CS698R: Deep Reinforcement Learning Project

Final Project Report

Team Name: Brute Force
Project Title: Restless Multi Arm Bandit
Project #: 4
Team Member Names: Roll No
Arpit Agarwal: 180139
Abhinav Kumar: 16907018
Vartika Gupta: 180849
Suman Singha: 180793

1 Introduction

The Restless Multi-Armed Bandit Problem (RMABP) is essentially a game between a player and an environment. There are K arms and the state of each arm keeps evolving according to an underlying distribution and it transitions at every time slot during the episode (one full play of the game). At each time slot, the player pulls one of the arms and receives a reward. The goal of the game is to maximize the reward received over T time steps.

The RMABP has myriad applications including but not limited to Job Allocation in a server, Channel Detection in a wireless network, Health Care Systems, Dynamic Posted Pricing, dynamic UAV routing, etc. as can be seen in Biswas et al. [2021]

Most existing work on RMABP solves the offline setting where the underlying working of the game is known. This does not cover most practical scenarios where the underlying MDP of the environment is unknown to the agent.

The more practical and more challenging is the online setting where the underlying MDP is unknown and a robust policy is needed to tackle the involved challenges. In the section ahead we discuss some of the state-of-art-work with RMABs. For the purpose of this project we will aim towards better understanding and following up on the Deep Reinforcement Learning heavy methods.

2 Related Work

The Restless Multi-Armed Bandit Problem (RMABP) and its variations with limited budget due to action cost and/or constraints over actions have been solved using various optimization techniques. First we look at methods with non-DRL techniques.

In paper Fu et al. [2019] and Borkar and Chadha [2018], authors build their work upon the Whittle Index Policy which requires the knowledge of the underlying MDP which is rarely known in a practical setting. The main idea is to assign an index to every arm and then choose the arm with the largest index and then update all indices and so on. Parallel Q-learning recursions are utilized to learn the whittle indices which can then solve sub-problems with reduced state space in order to maximize the average reward of the whole system in the long run. Whittle index policy is asymptotically optimal under certain conditions given in Weber and Weiss [1990].

In paper Wang et al. [2020], author implements a Restless-UCB framework which has a guaranteed regret bound which is of polynomial complexity in contrast with exponential complexity of previous methods. Moreover, this framework works with a general online restless bandit problem where the underlying MDP is not known.

Now, we discuss DRL based novel approaches to solve the RMABP.

In paper Killian et al. [2021], authors solves the challenging generalization of the RMABP by developing a minimax regret objective i.e., minimizing the worst possible regret for the optimal policy and Proximal Policy Approximation (PPO). It also takes into account any cost involved with taking actions. They also extend this approach to a multi-agent setting.

The below mentioned papers solve the channel access problem in wireless networks by using a RMABP formulation with Deep RL techniques.

3 Problem Statement

Like other multi-arm bandits, the four-armed bandit being modelled has one non-terminal (starting) state and four terminal states. This is a single time step (1-step horizon) process, which means each episode contains one action. The agent chooses one action vis-à-vis one arm. And this action places the agent in one state, specific to that action. In other words, there is no stochasticity in reaching the desired state and taking action. Here is a visual of the above-mentioned environment:

The environment gives rewards, specific to an action. This reward is sampled from a Gaussian distribution of time-varying mean and a fixed variance, which makes the restless bandits different from a stationary one. This fixed variance which is common to all bandits can be seen as expected variance. After each time step, the mean of reward function of every state is updated according to the following equation:

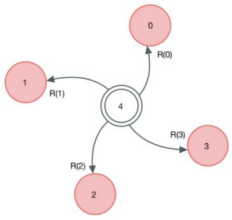


Figure 1: Restless Four-Arm Bandit

$$R_j(t) = \mu_j(t) + \epsilon_j(t)$$
$$\mu_j(t) = \lambda\mu_j(t-1) + \kappa_j + \zeta_j(t) \quad (1)$$

where $\epsilon_i - \mathcal{N}(0, \sigma_\epsilon) \forall j \in \{0, 1, 2, 3\}$ and $\zeta - \mathcal{N}(0, \sigma_\zeta) \forall j \in \{0, 1, 2, 3\}$.

Acc to Speekenbrink M [2015], the weighted factor λ makes sure that the reward values at a certain time are bounded within some limits. A number sampled from a Gaussian distribution of zero mean(μ) and variance(σ) is added to the previous mean, making the environment states time-dependent. Here, σ is called unexpected variance, which, unlike in the stationary bandit case. This unexpected variance can be time-dependent. The constant parameter(k_i) introduces a trend in the reward function.

