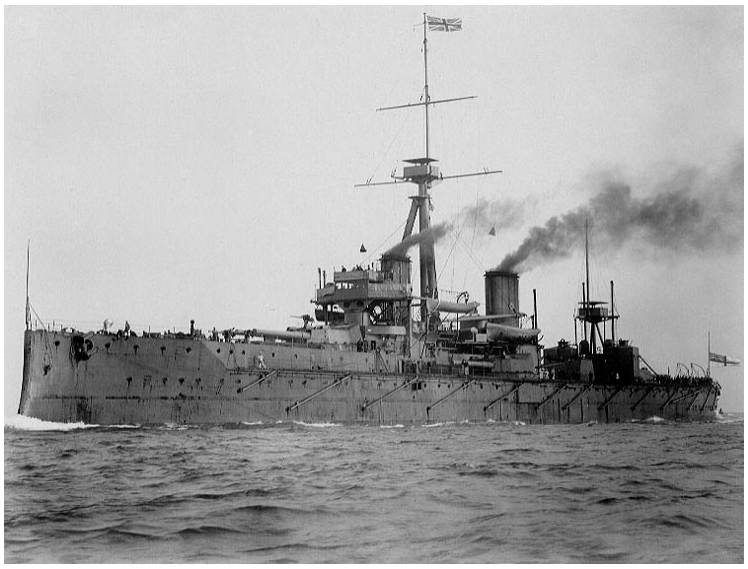
Lab Assignment #4: Classical vs Modern Reinforcement Learning



1 Introduction

In this lab we will attempt to compare the efficacy of state engineering plus classical Q-learning with DQN. We will use these algorithms to build agents capable of playing the game of Battleship. Battleship is partially observable; actions in the current turn reveal information about the hidden state in available for the agent to use in future turns.

2 Set up and learn about the environment

In the original board game, two players each take control of a fleet of five ships of different sizes, arranged in secret on a 10×10 grid. They then take turns selecting a grid cell on their opponent's grid on which to fire. If the cell has a part of one of their opponent's ships in it, they learn that they have a hit. Otherwise they learn that they have a miss. Once all locations on a ship are hit it is sunk and removed from the board. Play continues in this way until one player has had all of their ships sunk.

Our version of Battleship will simplify the classic version in several ways:

- We will work with a one-player version of the game in which there is a hidden board and only the agent gets to select targets;

- The hidden fleet will have three ships, one two grid cells long, one three grid cells long, and one four grid cells long;
- The grid will be 6×6 cells;

Although coding up a simulator is an important part of reinforcement learning we have already had opportunities to practice this; I will provide code that implements the Battleship game so that the focus can be on training agents to play.

1. Download the Battleship_Simulator.ipynb file. This contains a class that sets up and runs the simplified Battleship game.
2. Set up the environment class by specifying the appropriate config dictionary:

```
config = {'n':6,'ships':[2,3,4]}
```

3. Simulate 1000 games with randomly generated actions.

3 Training Agents to play Battleship with Classical Reinforcement Learning

In this part of the lab we will construct multiple agents capable of managing our Battleship environment and compare/contrast their performance.

- **Decide how you will represent the state.** Remember – this will be the only information that the agent has available to make decisions. Battleship is a partially observable Markov Decision Process, since the disposition of the enemy fleet is not known by the agent. The simplest way to represent the state would be to record, for each grid cell, whether it is a hit, a miss, or has not yet been targeted. This version of the state is how self.state in the environment is currently set up. Representing the state in this way is unlikely to work – you will need to develop an alternative conceptualization for the state for your agent to use. Think creatively! What information does your agent need to manage their choices effectively? And then, given how you represent the state, you will need to align the actions of the agent to the appropriate level of granularity.
- **Choose a reward function.** Battleship is, at its heart, a race. The goal of the agent is to win the game quickly and you need a reward function, $r(s', a, s)$, to incentivize the agent to do this. There are many possible reward functions. Perhaps the simplest would be to record the total number of hits or the total number of hits less the total number of misses.

- **Implement classical RL algorithms.** Build an agent class that includes at a minimum the Q-values, the policy, and the learning update rule for the Q-learning algorithm below.
- **Train agents.** Train two agents, a naive agent trained for 500 episodes and an experienced agent trained for an "appropriate" and much larger number of episodes. You will need to choose the probability of exploring ε, the discount factor γ, and the learning rate, α.

