Foundations of Reinforcement Learning Project Report: Application of Deep Reinforcement Learning on *Bulls and Cows*

Yu-Hsin Lee

November 2023

1 Introduction

Bulls and Cows is the inspiration behind the social media sensation *Wordle* and the board game *Mastermind*. It is a two (or more)-player game in which opponents try to guess each other's codes through trial and error. It has been shown that the average minimal game length is 5.21 turns and 4.34 turns for *Bulls and Cows* and *Mastermind* respectively [SS]. In this project, I simulated a single-player scenario and trained deep Reinforcement Learning (RL) agents to solve the correct code in the least number of steps possible.

2 The Game

The rules of the single-player version of *Bulls and Cows* are as follows:

1. A four-digit sequence with each digit distinct, e.g. 1234, is randomly generated.
2. For every turn, the player guesses a four-digit sequence with each digit distinct, e.g. 9824.
3. The player is then provided with the number of matches: *Bulls* for correct digits in the right position, and *Cows* for correct digits in the wrong position. For simplicity, we will use A and B in place of *Bulls* and *Cows* respectively - inspired by the games' alternative name *1A2B*.
4. This repeats until the player guesses the correct four-digit sequence.

Example Episode:
Correct Number: 1234
Guess 1: 9824; 1A1B
Guess 2: 4231; 2A2B
Guess 3: 1234; 4A0B END

All models are set up in a custom *OpenAI* Gym environment [1].

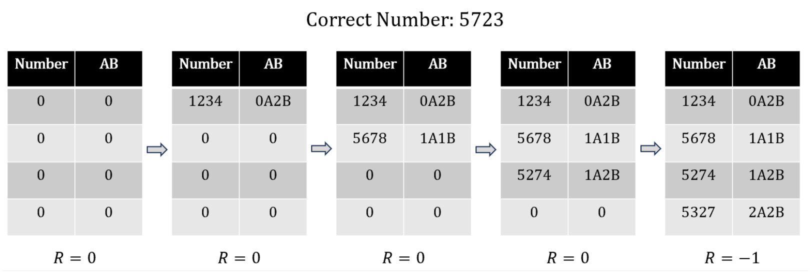
3 Model V1

In all models, the maximum number of guesses allowed is restricted for simplicity. Let this number be m. A 4 digit sequence x is first generated with each digit distinct.

State Space, S_k
S_k is an m by 2 grid: the first column represents the 4-digit sequence guess and the second column represents the number of 'A' and 'B'. S_0 starts off as a grid with every element denoted by 0 (empty guesses). At each step k (or the kth guess), the kth row is filled.

To estimate the state space complexity, the 4-digit sequence has $10 \times 9 \times 8 \times 7 = 5040$ possibilities, while the number of A and B has 15 possibilities (since they must sum up to at most 4). Hence the total complexity $|S|$ is $(5040 \times 15)^m$.

[1] full code available at `https://github.com/leeyuhsin00/BullsandCows`


Correct Number: 5723

Number	AB
0	0
0	0
0	0
0	0

$R = 0$

Number	AB
1234	0A2B
0	0
0	0
0	0

$R = 0$

Number	AB
1234	0A2B
5678	1A1B
0	0
0	0

$R = 0$

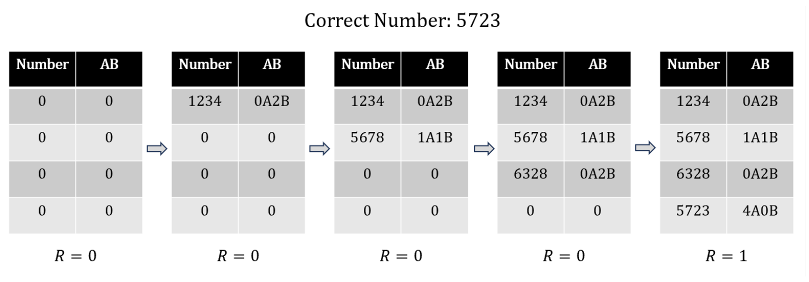
Number	AB
1234	0A2B
5678	1A1B
5274	1A2B
0	0

$R = 0$

Number	AB
1234	0A2B
5678	1A1B
5274	1A2B
5327	2A2B

$R = -1$

Figure 1: Example episode, $m = 4$ with -1 reward (V1)


Correct Number: 5723

Number	AB
0	0
0	0
0	0
0	0

$R = 0$

Number	AB
1234	0A2B
0	0
0	0
0	0

$R = 0$

Number	AB
1234	0A2B
5678	1A1B
0	0
0	0

$R = 0$

Number	AB
1234	0A2B
5678	1A1B
6328	0A2B
0	0

$R = 0$

Number	AB
1234	0A2B
5678	1A1B
6328	0A2B
5723	4A0B

$R = 1$

Figure 2: Example episode, $m = 4$, with +1 reward (V1)

Action Space, A_k
Guess a 4 digit sequence, with each digit distinct. The total possible number of guesses is $|A| = 5040$.

Reward, R_k
$R_k = 1$ if the kth row is (x, 4A0B); this is a terminal state. $R_m = -1$ if the mth row is filled and is not (x, 4A0B), this is a terminal state. Otherwise, $R_k = 0$.