From the different parameter values of σ and k_i, there can arise four cases, i) **Stable Variance with the trend**, ii) **Stable variance without trend**, iii) **Variable variance with trend** and iv) **Variable variance without trend**. Unexpected variance is low and constant for stable variable cases whereas it is low for some periods and high for some periods in variable variance cases. Trend value is zero for without trend cases. The agent tries to maximize its cumulative rewards for a certain number of episodes.

4 Environment Details and Implementation - Demo

Transition Function: We assume that before selecting an arm, the agent does not know the states of arms in the time step. We assume that the agent receives the reward of a single arm at a time and learns its state. Also, we assume that the state of each arm is dynamically switching based on the distribution described in (1). The probability of transition from one state to another is 1 since one action always takes the agent to a particular state. The agent aims to learn the pattern of variations of arm states based on previous decisions and received rewards.

Reward Function: The reward is time-dependent. At each time step, the agent gets a reward and the reward function gets updated according to the equation (1).

We have taken $\lambda = 0.99$ and initial reward means are -60, -20, 20, 60. For trend cases, κ_j are 0.5, 0.5, -0.5 and -0.5 for arms. The total episode length is 200. For stable cases, variance is 4 for all episodes and for variable case variance is 4 for 0 to 50 and 101 to 150, and variance is 16 for the rest of the episodes.

p-function/MDP: We have taken state 4 as the starting non-terminal state and 0, 1, 2, 3 are terminal states. From the above-mentioned transition and reward functions, we can write the p-function.

```
4 : {
        0: [(1, 0, N(μ0(t), σe), true]      state is same as action
        1: [(1, 1, N(μ0(t), σe), true]      transition probability is 1
        2: [(1, 2, N(μ0(t), σe), true]      mu's are function of time(t)
        3: [(1, 3, N(μ0(t), σe)), true]
    },
0 : {
        0: [(1, 0, 0, true)]                all the terminal state list will be similar
        1: [(1, 0, 0, true)]
        2: [(1, 0, 0, true)]
        3: [(1, 0, 0, true)]
    },
```

5 Proposed Solution

At first, we have used classical Reinforcement Learning techniques to train the agent. We have tried pure exploration and pure exploitation, epsilon greedy, decaying epsilon greedy, softmax and UCB. Though UCB and Softmax worked better than other algorithms, all these techniques got stuck at some sub-optimal level.

