BitFlipping with Deep Q-Network

Haojun Chen
Statistics
Beijing Jiaotong University
20271265@bjtu.edu.cn

Abstract

The purpose of this experiment report is to investigate and evaluate the application of Deep Q-Network(DQN) in the task of BitFlipping problem. BitFlipping problem is a classical problem in Computer Science. The conventional approach typically involves enumerating over a space of size 2^n, leading to a high time complexity. However, Reinforcement Learning methods can explore this vast search space by using rewards from the constructed environment to guide the agent towards optimizing for the highest expected rewards. This approach helps avoid unnecessary enumerations. In this study, I attempted to address this issue using the DQN algorithm from reinforcement learning. Besides, I designed DQN_with_GOAL to enhance the generalization of goals, leveraged hindsight experience replay tricks to mitigate the sparse reward issue and give an analysis of reward shaping. I release my code at https://github.com/chjchjchjchjchj/BitFlip.

1 Introduction

In this report, I explore how Deep Q-Network (DQN), a powerful machine learning technique, can help solve the BitFlipping problem in Computer Science. BitFlipping seems like a straightforward task—it involves reversing binary sequences. However, the challenge lies in the fact that it can involve a huge number of possibilities, making traditional methods time-consuming.

Recently, we've seen exciting developments in reinforcement learning, which teaches an agent to make decisions by interacting with its environment and aiming to maximize rewards. What's cool about it is that it's really good at handling big search spaces, avoiding the need to check every possibility.

Our goal in this study is to test if DQN, a specific reinforcement learning algorithm, can make BitFlipping easier and faster. We believe that by using DQN, creating the right environment, and using rewards, we can efficiently solve the BitFlipping problem. It might be a faster approach compared to the old ways.

In this report, firstly, I'll show how I build the environment BitFlipping ti solve this task. Then, I'll introduce the architecture of DQN, DQN_with_GOAL, DQN_with_HER I implemented. After that, I'll show the experiments part, which includes exploring the success rate with increasing the State space. Then I'll show the exploration process on reward shaping aspect. Finally, I'll show the challenges I faced during I conducted the experiment and show the conclusion.

2 Environment

To address the problem of bit flipping using reinforcement learning, it is necessary to establish a reinforcement learning environment. The Settings are as follows:

State space: $\mathcal{S} = \{0, 1\}^n$, $n \in \{1, ..., 50\}$

Action space: $\mathcal{A} = \{0, 1, ..., n-1\}$ for some integer n in which executing the i-th action flips the i-th bit of the state.

Reward: The reward is 0 if the final sequence generated is equal to the target sequence, and is -1 otherwise.

For each episode, a target is generated randomly.

3 Methods

3.1 Deep Q Network

3.1.1 Architecture

The architecture of DQN I implemented consists of several key components:

Neural Network: the DQN consist of 2 networks (namely Q-network and target Q-network), each of them consist of 2 hidden layers with 256 hidden units and 128 hidden units. Hidden layers use ReLU activation function. The input dimension is the length of the initial bits, and output dimension is 1.

Experience Replay Buffer: The experience replay buffer is set to a size of 10,000. This buffer stores past experiences (state, action, reward, next state) encountered during training. It enables efficient sampling of experiences to break temporal correlation in the data and stabilize the training process.

Epsilon-Greedy Policy: The epsilon-greedy exploration strategy starts with an initial epsilon value of 0.9. Epsilon is then gradually reduced by 1e-5 with each training iteration until it reaches a minimum value of 0.1. This strategy balances exploration and exploitation. Initially, the agent explores the environment extensively (with a higher probability of selecting random actions), and over time, it shifts toward more exploitation (choosing actions with the highest estimated Q-values).

Loss Function: The loss function used is Mean Squared Error (MSE). This loss measures the difference between the predicted Q-values and the target Q-values computed using the Bellman equation.

Batch Size: During training, mini-batches of experiences are sampled from the replay buffer, whose size is 128. These experiences are then used to update the Q-network's weights through gradient descent, minimizing the MSE loss.

3.1.2 Pseudocode of DQN

Algorithm 1 Deep Q-Network

Initialize network $Q_\omega(s, a)$ with random parameters ω
Initialize target network with the same parameters: $\omega^- \leftarrow \omega$
Initialize experience replay buffer R
for episode $e = 1$ to E **do**
    Get initial environment state s_1
    for time step $t = 1$ to $nbits$ **do**
        Choose action a_t based on the current network $Q_\omega(s, a)$ using an ϵ-greedy strategy
        Execute action a_t, receive reward r_t, and transition to the new state s_{t+1}
        Store (s_t, a_t, r_t, s_{t+1}) in the replay buffer R
        if R contains enough data **then**
            Sample N data points $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1,...,N}$ from R
            for each data point **do**
                Compute target $y_i = r_i + \gamma \max_a Q_{\omega^-}(s_{i+1}, a)$ using the target network
            Update the current network by minimizing the target loss $L = \frac{1}{N}\sum_i (y_i - Q_\omega(s_i, a_i))^2$
            Update the target network