However, the state and action space is computationally too large for my device, so I trained Deep RL agents on the following cases:

1. Numbers: [12,21]; AB: [0A2B, 2A0B]; $m = 3$
   $|S| = [(2 + 1) \times (2 + 1)]^3 = 729$. (+1 to account for empty space)
2. Numbers: [12,13,21,23,31,32]; AB: [0A1B, 0A2B, 1A0B, 2A0B]; $m = 3$
   $|S| = [(6 + 1) \times (4 + 1)]^3 = 42875$.

As shown, as the action space (numbers allowed) increases , the state space increases exponentially. To model the complete version of the game, taking maximum number of guesses as $m = 8$, then $|S| = 5040^8 \approx 10^{68}$. Because of the massive size of the state space, deep reinforcement learning comes into play.

4 Deep Reinforcement Learning

As its name suggests, Deep RL combines both deep learning - a sub-field of machine learning that uses a neural network to transform a set of inputs into outputs - and reinforcement learning - another area of machine learning that aims to learn the optimal actions that lead to maximal reward.

Deep learning enables RL to scale to decision-making problems that were previously intractable, i.e., settings with high-dimensional state and action spaces [ADBB17]. Some of the milestones of deep RL include the development of algorithms that can play a range of Atari 2600 games at a superhuman level (Deep Q-Networks), or defeat a human World Champion in Go (AlphaGo).

RL algorithms can be divided into model-based and model-free algorithms. In model-based algorithms, the agent has access to a model of the environment - a function that predicts state transitions and rewards. The main benefit of having a model is that it allows the agent to plan ahead, and

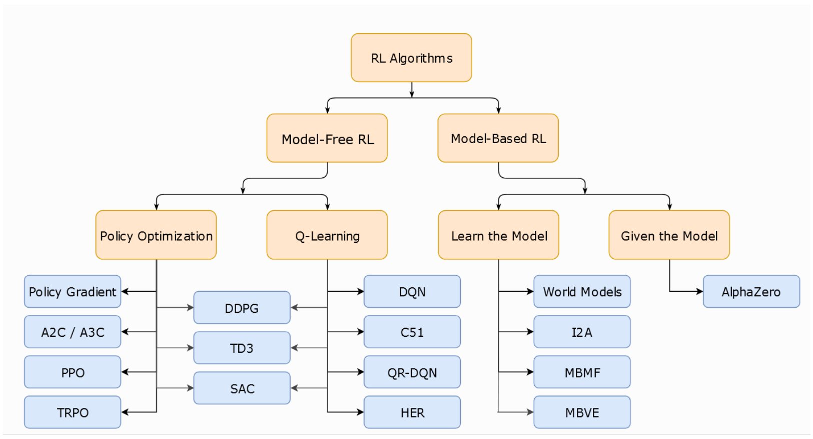


Figure 3: A Taxonomy of RL Algorithms (from OpenAI)

the main downside is that a ground-truth model of the environment is usually not available to the agent. A well-know model-based deep RL algorithm is *AlphaZero*. Model-free algorithms do not have these models to learn from. [Ope]

Since it is difficult to define a model for the environment in *Bulls and Cows*, particularly in the prediction of rewards - since the correct 4-digit sequence is randomly generated - model-free algorithms were used in this project.

There are two main approaches to model-free algorithms: policy optimization and Q-learning. Policy optimization methods aim to directly find policies by means of gradient-free or gradient-based methods. This optimization is almost always performed on-policy, which means that each update only uses data collected while acting according to the most recent version of the policy. Some of such methods are Actor-Critic methods like (Asynchronous) Advantage Actor-Critic (A2C/A3C), and Proximal Policy Optimization (PPO).

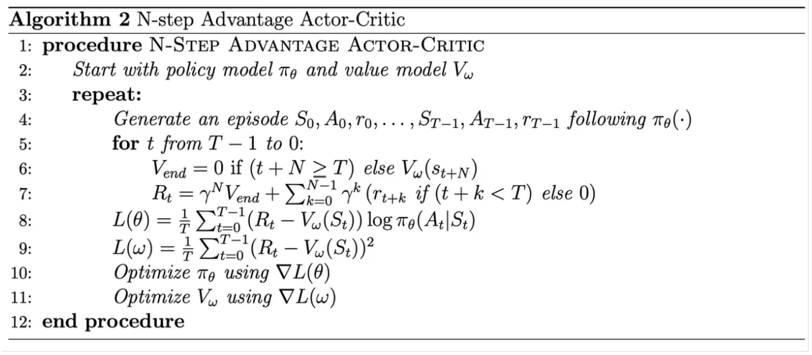
Q-learning methods learn an approximator $Q(s, a)$ for the optimal action-value function $Q^*(s, a)$, using an objective function based on the Bellman equation. This optimization is almost always performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. A notable such method is Deep Q-Networks (DQN) [Ope].

In this project, two policy optimization methods, A2C and PPO, and a Q-learning method, DQN, are used.