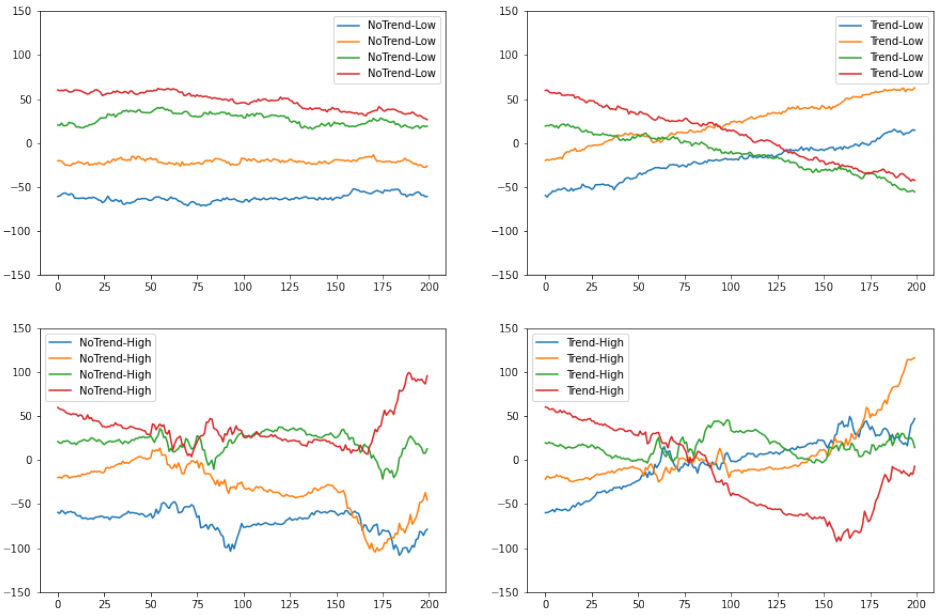


Figure 2: Reward Values of each arm averaged over 10 runs

We moved to deep learning-based algorithms to get better results. Our key challenge was to make the environment such that we can use deep learning based methods. From the problem statement itself, it's very intuitive that the agent needs some knowledge about the history of rewards. So, we define the whole game of 200 steps as an episode and each draw of arms as a time-step. And at each step, we define the state of the environment as a collection of previous m actions and corresponding observed rewards, i.e. $s_t = [a_{t-1}, r_{t-1}, \ldots, a_{t-m}, r_{t-m}]$. Now this length of vector m has to be such that it contains minimum information that is enough to predict the next arm which will have the highest reward. In the extreme case, this m can be 200, which means that we feed all the previous history to the agent. At first impression, this may seem like that environment is providing some inside information to the agent, but in another view, this is like the agent has some memory which can contain information of at most m actions which is very analogous to human. We have treated m as a hyperparameter and trained the agent using DQN. This approach gave a significant improvement over classical algorithms. With the greater value of m, the agent performed better. Though this approach gave a very good result, after a certain point, it was not improving even after increasing 'm'.

There is some flaw in the assumption that information about previous m actions and rewards is enough to predict the next step because if all the 'm' steps are of the same action the agent will have no information about other arm rewards. So, we modified our state as a 4*k matrix where i^{th} row contains the last 'k' reward history of the i^{th} arm. Now with this modification, the agent will always have some information about each arm. We trained the agent using k as a hyperparameter with DQN. After tuning hyperparameters, this agent takes optimal action almost every time. We have then used DDQN and PPO to further improve the performance. All the hyperparameters are mentioned in Appendix A.

6 Experiments

We have implemented our code on google colaboratory. We mainly implemented a DQN based algorithm Mnih et al. [2013] followed by the PPO approach Schulman et al. [2017]. We defined the input vector as a collection of n consecutive actions and rewards. The agent trained was failing to choose optimal action in cases with high variance, so we tried tuning the hyperparameters like gamma, exploration fraction, learning rate, batch size etc. but the performance did not improve much. We also tried it for different values of n. While increasing the value of n makes the agent learn in a better way, the problem with high variance cases is still present.

Then we modified the input vector as a 4*k memory matrix as mentioned in proposed solution. We trained the agent using DQN, did some hyperparameter tuning and found it to perform optimally almost every time. We then used DDQN and PPO to further improve the agents' performance. All the hyperparameters are mentioned in Appendix A.

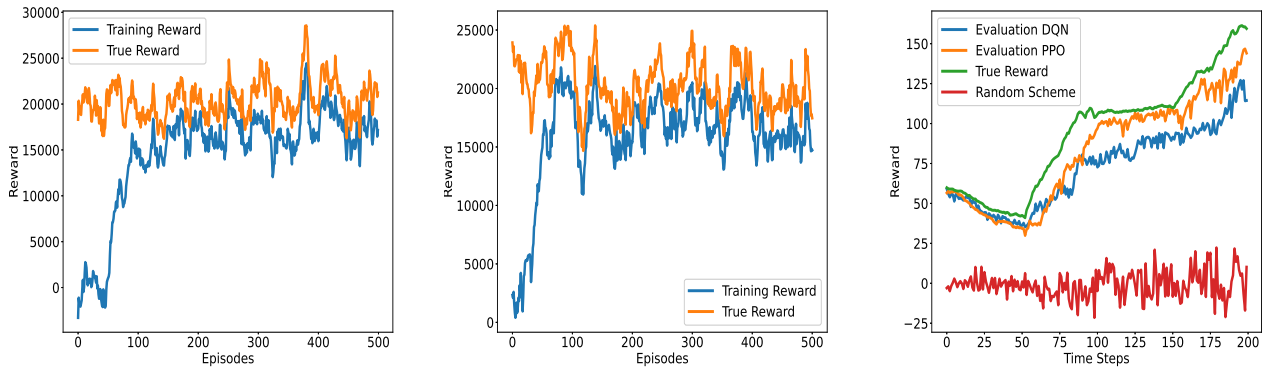
While hyperparameter tuning of the memory matrix of 4 arms, the number of previous values to be stored for each arm was optimized by iterating over the training for different values of k and observing the MSE Loss in the training reward. We observed that storing more than 10 timesteps for each arm slowed down the training and timesteps less than 4 were insufficient. In the [4, 10] range for the value of k, the least possible value of k without degradation in performance was k = 8.

