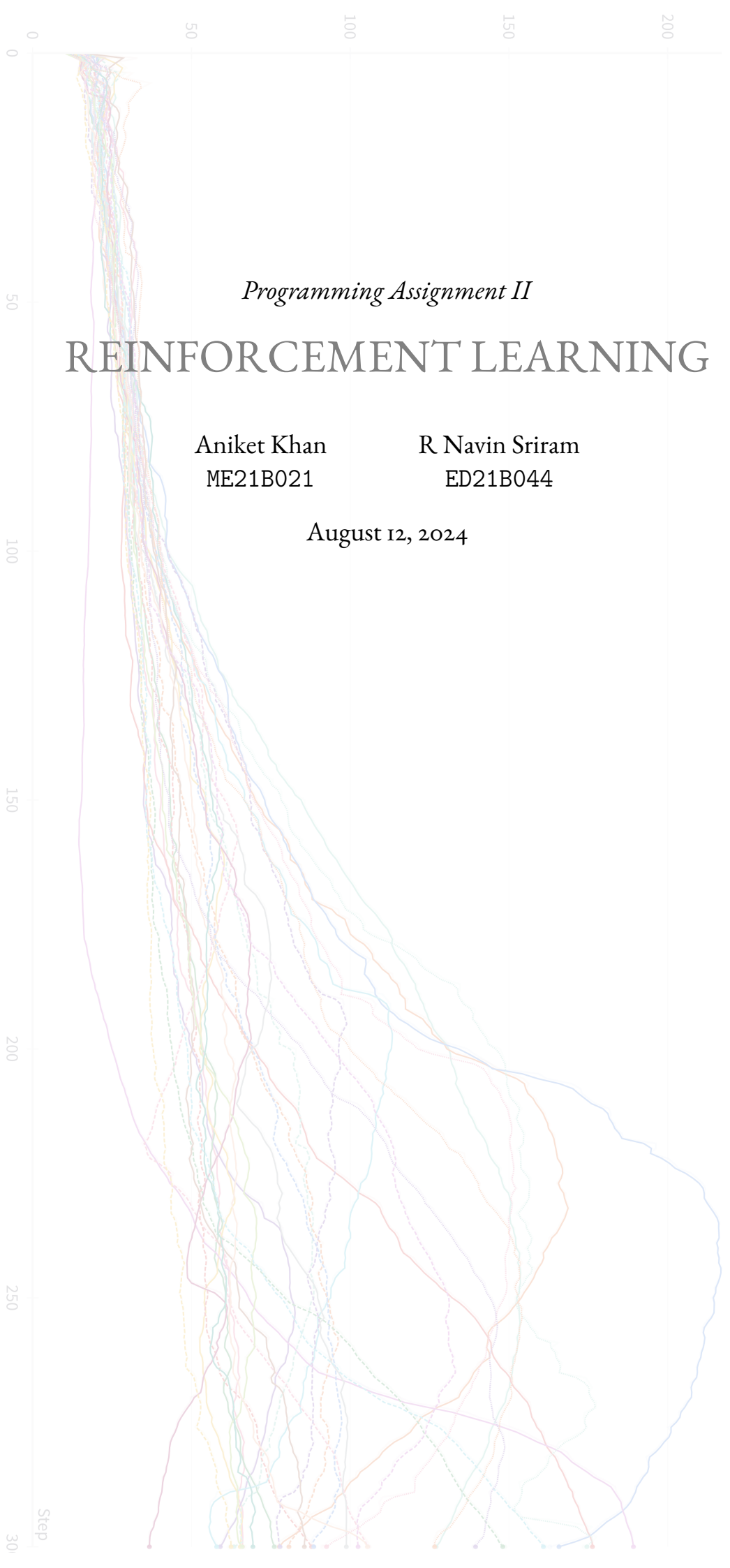
Programming Assignment II

REINFORCEMENT LEARNING

Aniket Khan
ME21B021

R Navin Sriram
ED21B044

August 12, 2024



Contents

1

Structure of the Project

1.1 Project Files

We've organized our assignment into two separate files, each dedicated to implementing different algorithms. Specifically, we've implemented both type 1 and type 2 dueling DQN in one script ($DuelingDQN.ipynb$), and MC reinforce with and without baseline in a unified script ($ReinforceMonteCarlo.ipynb$) with two classes for implementing With and Without Baseline. Additionally, at the end of each file, we've included a tuning script to adjust hyperparameters such as hidden layer size, learning rates and batch size. This structured approach allows for clearer separation of tasks and facilitates the optimization of algorithm performance through systematic parameter tuning. Moreover, we've set a fixed number of episodes for training termination to ensure consistency in the number of episodes across runs, enabling us to take averages and plot the results effectively.

2

Hyperparameters and Tuning :

We implement **WANDB** (Weights and Biases) to help us analyse the effects of changing the hyper-parameters and tuning them. WANDB is imported as a library and the tuning script (detailed in the section, Code Snippets) runs **bayesian optimisation** to fine tune the declared HyperParameters. WANDB syncs and saves all of the runs and logs values to the **cloud**, making it easy to maintain it across a group and monitor remotely.

- The objective of the optimization process was to **minimize the regret** during training of the agents on the various environment.
- We defined an objective function to evaluate the performance of the agent under different hyperparameter settings and conducted a **hyperparameter sweep**.
- Each iteration of the sweep involved training the agent with a specific configuration, logging the resulting average reward and regret, and iteratively refining the search space to converge towards optimal hyperparameters.

2.0.1 Dueling DQN:

- **Batch Size:** Tuning batch size is crucial because it determines the quality of the gradient estimates used for updating the network parameters, impacting convergence speed and stability. A larger batch size may lead to a more stable training process and smoother convergence, but it requires more memory and computational resources.