4.1 A2C/A3C

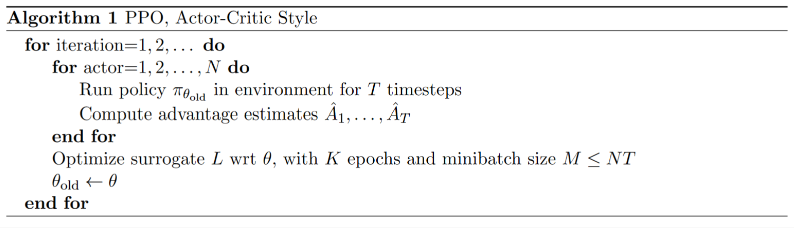
Asynchronous Advantage Actor-Critic (A3C) employs multiple actor-learners with different exploration policies running in parallel to maximize exploration diversity. This reduces training time and uses on-policy RL methods to stabilize learning. It incorporates this concept into Actor-Critic methods, where the 'Critic' estimates the value function, while the 'Actor' (e.g. neural network) updates the policy distribution in response to the 'Critic'. Advantage Actor-Critic specifically uses the estimates of the advantage function $A(s, a) = Q(s, a) - V(s)$ - where $Q(s, a)$ is the state-action value function and $V(s)$ is the state value function - for policy gradient updates. The advantage function can significantly reduce variance of policy gradient. The policy and value functions are updated after every t_{max} actions or when a terminal state is reached. This update can be seen as $\nabla_{\theta'} \log \pi(a_t|s_t; \theta')A(s_t, a_t; \theta, \theta_v)$ where $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function given by $\Sigma_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$. [MBM+16]

Advantage Actor-Critic (A2C) is a synchronous, deterministic implementation that waits for each actor to finish its segment of experience before performing an update, averaging over all of the actors. [WMLS]



```
Algorithm 2 N-step Advantage Actor-Critic
1: procedure N-STEP ADVANTAGE ACTOR-CRITIC
2:     Start with policy model $\pi_\theta$ and value model $V_\omega$
3:     repeat:
4:         Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ following $\pi_\theta(\cdot)$
5:         for $t$ from $T-1$ to 0:
6:             $V_{end} = 0$ if $(t+N \geq T)$ else $V_\omega(s_{t+N})$
7:             $R_t = \gamma^N V_{end} + \sum_{k=0}^{N-1} \gamma^k (r_{t+k}$ if $(t+k < T)$ else $0)$
8:         $L(\theta) = \frac{1}{T}\sum_{t=0}^{T-1}(R_t - V_\omega(S_t)) \log \pi_\theta(A_t|S_t)$
9:         $L(\omega) = \frac{1}{T}\sum_{t=0}^{T-1}(R_t - V_\omega(S_t))^2$
10:        Optimize $\pi_\theta$ using $\nabla L(\theta)$
11:        Optimize $V_\omega$ using $\nabla L(\omega)$
12: end procedure
```

Figure 4: Pseudocode of A2C



```
Algorithm 1 PPO, Actor-Critic Style
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 5: Pseudocode of PPO

4.2 PPO

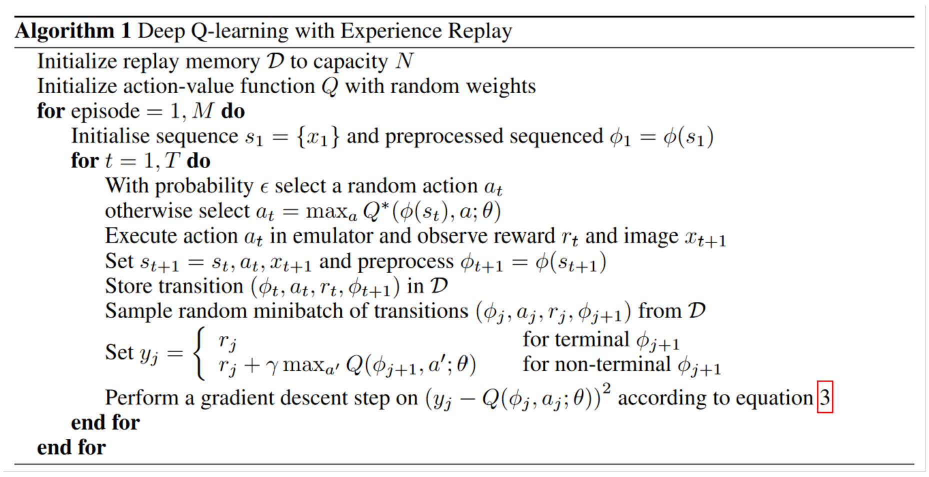
Proximal Policy Optimization (PPO) is a policy gradient method that has become the default RL algorithm at *OpenAI*. It is an improved version of Trust Region Policy Optimization, Actor-Critic style that is much easier to implement, more general and less complex. In PPO, it seeks to optimize the Clipped Surrogate Objective function: $L^{CLIP}(\theta) = \mathbb{E}[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$, where: $r_t(\theta)$ is the ratio function that calculates the probability of taking action a at state s in the current policy network divided by the previous version of the policy; $\hat{A}_t$ is the advantage function mentioned above; the clip function clips the policy ratio between $1-\epsilon$ and $1+\epsilon$ to prevent the current policy from deviating too far from the older policy (conservative). PPO uses first-order optimization to constrain the policy update, while TRPO uses second-order optimization for KL divergence constraints, hence is more efficient. [SWD+17]

4.3 DQN

In DQN, a neural network function approximator (Q-network) with weights θ is used to estimate the action-value function $Q(s, a; \theta) \approx Q^*(s, a)$. The Q-network is trained by minimizing a sequence of loss functions $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s, a; \theta_i))^2\right]$ at each iteration i where $y_i = r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$ and $\rho(s, a)$ is a probability distribution over sequences s and actions a. These loss functions are optimized by stochastic gradient descent and the probability distribution is selected by an $\epsilon-$greedy strategy. The most important technique of DQN is probably *experience replay*, where the agent's experience at each time step in a *replay memory* from which is constantly sampled and updated via Q-learning [MKS+13]. Experience replay has drawbacks such as using more memory and computation, and requiring off-policy learning algorithms that can update from data generated by an older policy. [MBM+16]

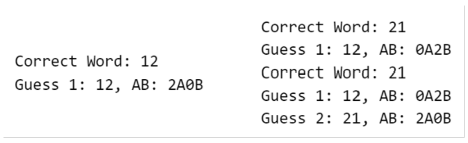
5 Model V1 Results

In the first model, I trained deep RL agents on the following two cases (explained in section 3): A. [12, 21] and B. [12, 13, 21, 23, 31, 32].



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
      end for
  end for
```