Hyperparameters of Environment	
k	8
γ	0
λ	0.99
Expected Variance	2
Unexpected Variance (Low)	4
Unexpected Variance (High)	16
κ_j	[-0.5, -0.5, 0.5, 0.5]

Design Choices for DQN	
Approximate action-value function	Q(s, a; θ)
State-in values-out feedforward neural network	Nodes : (32, 400, 300, 4)
Loss Function	$L(\theta_i) = \mathbb{E}_{s,a,r,s'}[(r + \gamma * max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i))^2]$
Decaying Epsilon-greedy Strategy	ϵ decaying from 1 to 0.01
learning rate	0.01
buffer size	1000000
batch size	32

Design Choices for PPO	
Policy Network	Nodes : (32, 400, 300, 4)
Value Network	Nodes : (32, 404, 300, 4)
Loss Function	$L(\theta, \theta^-) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{U}(\theta^-)}[max(G - V(s;\theta), G - (V + clamp(V(s,\theta) - V, -\delta, \delta)))]]$
learning rate	0.01
buffer size	1000000
batch size	32

7 Results and Analysis



(a) DQN training (b) PPO training (c) Evaluation averaged over 100 games

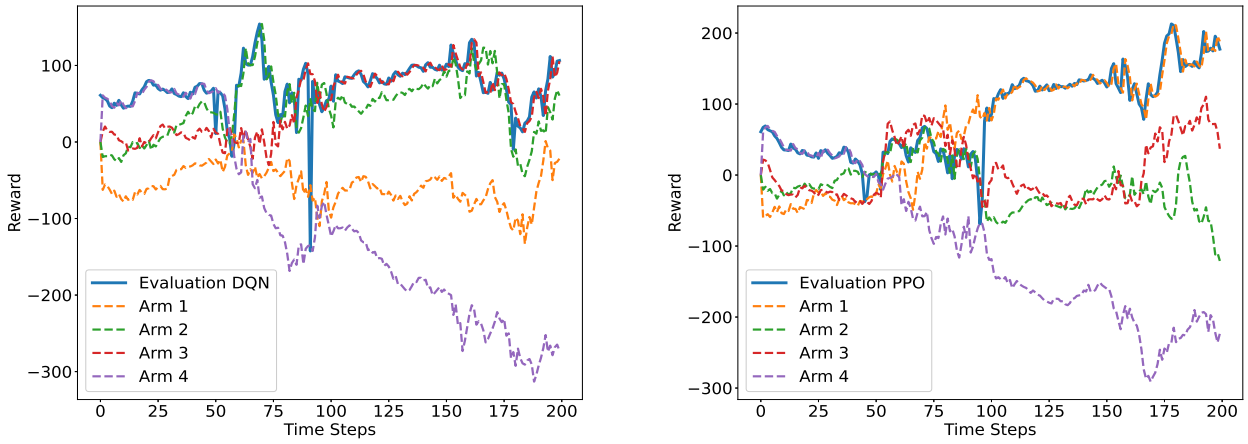
Figure 3: Environment with trend and high volatility

The reward function of the environment is noisy and the agent does not have access to environment dynamics and hence it can never reach the true achievable reward due to the randomness of the reward itself.

We plot in Fig. 7a the training performance of the DQN agent by comparing the evolution of reward with the true achievable reward. For the first 100 episodes the epsilon is decayed linearly from $\epsilon = 1$ to $\epsilon = 0.01$, and we see that as soon as epsilon settles, the DQN performance peaks as it starts exploiting the learnt policy and then maintains its performance.

In Fig. 7b we do the same for the PPO agent. PPO uses action space noise and is able to climb up to the peak performance much before 100 episodes without any falls. This is because PPO uses clipping while updating the gradients and therefore abrupt changes are avoided and hence it gradually moves towards the optimal policy instead of fluctuating to bad policies.

In Fig. 7c we evaluate our trained DQN and PPO agents on 100 instances of the game (environment) and average out the results. We also contrast the performance of our trained agents with true achievable reward and a random scheme in which an action is chosen at random at every time step. We observe that PPO outperforms DQN, especially in the later part of the game when trend and high volatility takes full effect.



(a) DQN performance (b) PPO performance

Figure 4: Single game-play with agents

In Fig. 4a, 4b we plot the performance of agents versus the evolution of 4 arms. We observe that as soon as the value of the arm that the agent is following falls down, agent tries to switch the arm and is successful in most cases.

8 Future Directions and Conclusions

It can be observed that, because the environment is highly stochastic, the agent can never achieve the optimal rewards. Also, the agent not only learns on the basis of the rewards it may achieve having selected an action, but also from the memory matrix. The memory matrix allows the agent to learn the general distribution of all the arms and exactly when to explore instead of exploiting the arm with high reward. Currently, we focussed on a single agent setting where a single arm was selected out of the four bandits. So, the optimal action only depended on the reward it achieved after selecting the action. This work can be extended to a multi-agent setting where multiple scenarios can be explored, for example, the arm selection of all the agents are trained on the basis of their cumulative reward. Similarly, there can be an extended condition where all the agents cannot select the same arm and hence they will be inter dependent.