- **Update Rate:** The frequency at which the target network is updated affects the stability of the training process and the speed of convergence.it helps stabilize the learning process by providing a stationary target for the value estimation, preventing oscillations or divergence during training.
- **Learning Rate:** Determines the step size during parameter updates. Optimizing the learning rate is vital as it governs the step size in the parameter space, influencing the speed and quality of convergence towards an optimal solution, thus directly impacting the overall performance.

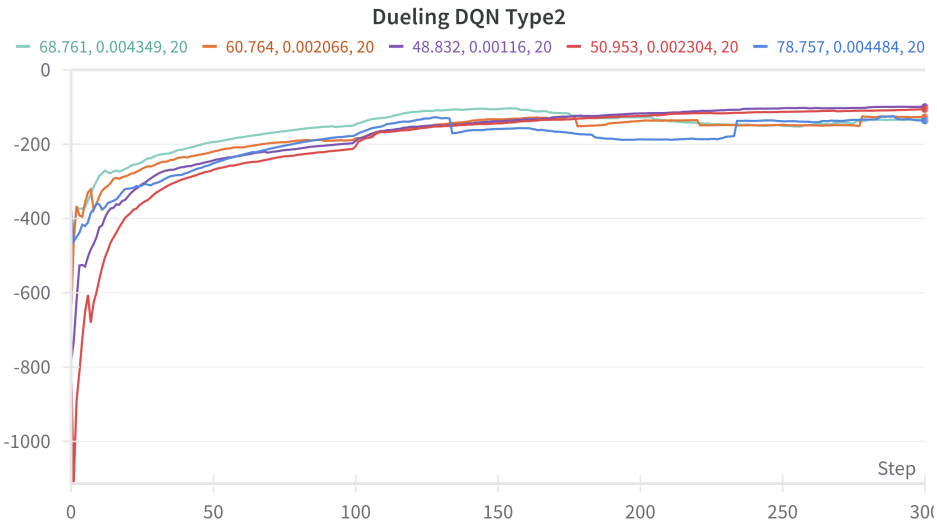


Figure 1: Type 2 update, Acrobot (Batch Size, lr, update)

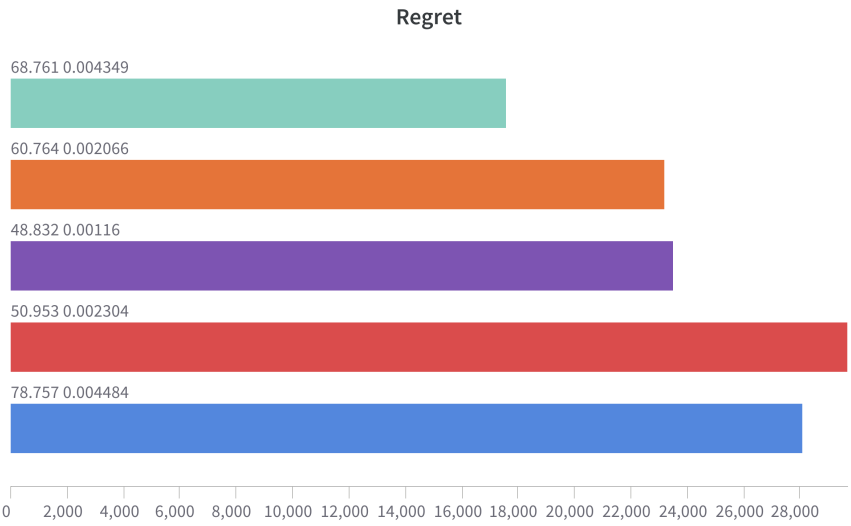


Figure 2: Type 2 Regret, Acrobot (Batch Size, lr, update)

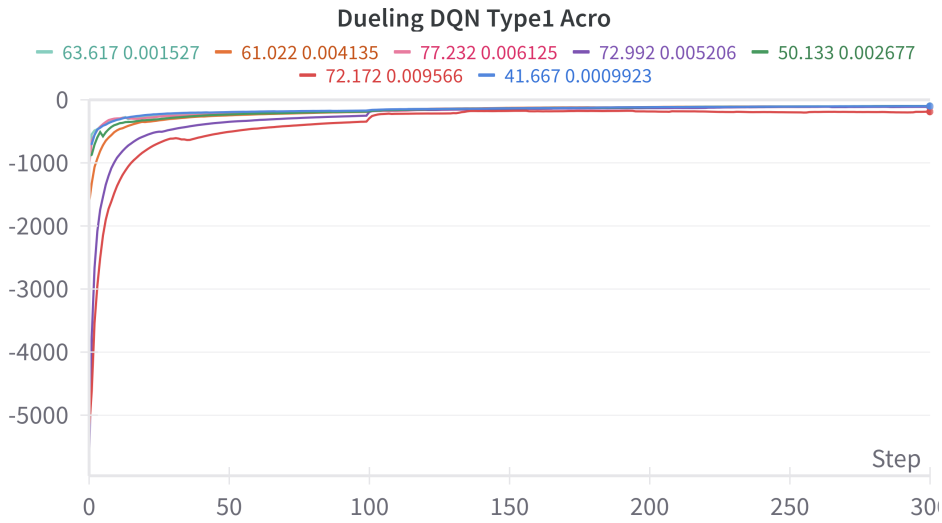


Figure 3: Type 1 update, Acrobot (Batch Size, lr)