Figure 6: Pseudocode of DQN



```
Correct Word: 12
Guess 1: 12, AB: 2A0B
```

```
Correct Word: 21
Guess 1: 12, AB: 0A2B
Correct Word: 21
Guess 1: 12, AB: 0A2B
Guess 2: 21, AB: 2A0B
```

Figure 7: Example of the predictions of a trained agent for case A

5.1 Case A

Case A is a trivial case. The agent first guesses a number, and if it is wrong, the next guess is always the other number. The number can be guessed within 2 tries (Fig 7). All algorithms converge quickly due to the small state space (Fig 8,9).

5.2 Case B

For Case B, the agent first guesses a number - in this case, 21 (Fig 10). Then, it makes logical guesses based on the information given: there are only two cases 23, 32 with the same information '1A0B'. All numbers can be guessed within 3 tries. It takes around 15000 timesteps for A2C to converge and 70000 timesteps for PPO and DQN to converge (Fig 11, 12). In both cases, A2C converges the fastest. This is likely because PPO clips the effect of the Advantage function so that the new policy does not deviate too far from the older policy, so exploration is more conservative.

Certainly, modeling the original problem of *Bulls and Cows* would be too difficult.

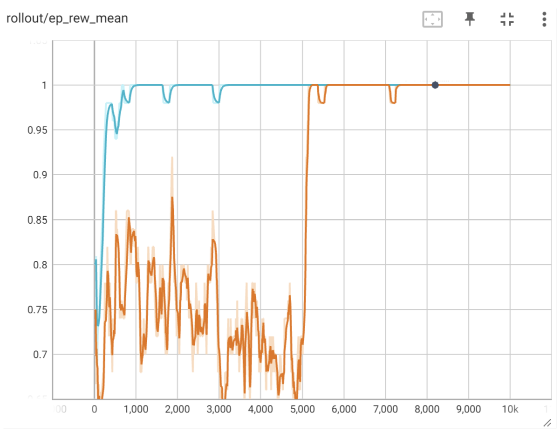


Figure 8: Mean episodic reward for case A - A2C (blue), PPO (purple), DQN (red)

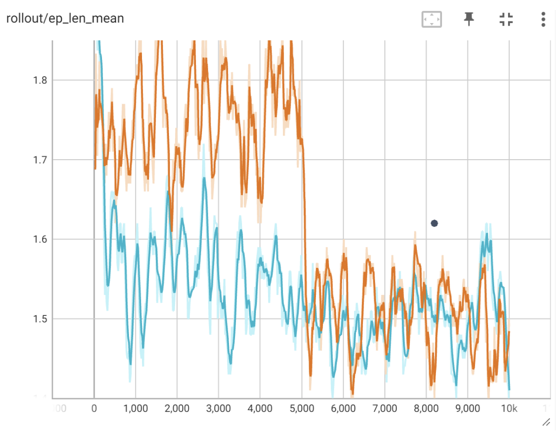
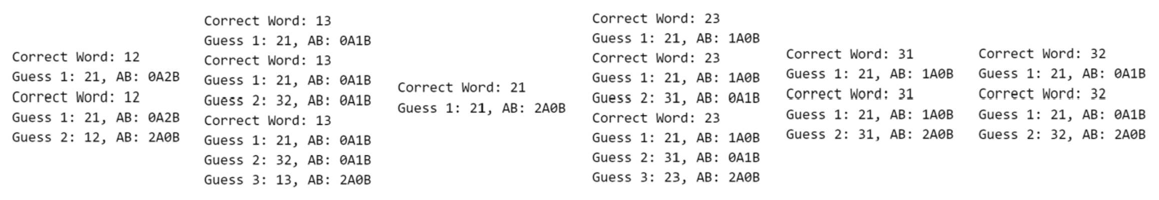