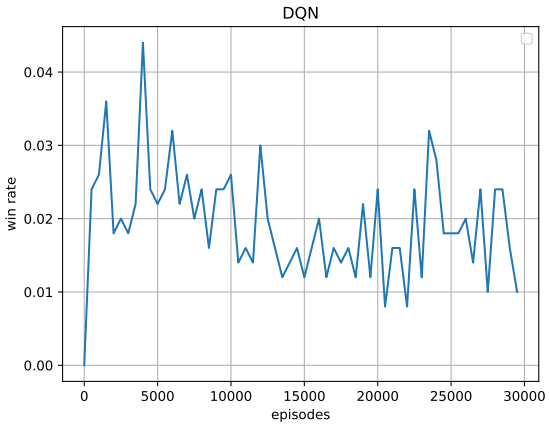

Figure 1: dqn's win rate in 8 bits

3.2 Deep Q Network with goal

Since the objective is to transition from the initial state to the goal state, and each episode's goal state is randomly generated, it is evident from Figure 1 that merely using DQN cannot yield effective strategies. This is because the randomly generated goal state results in learned policies that are also random, or in some cases, no effective strategy is learned at all. Therefore, I have introduced a condition to the agent's action selection, which incorporates the goal. This modification allows the agent's policy network to receive not only the current state as input but also the episode's goal state. The goal is to enable the agent to learn strategies for multiple goals.

Compared to the architecture of a standard DQN, the Neural Network for DQN with goals has twice the input dimension. This is because it needs to simultaneously process both the state and the goal as inputs.

3.3 Deep Q Network with Hindsight Experience Replay(HER)

The motivation behind introducing Hindsight Experience Replay (HER)[1] to DQN is driven by the fact that the BitFlipping environment has a reward function consisting of only two integers, and positive feedback is only obtained when the agent reaches the goal state. As observed from Figure 2, both the DQN and DQN with goal models struggle to successfully reach the goal state, especially when n is large. This indicates a challenge with sparse rewards. Therefore, the incorporation of Hindsight Experience Replay serves to alleviate the sparse reward problem.

4 Experiments

At first I set the experimental setting as Table 2 and set the hyper-parameters of DQN as Table 3 to find the hyper-parameter that make agent reach goal in an highest frequency. The parameters highlighted in bold in Table 3 represent the ones that, in the case of DQN with nbits=8, achieved the highest frequency of reaching the goal state. So the following experiments I conducted is based on the settings above.

Let's note **win rate** as the frequency that the agent successfully reach the goal

4.1 Environment Configuration

The experiment is conducted under the environment configuration that shows in Table 1.

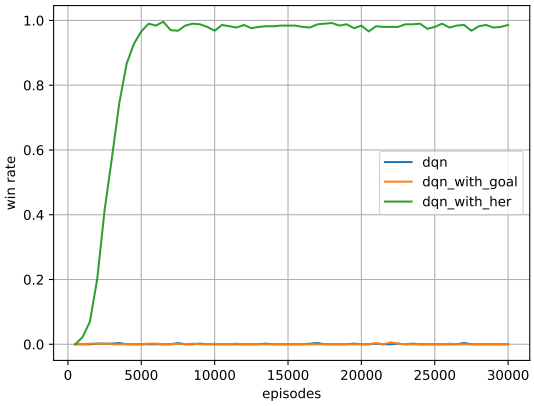

Figure 2: training process of 3 models in 14 bits

Table 1: Environment Configuration

Operating System	Ubuntu 20.04
CPU	64 vCPU Intel(R) Xeon(R) Platinum 8350C CPU @ 2.60GHz
GPU	NVIDIA GeForce RTX 3090 24GB RAM
Framework	PyTorch 2.0.1 CUDA 11.8

4.2 nbits

In this section, I varied the length of the state in the environment (nbits) to observe how the win rate of DQN changed with varying state lengths. This exploration aimed to determine whether DQN could still reach the goal state when nbits was set to a large value. Figure 3, it's evident that DQN struggles to learn effective strategies, which validates my earlier hypothesis. When we introduced the goal as an input to DQN's strategy, we observed that in cases where the state length was less than 10, the agent could successfully reach the randomly generated specified goal in each episode. This demonstrates the necessity of incorporating goal features into the agent's learning process, allowing it to generalize across different goals.

However, due to the sparse reward nature of the BitFlipping environment, when the state length exceeds 12, the agent finds it increasingly difficult to reach the goal. I suspect this is because the agent does not receive sufficient positive feedback signals; in most cases, it only receives a reward of -1. Consequently, as the state length increases, the agent encounters fewer instances of positive feedback, leading to a trial-and-error approach in the majority of cases.