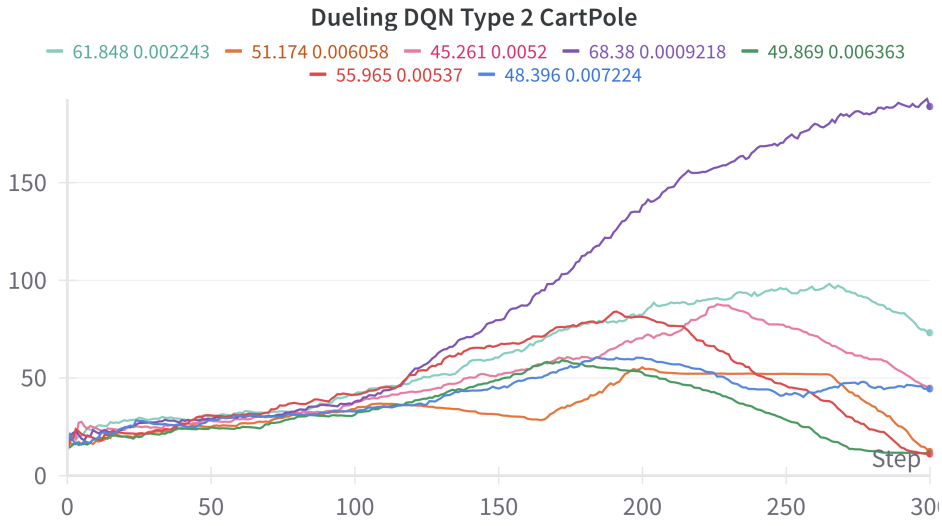


Figure 4: Type 2 update, CartPole (Batch Size, lr, update)

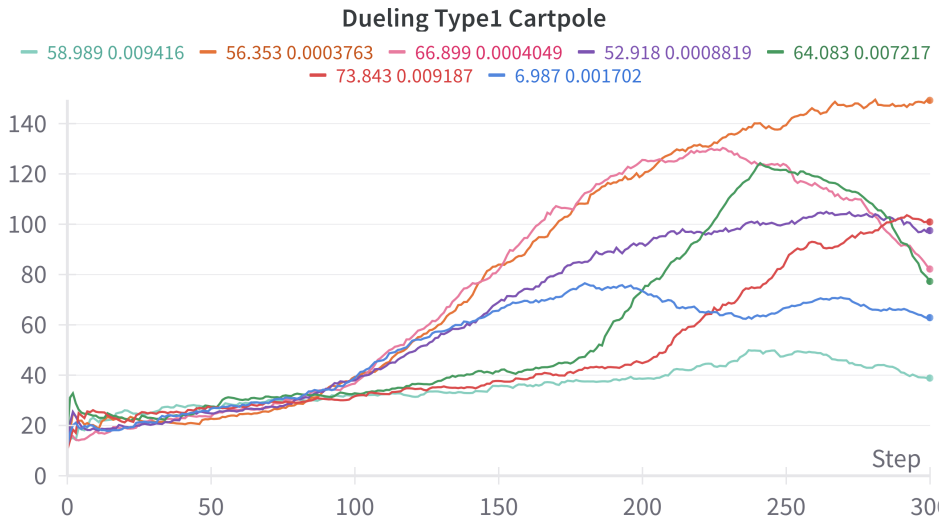


Figure 5: Type 1 update, CartPole (Batch Size, lr, update)

Selected Hyper Parameters :

- **Acrobot** Type 1 **Batch** : 63, **LR** : 0.001527
- **Acrobot** Type 2 **Batch** : 68, **LR** : 0.004349
- **CartPole** Type 1 **Batch** : 58, **LR** : 0.009416
- **CartPole** Type 2 **Batch** : 62, **LR** : 0.002243

2.0.2 MC Reinforce with Baseline

- **Learning rate of policy network:** Affects the rate at which the policy parameters are updated. Higher learning rates may lead to faster learning but can also cause instability and oscillations.
- **Learning rate of value network:** Determines how quickly the value function parameters are updated. Balancing this with the policy network's learning rate is crucial for ensuring stable training. This is usually greater than the Policy Net learning rate
- **Number of neurons in the hidden layer:** The size of the hidden layer in the neural network architecture impacts the model's capacity to learn complex representations. Increasing the number of neurons can improve the model's ability to capture intricate patterns but may also increase training time and risk overfitting.

2.0.3 MC Reinforce without Baseline

- **Learning rate of policy network:** Similar to Reinforce with Baseline, it determines the rate of policy updates and affects the speed and stability of learning.