Figure 9: Mean episodic length for case A - A2C (blue), PPO (purple), DQN (red)



```
Correct Word: 12
Guess 1: 21, AB: 0A2B
Correct Word: 12
Guess 1: 21, AB: 0A2B
Guess 2: 12, AB: 2A0B
```

```
Correct Word: 13
Guess 1: 21, AB: 0A1B
Correct Word: 13
Guess 1: 21, AB: 0A1B
Guess 2: 32, AB: 0A1B
Correct Word: 13
Guess 1: 21, AB: 0A1B
Guess 2: 32, AB: 0A1B
Guess 3: 13, AB: 2A0B
```

```
Correct Word: 21
Guess 1: 21, AB: 2A0B
```

```
Correct Word: 23
Guess 1: 21, AB: 1A0B
Correct Word: 23
Guess 1: 21, AB: 1A0B
Guess 2: 31, AB: 0A1B
Correct Word: 23
Guess 1: 21, AB: 1A0B
Guess 2: 31, AB: 0A1B
Guess 3: 23, AB: 2A0B
```

```
Correct Word: 31
Guess 1: 21, AB: 1A0B
Correct Word: 31
Guess 1: 21, AB: 1A0B
Guess 2: 31, AB: 2A0B
```

```
Correct Word: 32
Guess 1: 21, AB: 0A1B
Correct Word: 32
Guess 1: 21, AB: 0A1B
Guess 2: 32, AB: 2A0B
```

Figure 10: Example of the predictions of a trained agent for case B

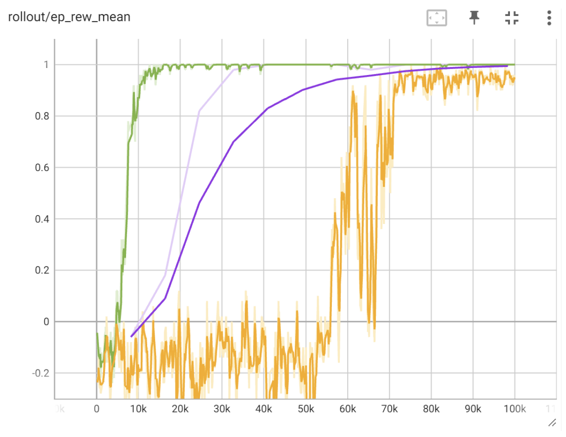


Figure 11: Mean episodic reward for case B - A2C (green), PPO (blue), DQN (orange)

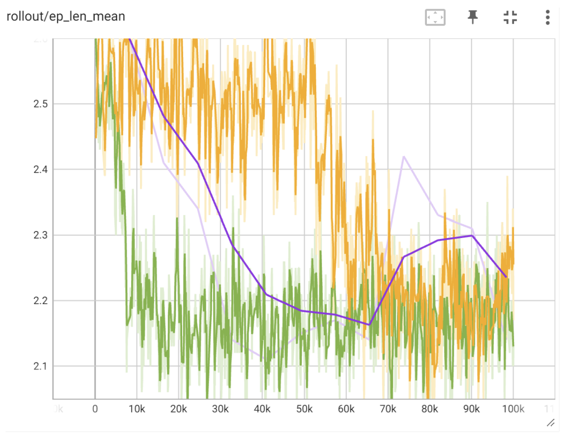


Figure 12: Mean episodic length for case B - A2C (green), PPO (blue), DQN (orange)

6 Model V2

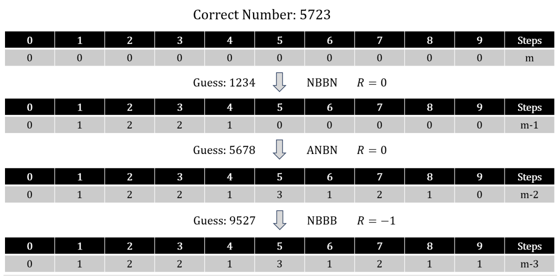
The observation space of Model V1 is too large, so I simplified the model by altering the rules such that they are similar to those of *Mastermind*. In this version, the digits that are assigned 'A' and 'B' are made known. Additionally, the game would return 'N' to any digit that does not exist in the four-digit sequence. Like V1, the maximum number of guesses allowed is restricted to m.

Example Episode:
Correct Number: 1234
Guess 1: 9824; NNBA
Guess 2: 4231; BAAB
Guess 3: 1234; AAAA   END

State Space, S_k
Using the trivial model with a grid that records each guess at each step would result in a state space extremely large. Each row would take in the number guessed, and each of the 4 digits would have 4 possibilities - 'unexplored', 'N', 'B', 'A'. An m-guesses grid would have state space of $|S| = (10 \times 4)^{4\times m}$. Assuming $m = 8$, then $|S| \approx 10^{118}$.

Hence, another model is proposed. S_k is an array of size 11: each index represents each digit $0 - 9$, and the final index represents the remaining number of guesses left (initialized as m). S_0 starts off as an array with every element denoted by 0 (unexplored). At each step k, the value of each digit of the guess is updated to: 1 if the digit has been explored and has ever attained 'N', 2 if the digit has been explored but has ever attained 'B', 3 if the digit has been explored and has ever attained 'A'. The value stays at 3 for a digit that have been found at the correct position, no matter its current position. It is only logical that learning the correct digit at the correct position would allow the agent to guess the correct digit at the correct position.