Algorithm 1 TD Control via Q-learning with ε-greedy policies

Input: $\varepsilon, \gamma, \alpha \in (0,1)$, and $N \in \mathbb{N}_+$ ▷ Specify policy and hyperparameters

Init: $Q_\pi(s,a) \in \mathbb{R}$ for all s, a ▷ Initialize Q-values

for $n = 1, 2, 3, \ldots, N$ **do** ▷ Loop over episodes

  Sample $s \in S$ ▷ Choose initial state

  for $t = 1, 2, 3, \ldots, T$ **do** ▷ Loop over turns

    Sample a according to: ▷ Choose new action

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|A|} & \text{if } a = \operatorname{argmax}\{Q_\pi(s,a)\} \\ \frac{\varepsilon}{|A|} & \text{else.} \end{cases}$$

    Implement a to generate r and s' ▷ Get new reward/state

    Update the state-action value for s, a according to: ▷ Update Q-values

$$Q_\pi(s,a) \leftarrow Q_\pi(s,a) + \alpha \left[r + \gamma \max_{a' \in A} \{Q_\pi(s',a')\} - Q_\pi(s,a) \right];$$

    Update $s \leftarrow s'$ ▷ Update current state

  end for

end for

4 Training Agents to play Battleship with DQN

- **Use the current board as the state.**
- **Choose a reward function.** You may use the same reward function as in section 3 if you like, though there may be more information to work with this time.
- **Implement DQN.** Build an agent class that includes at a minimum the Q-values, the policy, and the learning update rule for the DQN algorithm below.
- **Train agents.** Train an agent using DQN for an "appropriate" number of episodes. You will need to choose the probability of exploring ε, the discount factor γ, the learning rate, α, the batch size, B, the size of the replay buffer, M, and the frequency with which the target approximator is updated, C.

Algorithm 2 Deep Q-learning (DQN)

Input: $\varepsilon, \gamma, \alpha \in (0,1)$, and $B, N, M, C \in \mathbb{N}_+$ ▷ Specify policy and hyperparameters

Init: $\mathbf{w} \in \mathbb{R}^d$, replay buffer D with fixed memory M ▷ Initialize approximator & agent memory

Set $\mathbf{w}' = \mathbf{w}$ and $\pi(s)$ as ε-greedy w/r/t $Q_\pi(s, \cdot|\mathbf{w})$; ▷ Initialize target approximator & policy

for episode $1, 2, 3, \ldots, N$ **do** ▷ Loop over episodes

  Sample $s \in S$ ▷ Choose initial state

  for $t = 1, 2, 3, \ldots$ **do** ▷ Loop over samples

    Sample a according to $\pi(s)$, observe r and s'

    Add (s, a, r, s') to D and if $|D| > M$ drop the oldest ▷ Update agent memory

    if $|D| \geq B$ **then**

      Sample a subset of tuples $(s_j, a_j, r_j, s'_j)_{j=1}^B$ from D uniformly; ▷ Choose batch

      Compute: ▷ Generate target values

$$y_j \leftarrow r_j + \begin{cases} \gamma \max_a \left\{ Q_\pi(s'_j, a_j|\mathbf{w}') \right\} & \text{if } s' \text{ is not terminal} \\ 0 & \text{else;} \end{cases}$$

      Update: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \left(\frac{1}{N} \sum_{j=1}^N \left(Q(s_j, a_j|\mathbf{w}) - y_j \right)^2 \right)$; ▷ Take a single gradient step

      Every C steps $\mathbf{w}' \leftarrow \mathbf{w}$ ▷ Update the target approximator

    end if

    Update $s \leftarrow s'$

  end for

end for

5 Compete the agents against one another

In the final part of the lab we will compete the agents we have trained against one another to see if they have learned similar policies or if one training algorithm is significantly more effective in learning to play simple Battleship. We will keep this simple – take each trained agent, have them play 100 episodes (no updating Q or π), and record for each the number of turns it takes to win. Then to decide which agent in a matchup would have won a given game, choose a random episode for each agent and give the victory to the agent that took the fewest turns to finish that game. Do this 100 times for each of the following matchups:

- Naive vs Experienced Q-learning;
- Naive Q-learning vs DQN;
- Experienced Q-learning vs DQN.

Lab Assignment Report

Write up your analysis in a separate report of no more than 2 pages. Your report should answer the following questions:

1. Describe in detail how the Battleship works and what each method within it does.

2. Provide and interpret the plot of the distribution over game length that you created in 2.3.

3. Describe how you represented the state for classical Q-learning. How many total states are there given your representation of the state?

4. Describe the reward function you chose. How do you think it impacted the learning of the agent?

5. What does the learned policy of the agent look like? Specifically, create and provide a visualization of what the agent chooses to do given the board state.

6. What effect (if any) do γ and ε have on the performance of these agents? How would you assess this?

7. Summarize the win rates of the different agent matchups. Have you trained qualitatively different agents via these different algorithms? Do they do substantially better than randomly chosen actions? Do they finish the game in less than 36 turns?

6 Submission Instructions

- Write up the answers to the questions in a short word document; aim for no more than 2 pages of text and include all graphics generated. Add footnotes identifying which sentence addresses which questions. Write in complete sentences organized into paragraphs – your goal is to explain what you've done and what you've learned to your audience (me!). Include the appropriate plots you've generated as mentioned above. Convert this to pdf and submit it. Submit your .ipynb file and an .html of it as well.

- The grading rubric for this assignment will be available in Canvas.

- NO OTHER SUBMISSION TYPES WILL BE ACCEPTED.

- You are welcome to use generative AI as you code up your solution. If you do this you must include all prompts used as an appendix in your written report or provide a shareable link to them. Any uncited use of generative AI (i.e. prompts not provided) will be considered plagiarism. You may not use generative AI to write your report.