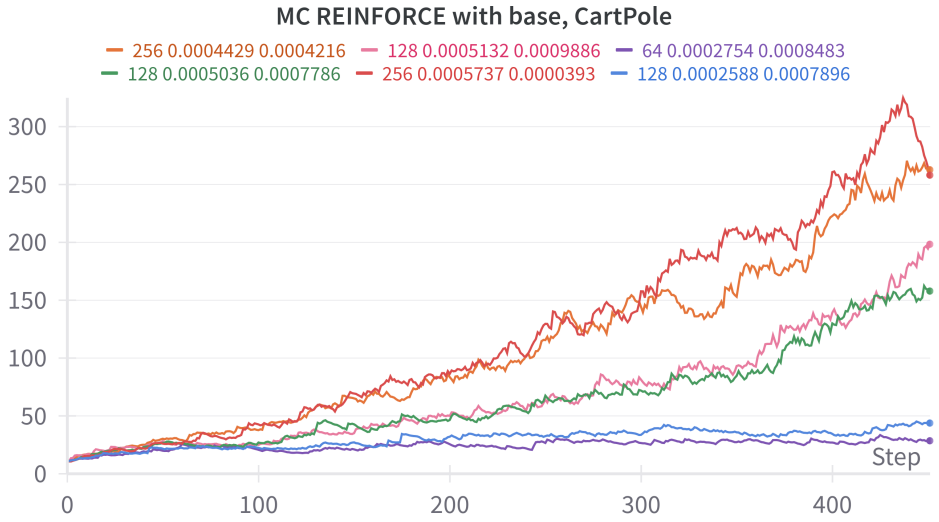


Figure 6: MC Reinforce with base for CartPole

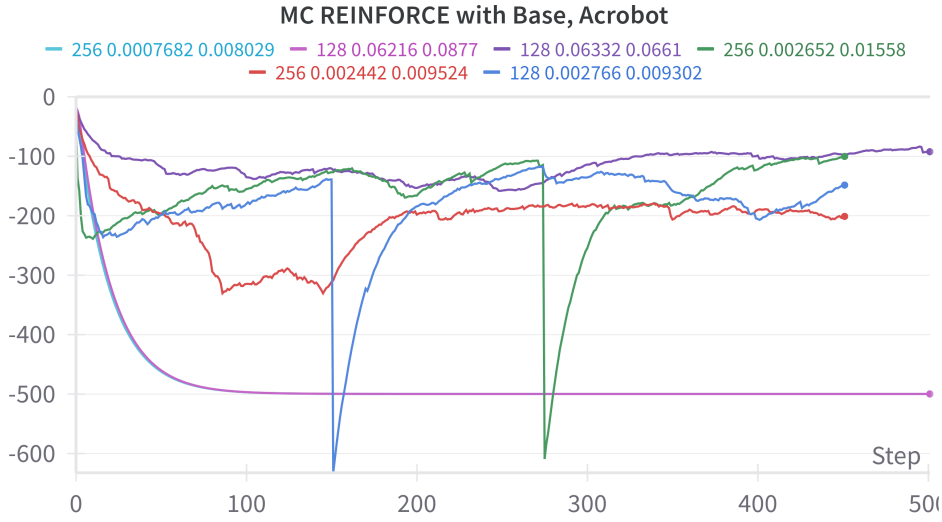


Figure 7: MC Reinforce with base for Acrobot

- **Number of neurons in the hidden layer:** Similar to Dueling DQN, the size of the hidden layer influences the model's capacity to learn complex representations and can impact training stability and convergence.

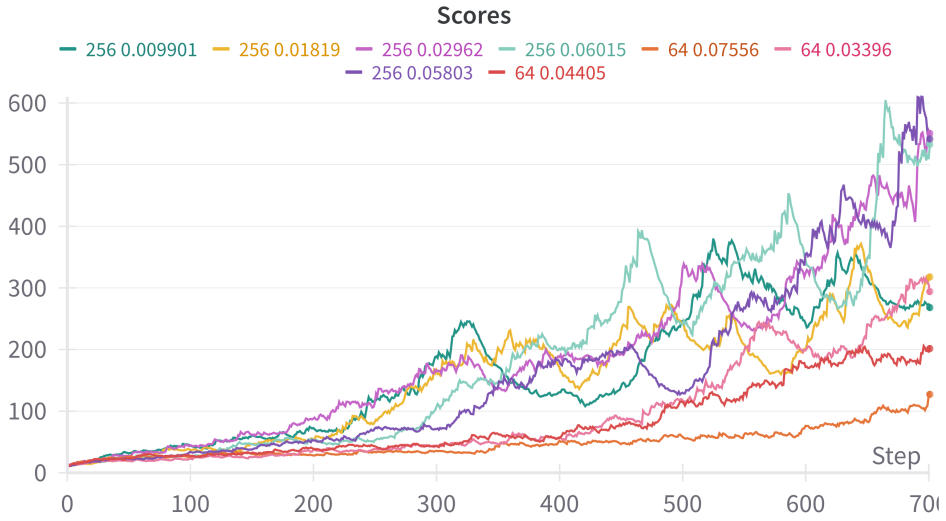


Figure 8: MC Reinforce without Baseline, Cartpole

Parameters : (Hidden Neurons size , Policy Network LR, Baseline Network LR)

- **CartPole Baseline** (256, 0.0004429, 0.0004216)
- **CartPole without** (256, 0.009901)
- **Acrobot Baseline** (128, 0.006332, 0.0661)
- **Acrobot without** (256, 0.0007682)

2.1 Plots

- **Average Episodic Reward vs Episodes** for each Type-1 and Type-2 algorithms in each of the environments the best policy

2.2 Code

- Here is the github repository link for the assignment.

3 Code-Snippets

3.1 Dueling DQN:

Note: Here, the variable *hyp* refers to the hyper-parameter in either of the 2 algorithms.

Action Selection

```
def act(self, state, policy, hyp=0.):

    state = torch.from_numpy(state).float().unsqueeze(0).to(
       device)
    self.qnetwork_local.eval()
    with torch.no_grad():
        action_values = self.qnetwork_local(state)
    self.qnetwork_local.train()

    ''' Epsilon-greedy action selection '''
    if(policy == "eps greedy"):
      if random.random() > hyp:
          return np.argmax(action_values.cpu().data.numpy())
      else:
          return random.choice(np.arange(self.action_size))

    ''' Softmax action selection '''
    if(policy == "softmax"):
      action_probs = softmax(action_values.cpu().data.numpy().
         flatten() / hyp)
      return np.random.choice(np.arange(self.action_size), p=
         action_probs)
```

Implementing Type-1 or Type-2 Duelling DQN

```
def forward(self, state):
    x = F.relu(self.fc1(state))
    x = F.relu(self.fc2(x))
    advantage = self.advantage_fc(x)
    value = self.value_fc(x)
    if self.type ==1:
        q_values = value + (advantage - advantage.mean(dim = 1,
            keepdim = True))
    else:
        q_values = value + (advantage - torch.max(advantage))
    return q_values
```

3.2 MC REINFORCE:

MC-Reinforce with no baseline

```
def update(self,rewards, states, actions):
        G = 0
        gamma = 0.99
        self.optimizer.zero_grad()
        for i in reversed(range(len(rewards))):
            reward = rewards[i]
            state = torch.tensor(states[i].reshape(1, -1),
                                 dtype=torch.float)
            action = torch.tensor(actions[i]).view(-1, 1)
            log_prob = torch.log(self.model(state).gather(1,
               action))
            G = gamma * G + reward
            loss = -log_prob * G
            loss.backward()
        self.optimizer.step()
        del self.model.episode_rewards[:]
        del self.model.saved_actions[:]
        del self.model.episode_states[:]
```