To estimate the state space complexity, each digit has 4 possible values, thus a total of $|S| = (3 + 1)^{10} \times m = 1048576 \times m$ possibilities - still considerably large, but much smaller than Model V1.


Correct Number: 5723

0	1	2	3	4	5	6	7	8	9	Steps
0	0	0	0	0	0	0	0	0	0	m

Guess: 1234 ⇩ NBBN $R = 0$

0	1	2	3	4	5	6	7	8	9	Steps
0	1	2	2	1	0	0	0	0	0	m-1

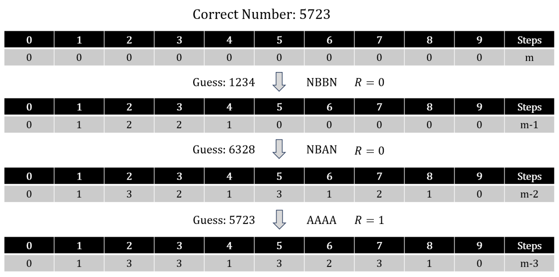
Guess: 5678 ⇩ ANBN $R = 0$

0	1	2	3	4	5	6	7	8	9	Steps
0	1	2	2	1	3	1	2	1	0	m-2

Guess: 9527 ⇩ NBBB $R = -1$

0	1	2	3	4	5	6	7	8	9	Steps
0	1	2	2	1	3	1	2	1	1	m-3

Figure 13: Example episode, $m = 4$, with -1 reward (V2)


Correct Number: 5723

0	1	2	3	4	5	6	7	8	9	Steps
0	0	0	0	0	0	0	0	0	0	m

Guess: 1234 ⇩ NBBN $R = 0$

0	1	2	3	4	5	6	7	8	9	Steps
0	1	2	2	1	0	0	0	0	0	m-1

Guess: 6328 ⇩ NBAN $R = 0$

0	1	2	3	4	5	6	7	8	9	Steps
0	1	3	2	1	3	1	2	1	0	m-2

Guess: 5723 ⇩ AAAA $R = 1$

0	1	2	3	4	5	6	7	8	9	Steps
0	1	3	3	1	3	2	3	1	0	m-3

Figure 14: Example episode, $m = 4$, with +1 reward (V2)

Action Space, A_k
Guess a 4 digit sequence, with each digit distinct. The total possible number of guesses is $|A| = 10 \times 9 \times 8 \times 7 = 5040$.

Reward, R_k
$R_k = 1$ if at the kth guess, the correct digits all have a value of 3, no matter the values of other digits; this is a terminal state. $R_m = -1$ if this is not achieved before the number of remaining guesses reaches 0, this is a terminal state. Otherwise, $R_k = 0$.

For this model, I trained deep RL agents on the following cases:

1. Numbers: Permutation of 2-digit sequence without replacement from digits $0-3$; $m = 3$
   $|A| = (3+1)^4 \times (3+1) = 1024$.
2. Numbers: Permutation of 4-digit sequence without replacement from digits $0-4$; $m = 5$
   $|A| = (3+1)^5 \times (5+1) = 6144$.

The observation space is much smaller than that of Model V1.

7 Model V2 Results

Because the value of the correct digit stays at 3 if it had ever been guessed at the correct position, the objective of the agent would be to guess the correct digits at the correct positions within a number of guesses, but not necessarily all together in one final guess. However, it can be assumed that once the agent has ever guessed the correct positions for each correct digit, guessing the correct 4-digit sequence is trivial.

As mentioned, models were trained on two cases: C. Permutation of 2-digit sequence without replacement from digits $0-3$; D. Permutation of 4-digit sequence without replacement from digits $0-4$.

7.1 Case C

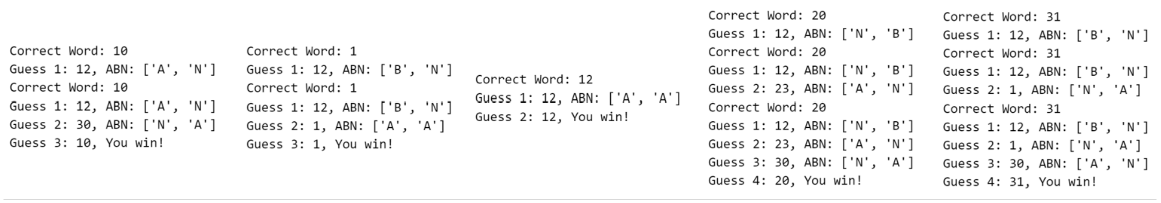
Note that in Fig 15, guess n (e.g. Guess 3: 4320, You win!) is included only to consolidate that the agent has guessed all the correct digits at the correct positions in the previous guesses. The episode ends - the agent satisfied the objective - at guess $n-1$ (e.g. Guess 2: 30, ABN:['N','A']). In this case, the agent guessed all the numbers within 3 tries.