4.2.1 Why does DQN with HER appear to be more effective than DQN and DQN with goal?

I implemented DQN with HER, whose pseudocode is showed in Pseudocode 2. In pseudocode, the portions highlighted in red represent the differences between DQN with HER and DQN. DQN with HER accomplishes this by establishing an additional hindsight experience replay buffer, denoted as R_2, to store trajectories where the agent did not reach its goal. Subsequently, it sets a new goal as the last state in the trajectory, effectively transforming a failed trajectory into a successful one. The algorithm then calculates the rewards within the trajectory, thereby increasing the frequency with which the agent receives positive feedback during its interactions with the environment. This mechanism serves to alleviate the issue of sparse rewards. From my perspective, the described process can be seen as a form of data augmentation.

4.3 Reward Shaping

In this section, I will present my exploration of improving the algorithm's performance by altering the reward function.

Algorithm 2 Deep Q-Network with Hindsight Experience Replay

Initialize DQN, replay buffer of DQN R_1, hindsight experience replay buffer R_2
for episode $e = 1$ to E **do**
    Sample a goal g and an initial state s_1
    for time step $t = 1$ to $nbits$ **do**
        Choose action a_t based on the DQN's policy:

$$a_t \leftarrow \pi(s_t, g)$$

        Execute action a_t, receive reward r_t, and transition to the new state s_{t+1}
        Store $(s_t, a_t, r_t, s_{t+1}, g)$ in the replay buffer R_1
        if R_1 contains enough data **then**
            Sample N data points $\{(s_i, a_i, r_i, s_{i+1}, g)\}_{i=1,\ldots,N}$ from R_1
            Train DQN's neural networks
    if agent fail to reach the goal state **then**
        new goal $g' \leftarrow s_{nbits}$
        for time step $t = 1$ to $nbits$ **do**
            $r_t \leftarrow R(s_t, a_t, g')$
            Store $(_t, a_t, r_t, s_{t+1}, g')$ in the hindsight experience replay buffer R_2
            if R_2 contains enough data **then**
                Sample N data points $\{(s_i, a_i, r_i, s_{i+1}, g')\}_{i=1,\ldots,N}$ from R_2
                Train DQN's neural networks

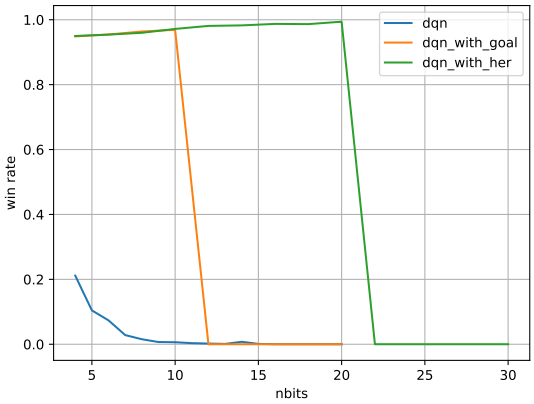

Figure 3: dqn's win rate

Table 2: Experiment Settings

Parameter	Value
training episodes	30000
batch size	32
first hidden dim	256
second hidden dim	128
optimizer	Adam
learning rate	1e-3
buffer size	10000
loss function	MSE loss
minimal epsilon	0.1

Table 3: Hyperparameters

Parameter	Values
initial epsilon	**0.9**, 0.5, 0.1
delta epsilon	1e-2, 1e-3, 1e-4, **1e-5**
minimal size	**64**, 128

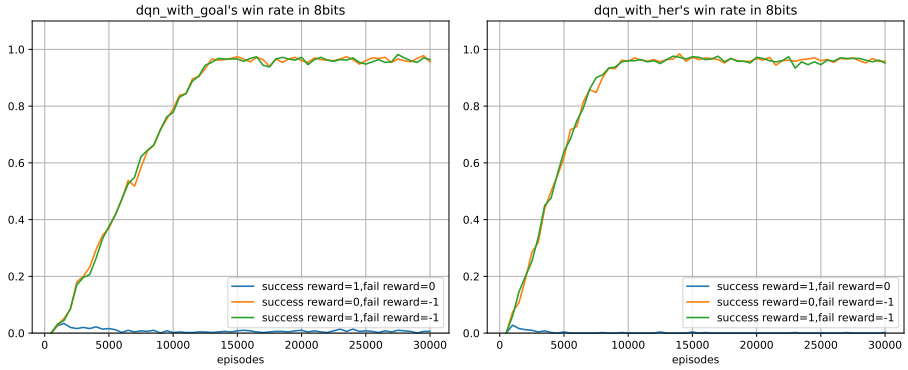

Figure 4: dqn_with_goal and dqn_with_her 's win rate with various reward settings

