible outcomes and, based on that, calculate the backpropagation reward.

First of all, we should definitely penalize the model if it, for any reason, fails to reach the finish line. Naturally would be to punish the whole path that led to the current point. Also, quite naturally would be to punish those attempts that fail fast. It means that something went wrong in the decision-making really early. However, in doing that, we have to be very careful. Since punishing the short path would mean we punish some initial decisions, which itself means, that we might spoil the optimal start. Because the first move that model makes is not necessarily bad or wrong. Let’s say the model failed after 5 steps. The fifth step itself is indeed a bad decision, yet we have no idea if the previous steps were as bad. While punishing them too much could lead to the exclusion from the further process for a long time and potentially could slower the convergence and decrease optimality. A workaround that we apply here is the backpropagation decay. That means that we still penalize the whole path if it leads to loss, yet, we strongly punish the last decision, and the further we move from the last decision, the less penalty we apply to the model. Thus, if the model failed after 200 steps, the last steps before failure would be penalized very hard, yet the closer we get to step 0, the less penalization happens. This approach helps us maintain the right balance between bad paths avoidance and model variance. The parameter to control how much we decrease the penalty with the steps is backpropogate_decay. To avoid decay, we put it as 1.0; to prevent other steps rather than the last one from punishing, we put it as 0. The closer this parameter gets to 1, the earlier steps are penalized. We use the value 0.9. It reasonably punishes the last ~10 steps; however, if the number of steps is relatively big (like 200), it will very lightly punish the earlier steps since if the model lived for so long, they must have been good.

The same is true for the case if we don't want to penalize the whole path but reward it. Since we are not only trying to solve the puzzle (finish the game) but to do it as fast as possible, over-rewarding any solution would lead to model overfit on that solution. It would make it almost impossible to try some other ways to find a faster option. Thus we should also apply the same technique of rewarding the path, rewarding the earlier steps less.

The Code of the function conducting the backpropagation is given below.

```
def _backpropogate_path_reward(self, path, reward):
    decay = self.backpropogate_decay
    for state, action in reversed(path):
        self.q_table[state, action] += reward * (log_scale(decay, max_val=1) + 1) / 2
        decay *= self.backpropogate_decay
```

The incoming parameter reward here is identified through a different function. It calculates the initial reward or penalty that the model deserves for this iteration based on the results we obtained. If the model has lost very fast, we would like to avoid that, and we would penalize that a lot. If we failed, but after a decent amount of progress, then this penalty would be less hurtful. Finally, if the model succeeded much faster than ever before, it would receive a big reward, while succeeding slower than before would result in some smaller rewards. The function to calculate the reward is provided below.

```
def _calc_path_reward(self, min_steps, steps, result):
    """

    :param min_steps: min_steps to win previously
    :param steps: steps to reach the result of this iteration
    :param result: the result of the iteration
    :return: reward|penalty
    """
    reward = 5
    if result == GameStatus.LOST:
        reward *= -1
        reward = reward + self.alpha * reward * np.exp(self.fail_path_penalty / steps)
    elif result == GameStatus.WIN:
        reward = 10 * reward + self.alpha * reward * np.exp(steps / min_steps)
```

```
    elif result == GameStatus.PLAYING:
        reward *= -1
    return self.gamma * reward
```

All the previous descriptions concentrated on the path reward. An important and innovative concept applied to this model. But besides, there is a step reward. Step reward is the reward model gets for each individual step. Step reward would be a big positive if the last move was in the right direction (here "direction" is not just left or right, we consider how the distance between the player and the finish has changed between current and previous steps, decreasing would lead to higher reward). We need that concept since moving in the wrong direction or staying in the same spot does not allow the model to win. To measure how well this type of compensation works, we calculate how often the models choose various actions. These are relative numbers since they are only applicable to the specific setup and the specific level design. The finish here is to the right of the player, and the level design requires a lot of jumping to pass the level. Below is the table of stats of actions used for this type of level after 1000 thousand steps.

STAY	JUMP	MOVE_LEFT	MOVE_RIGHT
17%	14%	9%	60%

For this same level, the average steps per 100 iterations are 161.59.

Empirically it was discovered that, on average, the Manhattan distance for per-step reward calculation is the best option. This is probably because it evenly considers the vertical and horizontal distance, which is more natural, compared to the euclidean distance considering only the shortest distance.

The maximum number of steps is set to 1000. If the model is stuck for more than that, then the attempt is considered as failed automatically. This would be punished by the previously introduced approach. The convergence is reached after around 700 steps for one specific level.

Conclusion

During the work on this project, the fully-functioning reinforcement environment system was built. The model uses some of the very common approaches based on Q-learning, as well as some of the novel ideas, like path reward and reward backpropagation. We have proven the real value of the proposed techniques and have proven the capability of such an approach to solving optimization tasks in the field of unsupervised learning. Besides, the whole model logic behind the

model training is built upon the epsilon-greedy technique, which was introduced during the course, and has proven its real-life value.

The source code of the described system will be publicly available on GitHub. The level editing feature allows the researchers to train models and test hypotheses with various parameters with simple JSON file editing. One of the further developments of the project is to make enemies as the new system agents that can interact with the player and will simultaneously learn how to play and beat the player while the player will learn how to solve the puzzle and avoid enemies.