Again, A2C converges the fastest (within 20000 timesteps), while PPO converges after around 90000 timesteps (Fig 16, 17).

7.2 Case D

The agent guesses all numbers within 5 tries (Fig 18). A2C is faster to converge, but PPO ultimately performs better after 300000 timesteps (Fig 19, 20).

In both cases, DQN does not converge towards an expected mean episode reward of 1. This could be due to the hyper-parameters or the neural network structure. It could also be due to the sparse



```
Correct Word: 10
Guess 1: 12, ABN: ['A', 'N']
Correct Word: 10
Guess 1: 12, ABN: ['A', 'N']
Guess 2: 30, ABN: ['N', 'A']
Guess 3: 10, You win!
```

```
Correct Word: 1
Guess 1: 12, ABN: ['B', 'N']
Correct Word: 1
Guess 1: 12, ABN: ['B', 'N']
Guess 2: 1, ABN: ['A', 'A']
Guess 3: 1, You win!
```

```
Correct Word: 12
Guess 1: 12, ABN: ['A', 'A']
Guess 2: 12, You win!
```

```
Correct Word: 20
Guess 1: 12, ABN: ['N', 'B']
Correct Word: 20
Guess 1: 12, ABN: ['N', 'B']
Guess 2: 23, ABN: ['A', 'N']
Correct Word: 20
Guess 1: 12, ABN: ['N', 'B']
Guess 2: 23, ABN: ['A', 'N']
Guess 3: 30, ABN: ['N', 'A']
Guess 4: 20, You win!
```

```
Correct Word: 31
Guess 1: 12, ABN: ['B', 'N']
Correct Word: 31
Guess 1: 12, ABN: ['B', 'N']
Guess 2: 1, ABN: ['N', 'A']
Correct Word: 31
Guess 1: 12, ABN: ['B', 'N']
Guess 2: 1, ABN: ['N', 'A']
Guess 3: 30, ABN: ['A', 'N']
Guess 4: 31, You win!
```

Figure 15: Example of the predictions of a trained agent for case C

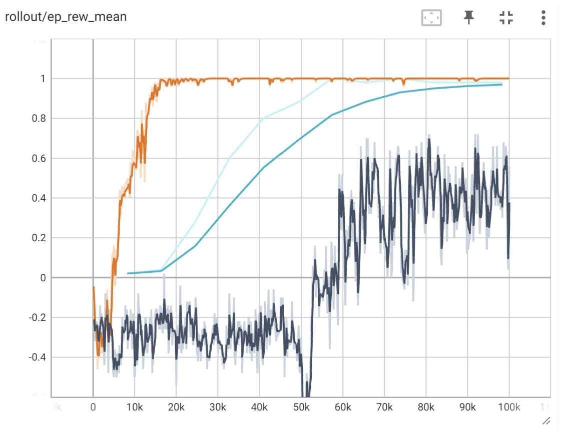


Figure 16: Mean episodic reward for case C - A2C (red), PPO (blue), DQN (dark blue)

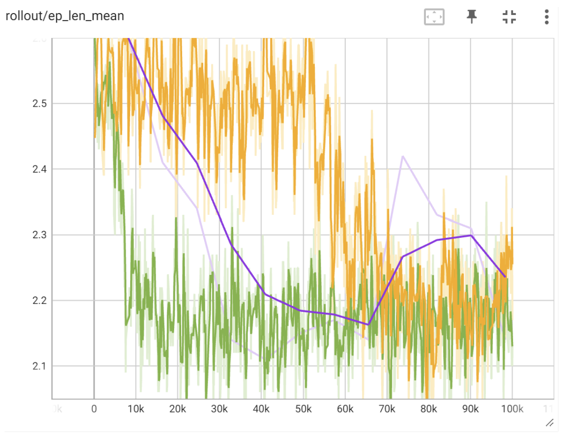


Figure 17: Mean episodic length for case C - A2C (red), PPO (blue), DQN (dark blue)

amount of rewards that would make it more difficult to run Q-learning. Sparse positive rewards in this model would require more exploration to obtain the reward that we want, thus more episodes and a longer run time.

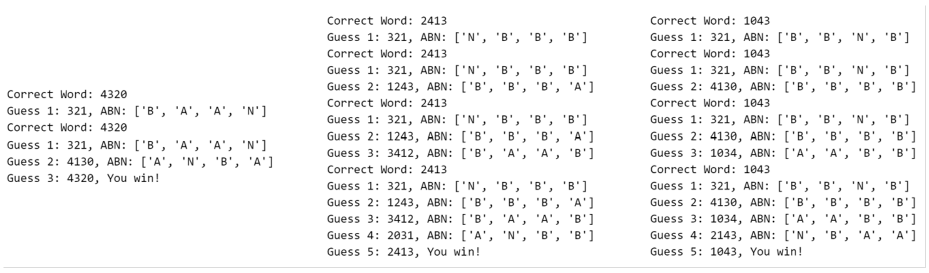
8 Possible Further Extensions

Other than possibly running the models with larger observation space with a more powerful device, there are a few things that could be expanded upon in the project to help improve learning efficiency and performance.