MC-Reinforce with baseline

```
def learn_Value(self, states, actions, rewards, next_states
    , dones):
    next_states = torch.tensor(next_states).to(device)
    V_targets_next = self.vnetwork_target(next_states).
        detach()
    V_targets =  + (gamma * V_targets_next * (1 - dones))
    actions = torch.tensor(actions).view(-1, 1).to(device)
    V_expected = self.vnetwork_local(torch.tensor(states).
        to(device))
    loss = F.mse_loss(V_expected, V_targets)
    self.optimizerV.zero_grad()
    loss.backward()
    for param in self.vnetwork_local.parameters():
            param.grad.data.clamp_(-1, 1)
    self.optimizerV.step()

def update(self,rewards, states, actions):
    G = 0
    gamma = 0.99
    self.optimizerP.zero_grad()
    for i in reversed(range(len(rewards))):
        reward = rewards[i]
        state = torch.tensor(states[i].reshape(1, -1),
                             dtype=torch.float).to(device)
        action = torch.tensor(actions[i]).view(-1, 1).to(
            device)
        log_prob = torch.log(self.policy(state)).gather(1,
            action)
        G = gamma * G + reward
        advantage =  G  - self.vnetwork_local(state)
        loss = -log_prob * advantage
        loss.backward()
    self.optimizerP.step()
    del self.policy.episode_rewards[:]
    del self.policy.saved_actions[:]
    del self.policy.episode_states[:]
```

3.3 WANDB

Analysis and Tuning script for WANDB for REINFORCE with Baseline

```
def tune(hidden,lp,lv):
    reinforce = REINFORCE_MCB(hidden,lp,lv)
    regret, avg_reward = reinforce.train()
    return regret, avg_reward

def run_training():
     config_defaults = {
    "lp": 5e-4,
    "lv": 5e-4,
    "hidden": 64
}
config = wandb.init(config=config_defaults, project="
   withBaseline_acro2")
lp = config.config["lp"]
lv = config.config['lv']
hidden = config.config["hidden"]
regret, all_scores = tune(hidden,lp,lv)
for i in all_scores:
    wandb.log({'Scores':i})
wandb.log({"regret": regret})

sweep_config = {
    "method": "bayes",
    "metric": {"name": "regret", "goal": "minimize"},
    "parameters": {
    "lp": {"min": 0.001, "max": 0.1},
    "lv": {"min": 0.001, "max": 0.1},
    "hidden": {"values": [128, 256]}},

"project": "withbase_acro_optim",
"early_terminate": {
    "type": "hyperband",
    "min_iter": 3,
    "max_iter": 100}}

sweep_id = wandb.sweep(sweep_config)
wandb.agent(sweep_id, function=run_training)
```

3.4 Miscellaneous

Regret Calculation and Training

```
    def regret(self,avg_reward):
        regret = 0
        for i in range(len(avg_reward)):
            regret += self.env.spec.reward_threshold -
               avg_reward[i]
        return regret

    def train(self):
        total_reward = []
        avg_reward = []
        running_reward = 10
        # run infinitely many episodes
        for i_episode in range(self.episodes):
            state, _ = self.env.reset()
            ep_reward = 0
            for t in range(1, self.max_len):
            # select action from policy
                self.policy.episode_states.append(state)
                action = self.select_action(state)
                next_state, reward, done, _, _ = self.env.step(
                   action)
                self.learn_Value(state, action, reward,
                   next_state, done)
                self.policy.episode_rewards.append(reward)
                self.policy.saved_actions.append(action)
                state = next_state
                ep_reward += reward
                if done:
                    break
            if ep_reward > 500 or ep_reward < -500:
                ep_reward = 500*ep_reward/abs(ep_reward)
            total_reward.append(ep_reward)
            self.update(self.policy.episode_rewards, self.
               policy.episode_states, self.policy.saved_actions
               )
            running_reward = 0.05 * ep_reward + (1 - 0.05) *
               running_reward
            avg_reward.append(running_reward)
            if np.mean(avg_reward)>= self.env.spec.
               reward_threshold or i_episode >350:
                break
            regret = self.regret(avg_reward)
        return regret, avg_reward
```

Training and Experiment execution

```
reinforce_base = REINFORCE_mcb(hidden1,lp,lv)
rb, ab, tb = reinforce_base.train()

reinforce_without = REINFORCE_mcwb(256,lr)
rw, aw, tw = reinforce_without.train()

def PerformExpmt(num_expmt):
        reward_avgs = []
        for i in range(num_expmt):
            print("Experiment: %d"%(i+1))
            regret,rewards = reinforce_base.run(-200)
            reward_avgs.append(np.asarray(rewards))
        mean_base = np.mean(np.array(reward_avgs), axis=0)
        std_base = np.std(reward_avgs, axis=0)

        reward_avgs = []
        for i in range(num_expmt):
            print("Experiment: %d"%(i+1))
            regret,rewards = reinforce_without.run()
            reward_avgs.append(np.asarray(rewards))
        mean_wbase = np.mean(np.array(reward_avgs), axis=0)
        std_wbase = np.std(reward_avgs, axis=0)
        return mean_base, std_base, mean_wbase, std_wbase

mean_base, std_base, mean_wbase, std_wbase = PerformExpmt
    (5)

plt.figure(figsize=(8, 6))
plt.plot(range(352), mean_base, label='With Base', color='
    green')
plt.fill_between(range(352), mean_base - std_base,
    mean_base + std_base, alpha=0.3, color='green')
plt.plot(range(352), mean_wbase, label='Without Base',
    color='blue')
plt.fill_between(range(352), mean_wbase - std_wbase,
    mean_wbase + std_wbase, alpha=0.3, color='blue')
plt.xlabel('Episode')
plt.ylabel('Averaged Episodic Returns')
plt.title("Averaged Episodic Returns over 5 Eexperiments")
plt.legend()
plt.grid()
plt.savefig('withoutbaseline.png')
plt.show()
```