4.3.1 Binary Reward Settings

Firstly, I configured the reward function in the BitFlipping environment such that the agent receives a reward of 1 when it reaches the goal state, and 0 otherwise. However, in multiple scenarios, the model exhibited a low win rate and struggled to learn effective strategies. Subsequently, I referred to the reward setting in reference [1], where the agent receives a reward of 0 upon reaching the goal state and -1 otherwise. Under this condition, the model's win rate improved significantly, and it converged quickly, as depicted in Figure 4. Based on these experiments, I surmise that providing the agent with negative rewards for taking incorrect actions is crucial. Therefore, I modified the reward function to give the agent a reward of 1 when it reaches the goal state and -1 otherwise. Under this scenario, the model's training performance is similar to that observed when using the reward setting of (success reward=0, fail reward=1).

4.3.2 Step by Step vs Euclidean Distance

In this section, I initially employ a "Step by Step" reward design approach to enable the agent to receive rewards on a per-bit basis. Specifically, I evaluate whether the bit flipped by the agent at a given position matches the bit in the same position in the goal state. If they match, the agent receives a reward of 0; otherwise, it receives a reward of -1. The motivation behind this reward function design is as follows: when relying solely on rewards based on the final state compared to the goal state, positive feedback signals are infrequent. However, by comparing the agent's progress on each bit of the state to the corresponding bit in the goal state, the agent can gain insight into whether the flipping of the current bit brings it closer to the goal state. This approach helps guide the agent in updating its state effectively.

If we place the state in Euclidean space, the "Step by Step" reward design approach is akin to guiding the agent along a particular dimension in Euclidean space to reach the goal state. Based on the triangle inequality, I conjecture that directly informing the agent of the Euclidean distance between its current state and the goal state could enable the agent to reach the goal state more quickly. Therefore, in this

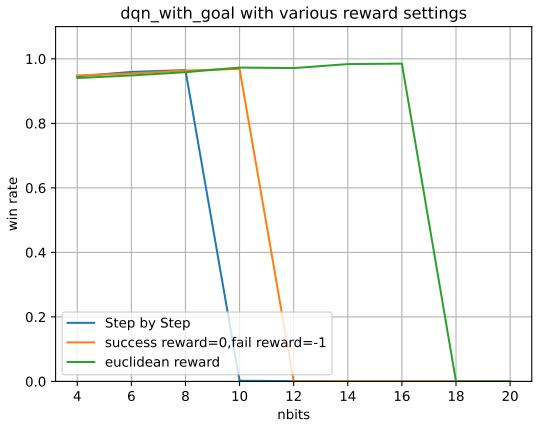

Figure 5: dqn_with_goal and dqn_with_her 's win rate with various reward settings

context, I propose changing the reward to be the negative of the Euclidean distance between the agent's current state and the goal state. This adjustment encourages the agent to update its state in a direction that minimizes the penalty, ultimately leading it closer to the goal state.

From Figure 5, we observe that the reward constructed using the negative Euclidean distance leads to higher win rates than the "Step by Step" design for states of equal length. However, the "Step by Step" design did not improve the win rate compared to the initial environment reward setting; instead, it decreased.

One possible explanation for this observation is that, even if the agent fails to reach the goal state, it can receive positive feedback during the trajectory if certain bits in the state match the corresponding bits in the goal state. This means that, when viewed from the perspective of the entire trajectory, the agent can still accumulate high rewards despite failure, which does not align with the desired behavior. In contrast, the Euclidean distance-based reward design considers the similarity between all bits of the state and the goal state, providing a more comprehensive assessment that significantly improves the win rate compared to the original model.

5 Challenges

1. After setting up the dqn_with_goal model, I noticed that regardless of how simple I made the environment (with a small nbits), it wasn't learning effectively. While debugging later, I discovered that both the state and next state in the replay buffer were the same. I realized that the reason for this occurrence was that I hadn't handled the issue of deep copying properly when creating the BitFlipping environment.

2. Initially, I set up the environment with a reward of 1 when the agent reached the goal state and a reward of 0 otherwise. I noticed that regardless of the model I used, it was mostly unsuccessful. In fact, the win rate curve was even showing a declining trend as the number of episodes increased. Later on, I changed the reward structure to give a reward of 0 upon reaching the goal state and -1 when not reaching it. This simple adjustment resolved the previously mentioned issues, and all models started working effectively right away.

6 Conclusion

1. DQN algorithm is ineffective due to random goal state generation.

2. DQN_with_GOAL can enhance the goal generalization.

3. DQN_with_HER can significantly mitigating the sparse reward issue.

4. Negative rewards for agent errors proved essential, which can prevent agent to choose bad action during the exploration.

5. Setting the negative of Euclidean distance as the reward can leverage the performance.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.