1. Reward Shaping. In the first model, the agent can be rewarded incrementally for obtaining 'B' and 'A', e.g. reward of 2 for the number of 'B' and 5 for the number of 'A'. In the second model, the agent can be rewarded similarly for the number of 'N', 'B' and 'A'.

2. Tweaking hyper-parameters. The general parameters such as learning rate, gamma can be altered. Specifically, for A2C and PPO, parameters like entropy and value function coefficients for loss calculation; for PPO, parameters for the clip function; and for DQN, parameters for the replay buffer and gradient descent can be changed.

3. Changing neural-network structure. The size and depth of the neural-network can be determined using prior experience from similar research - literature review on this can be done. Then, trial and error could be used to find the best architecture.



```
Correct Word: 4320
Guess 1: 321, ABN: ['B', 'A', 'A', 'N']
Correct Word: 4320
Guess 1: 321, ABN: ['B', 'A', 'A', 'N']
Guess 2: 4130, ABN: ['A', 'N', 'B', 'A']
Guess 3: 4320, You win!
```

```
Correct Word: 2413
Guess 1: 321, ABN: ['N', 'B', 'B', 'B']
Correct Word: 2413
Guess 1: 321, ABN: ['N', 'B', 'B', 'B']
Guess 2: 1243, ABN: ['B', 'B', 'B', 'A']
Correct Word: 2413
Guess 1: 321, ABN: ['N', 'B', 'B', 'B']
Guess 2: 1243, ABN: ['B', 'B', 'B', 'A']
Guess 3: 3412, ABN: ['B', 'A', 'A', 'B']
Correct Word: 2413
Guess 1: 321, ABN: ['N', 'B', 'B', 'B']
Guess 2: 1243, ABN: ['B', 'B', 'B', 'A']
Guess 3: 3412, ABN: ['B', 'A', 'A', 'B']
Guess 4: 2031, ABN: ['A', 'N', 'B', 'B']
Guess 5: 2413, You win!
```

```
Correct Word: 1043
Guess 1: 321, ABN: ['B', 'B', 'N', 'B']
Correct Word: 1043
Guess 1: 321, ABN: ['B', 'B', 'N', 'B']
Guess 2: 4130, ABN: ['B', 'B', 'B', 'B']
Correct Word: 1043
Guess 1: 321, ABN: ['B', 'B', 'N', 'B']
Guess 2: 4130, ABN: ['B', 'B', 'B', 'B']
Guess 3: 1034, ABN: ['A', 'A', 'B', 'B']
Correct Word: 1043
Guess 1: 321, ABN: ['B', 'B', 'N', 'B']
Guess 2: 4130, ABN: ['B', 'B', 'B', 'B']
Guess 3: 1034, ABN: ['A', 'A', 'B', 'B']
Guess 4: 2143, ABN: ['N', 'B', 'A', 'A']
Guess 5: 1043, You win!
```

Figure 18: Example of the predictions of a trained agent for case D

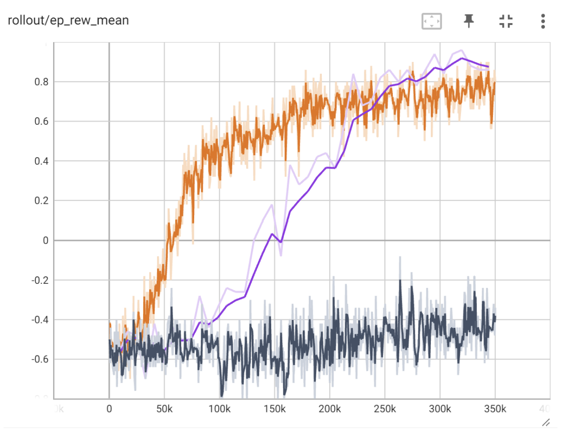


Figure 19: Mean episodic reward for case D - A2C (red), PPO (blue), DQN (dark blue)

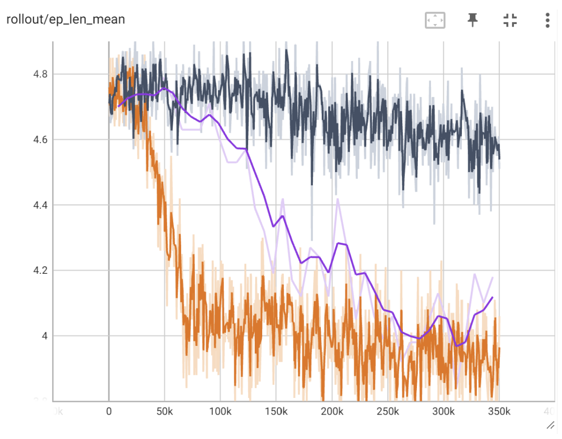


Figure 20: Mean episodic length for case D - A2C (red), PPO (blue), DQN (dark blue)

References

[ADBB17] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[MBM$^+$16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[MKS$^+$13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[Ope] OpenAI. Part 2: Kinds of rl algorithms.

[SS] Alexey Slovesnov Slovesnov.

[SWD$^+$17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[WMLS] Yuhuai Wu, Elman Mansimov, Shun Liao, and Alec Radfordand John Schulman. Openai baselines: Acktr a2c.