4 Dueling DQN

- **Policy:** Epsilon Greedy
- **Training Termination Condition** - **350 episodes** set up as the termination condition for training over average reward based conditions, this corresponded to a reward threshold around **200 for CartPole and -100 for Acrobot**. The result obtained are highly dependent on choice of these termination condition
- **Experiment Termination Condition** - **350 episodes** to ensure uniform size of all the reward arrays to calculate the mean and standard deviation. This number does not usually matter since convergence happens within the first 100 episodes, so this is a good termination condition over which we can benchmark results.
- Rewards are clamped between 500 and -500.
- **Discount factor** (γ) = 0.99

4.1 Acrobot-v1

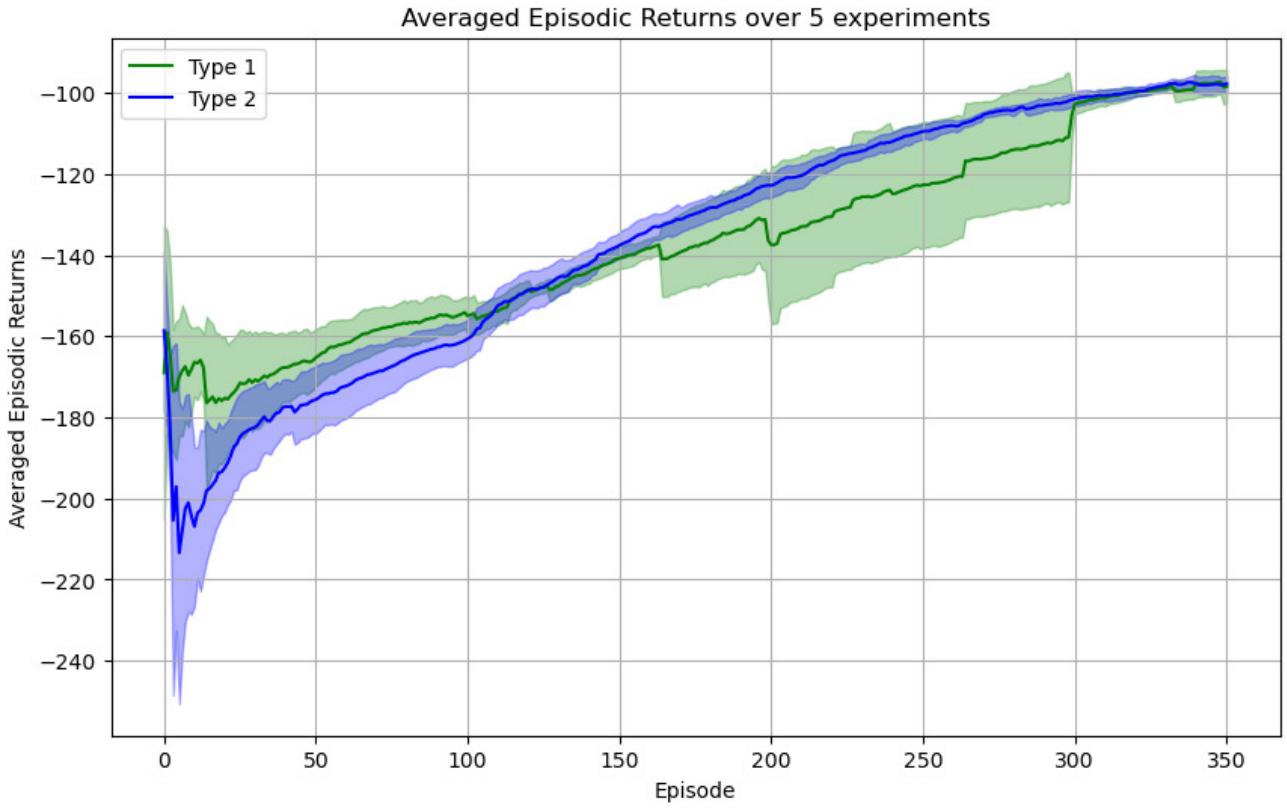


Figure 9: Average Returns vs Episodes

4.2 CartPole-v1

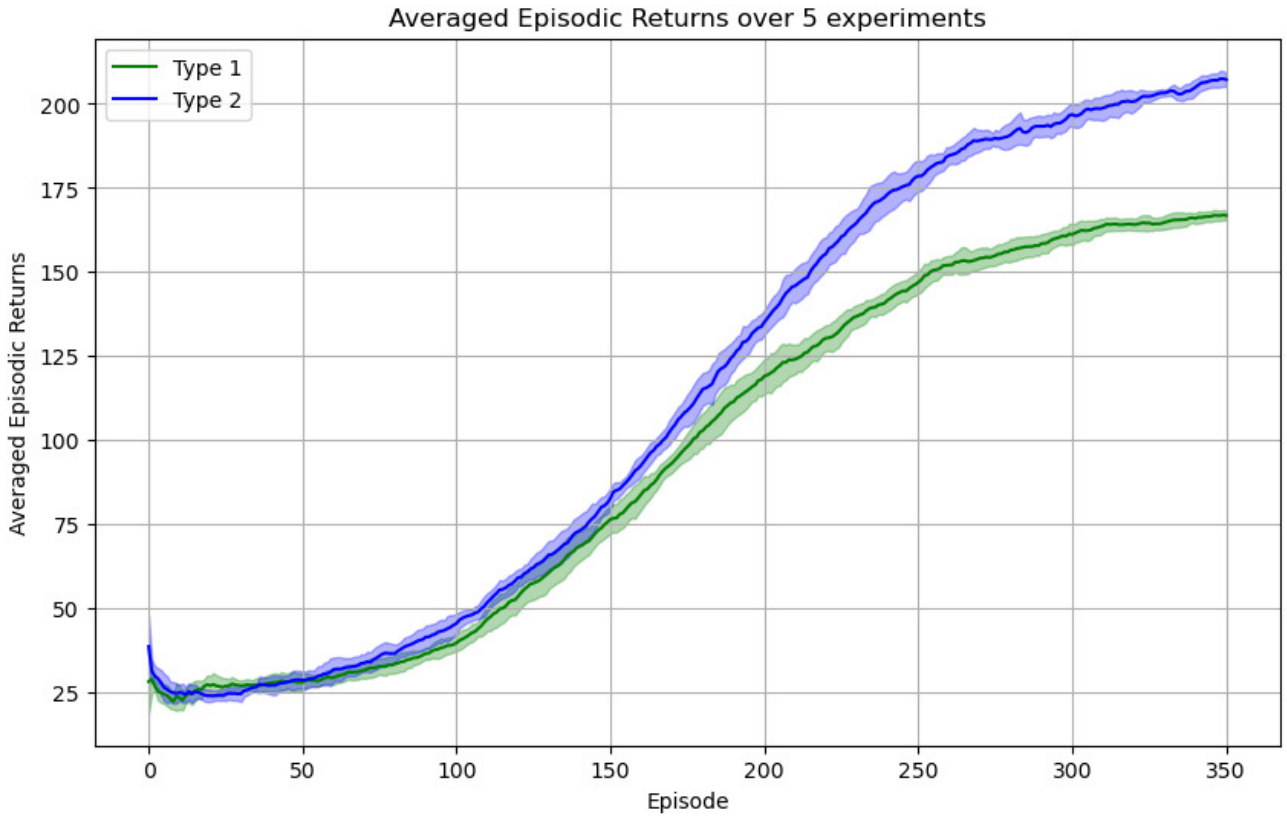


Figure 10: Average Returns vs Episodes

5

MC Reinforce

- **Training Termination Condition - 350 episodes** set up as the termination condition for training over average reward based conditions, this corresponded to a reward threshold around **200 for CartPole and -100 for Acrobot**. The result obtained are highly dependent on choice of these termination condition
- **Experiment Termination Condition - 350 episodes** to ensure uniform size of all the reward arrays to calculate the mean and standard deviation. This number does not usually matter since convergence happens within the first 100 episodes, so this is a good termination condition over which we can benchmark results.
- **Discount factor** $(\gamma) = 0.99$