Here, in our problem statement, the total time steps for which the agent has to choose an action is already known. If there in no finite time steps given beforehand, then the agent has to be trained in a online setting. The agent need to be trained in a very robust environment with a large dataset so that the neural network can ganeralize each and every situation that can arise in any time step.

9 Member Contributions

Member	Work Done			
Arpit	Literature Review	DQN, PPO Implementation	Analysis and Results	PPT and Report
Abhinav	Literature Review	DQN, PPO Implementation	Analysis and Results	PPT and Report
Vartika	Literature Review	DQN, PPO Implementation	Analysis and Results	PPT and Report
Suman	Environment Implementation	DQN, PPO Implementation	Analysis and Results	PPT and Report

References

Arpita Biswas, Gaurav Aggarwal, Pradeep Varakantham, and Milind Tambe. Learn to intervene: An adaptive learning policy for restless bandits in application to preventive healthcare. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4039–4046. International Joint Conferences on Artificial Intelligence Organization, 2021. doi: 10.24963/ijcai.2021/556.

Vivek S Borkar and Karan Chadha. A reinforcement learning algorithm for restless bandits. In *2018 Indian Control Conference (ICC)*, pages 89–94. IEEE, 2018.

Jing Fu, Yoni Nazarathy, Sarat Moka, and Peter G Taylor. Towards q-learning the whittle index for restless bandits. In *2019 Australian & New Zealand Control Conference (ANZCC)*, pages 249–254. IEEE, 2019.

Jackson A Killian, Lily Xu, Arpita Biswas, and Milind Tambe. Robust restless bandits: Tackling interval uncertainty with deep reinforcement learning. *arXiv preprint arXiv:2107.01689*, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

Konstantinidis E Speekenbrink M. Uncertainty and exploration in a restless bandit problem. pages 351–67, 2015. doi: 10.1111/tops.12145.

Siwei Wang, Longbo Huang, and John Lui. Restless-ucb, an efficient and low-complexity algorithm for online restless bandits. *arXiv preprint arXiv:2011.02664*, 2020.

Richard R Weber and Gideon Weiss. On an index policy for restless bandits. *Journal of applied probability*, 27(3): 637–648, 1990.

Appendix

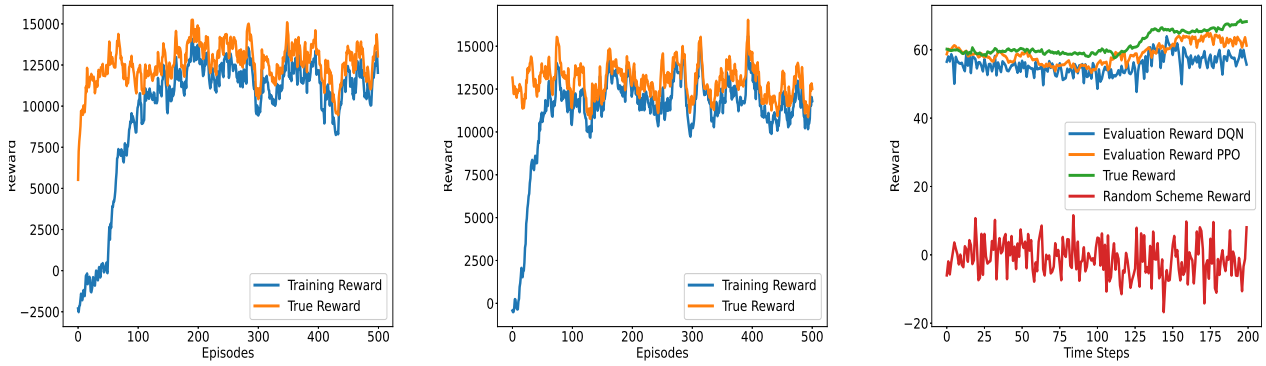
10 Extra Plots

(a) DQN training (b) PPO training (c) Evaluation averaged over 100 games

Figure 5: Environment with trend and low volatility

(a) DQN training (b) PPO training (c) Evaluation averaged over 100 games

Figure 6: Environment with no trend and high volatility



(a) DQN training (b) PPO training (c) Evaluation averaged over 100 games

Figure 7: Environment with no trend and low volatility