5.1 Acrobot-v1

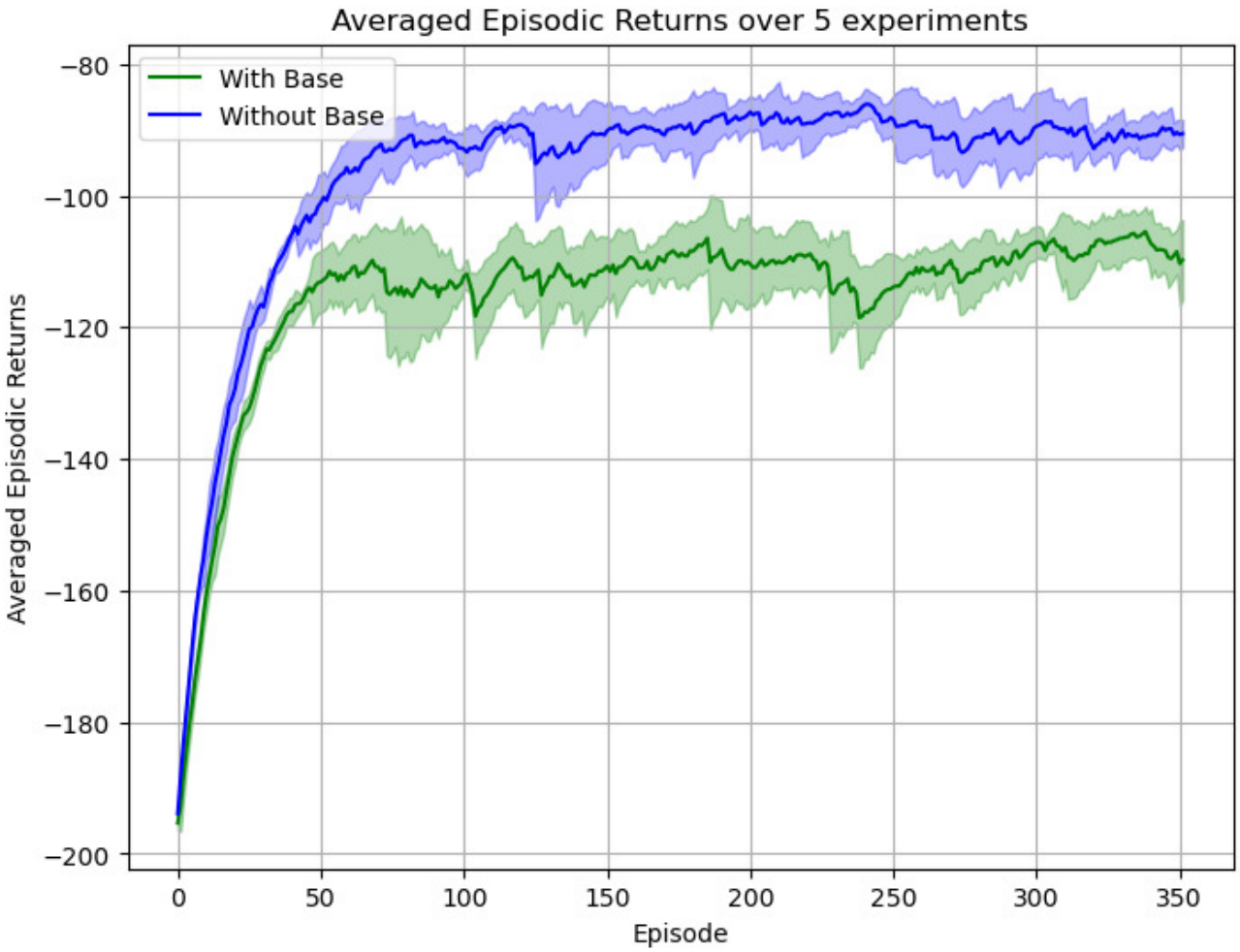


Figure 11: Average Returns vs Episodes

5.2 CartPole-v1

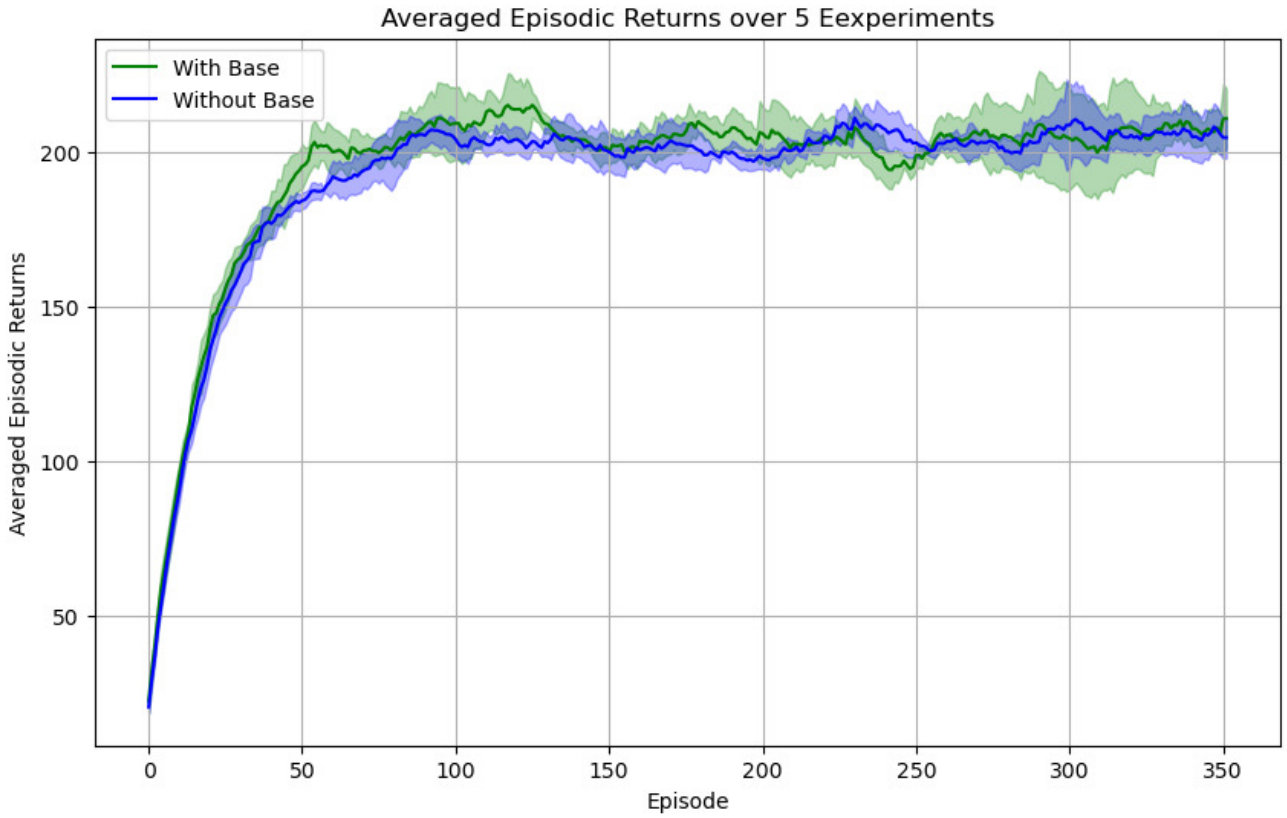


Figure 12: Average Returns vs Episodes

6 Inference and Conclusion

Before moving on to the Inferences, it is important to state the importance of the **Termination condition** for Training and the experiments. We adopted a episode based termination condition for both, using the tuned hyper-parameters for performing the experiment. This episode limit of 350, sometimes limits the reward growing up till 200, which can grow even higher, but we decided to adopt this threshold and optimize for minimum regret.

6.1 Dueling DQN

- **T**raining and tuning data from 1 and 3, training the dueling network for acrobot shows a very quick convergence in the case of Type 1 compared to Type 2, Type 1 plateaus much quicker compared to Type 2. Type 1 provides regularization by encouraging exploration, as actions with relatively high advantages are toned down, and actions with relatively low advantages are amplified. This prevents large positive or negative values from dominating the training process, potentially leading to more exploration and much smoother convergence.
- The max function in type2 results in more exploitative behaviour with faster convergence compared to type1. There is significant deviation between neighbouring hyper-parameter's curves, this can be attributed to sensitivity to small jumps due to the maximization bias.
- This is supported from Fig.4 and Fig.5, where the deviation between the curves is higher in type2 compared to Type1 with Type2 being about 1.5 times faster than type1 in training.
- Both types result in similar rewards, in the order of -100 for Acrobot and 190 for Cartpole with type1 lagging behind in terms of speed. There

6.2 MC Reinforce

- **D**uring Tuning and Training, it was observed that Acrobot without baseline almost never converged except for a narrow band of hyper-parameters which was veyr painful to narrow down to. Training data shows large variance during training in the case without baseline, supported by the fact that updates are directly based off of sampled returns and a smoother trajectory followed by the case with baseline.
- From the plots, it can be seen that baseline plateaus quicker than without-baseline in both cases. The difference in reward can be attributed to different hyper-parameters.
- For most hyper-parameters, acrobot without baseline couldnt complete an episode resulting in large delays (thanks to termination based on number of time steps) affecting training due to Monte Carlo Nature.