Anish Mulay:ED21B011
Shiv Patil:ED21B059

CS6700 : Reinforcement Learning

Programming Assignment 2

Dueling-DQN and Monte-Carlo REINFORCE Cartpole and Acrobat Implementations

You can find the code for this project at: RL PA2

Dueling DQN

Acrobot Environment

Definitions

- **Type 1 DQN**:
$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \frac{1}{|A|}\sum_{a' \in \mathcal{A}} A(s,a';\theta)\right) \tag{1}$$

- **Type 2 DQN** :
$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \max_{a' \in \mathcal{A}} A(s,a';\theta)\right) \tag{2}$$

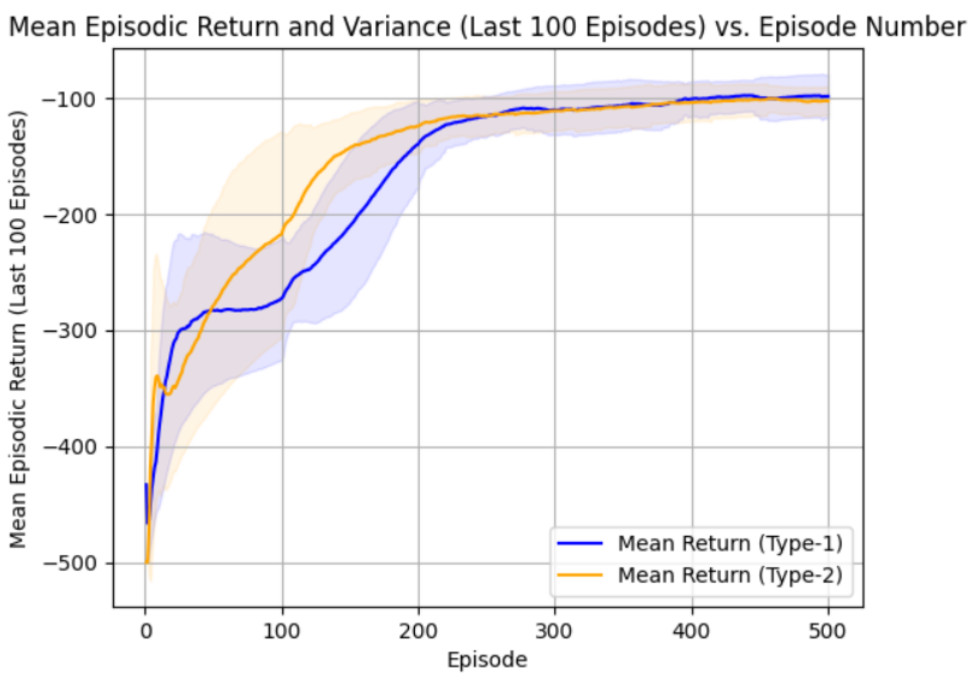


Figure 1: Moving average reward curve

Observations

- Variance for Type1-DQN is significantly higher than Type2-DQN.
- The number of episodes taken to reach the threshold is comparatively lesser for Type2- DQN
- The learning curve for Type2-DQN is smoother.

Inferences

- By focusing on the maximum advantage value, Type-2 prioritizes the most advantageous action more strongly. This could lead to more consistent and reliable improvements in the agent's performance, resulting in a smoother reward curve.
- The Q value in this case is obtained by subtracting the maximum advantage value, and action is learned according to it. It could lead to lower variance
- The number of episodes to reach threshold is typically lesser for maximum Advantage Q function as emphasizing on the most advantageous action, could lead to more effective action selection during training and hence lesser number of episodes.
- It can be observed that max advantage Q function performs nearly as good and nearly better than the average advantage function in case of environments with higher action space.
- This can probably be because of the fact that average advantage Q function considers average of the action advantages thus nullifying the effect of one action being favoured over the other potentially leading to suboptimal learning outcomes.

Code Implementations

- Two separate forward functions for the two types have been defined

```python
def forward(self, state):
    y = self.relu(self.fc1(state))
    value = self.relu(self.fc_value(y))
    adv = self.relu(self.fc_adv(y))

    value = self.value(value)
    adv = self.adv(adv)

    advAverage = torch.mean(adv, dim=1, keepdim=True)
    Q = value + adv - advAverage

    return Q
```

Figure 2: Forward function for Type 1

```python
def forward_2(self, state):
    y = self.relu(self.fc1(state))
    value = self.relu(self.fc_value(y))
    adv = self.relu(self.fc_adv(y))

    value = self.value(value)
    adv = self.adv(adv)

    advMax = torch.max(adv, dim=1, keepdim=True).values
    Q = value + adv - advMax

    return Q
```

Figure 3: Forward function for Type 2

```python
class Memory(object):
    def __init__(self, memory_size: int) -> None:
        self.memory_size = memory_size
        self.buffer = deque(maxlen=self.memory_size)

    def add(self, experience) -> None:
        self.buffer.append(experience)

    def size(self):
        return len(self.buffer)

    def sample(self, batch_size: int, continuous: bool = True):
        if batch_size > len(self.buffer):
            batch_size = len(self.buffer)
        if continuous:
            rand = random.randint(0, len(self.buffer) - batch_size)
            return [self.buffer[i] for i in range(rand, rand + batch_size)]
        else:
            indexes = np.random.choice(np.arange(len(self.buffer)), size=batch_size, replace=False)
            return [self.buffer[i] for i in indexes]

    def clear(self):
        self.buffer.clear()
```

Figure 4: Memory Class for Replay Buffer

```python
def objective(trial,environment):
    # Define the search space
    params = {

        'initial_epsilon': trial.suggest_loguniform('initial_epsilon', 0.1, 0.2),
        'lr': trial.suggest_loguniform('lr', 1e-5, 1e-4),
        'batch_size': trial.suggest_categorical('batch_size', [32, 64, 128]),
        'replay_size':trial.suggest_categorical('replay_size', [50000,75000,100000]),

    }
    # Train the model with the given hyperparameters
    seed=1
    episode_rewards = train(seed,params,environment=environment)
    # Return the mean episode reward as the objective value
    return np.mean(episode_rewards)
```

Figure 5: Objective function that is maximised during tuning

```
done = False
while not done:
    p = random.random()
    if p < epsilon:
        action = random.randint(0, 1)
    else:
        tensor_state = torch.FloatTensor(state).unsqueeze(0).to(device)
        action = onlineQNetwork.select_action(tensor_state)
    next_state, reward, done, _ = env.step(action)
    episode_reward += reward
    memory_replay.add((state, next_state, action, reward, done))

    # Checks if the replay buffer has enough samples to sample from
    if memory_replay.size() > 128:
        if not begin_learn:
            print('learn begin!')
            begin_learn = True
        learn_steps += 1
        if learn_steps % UPDATE_STEPS == 0:
            targetQNetwork.load_state_dict(onlineQNetwork.state_dict())
    # Sampling batch size number of samples for target network
        batch = memory_replay.sample(BATCH, False)
        batch_state, batch_next_state, batch_action, batch_reward, batch_done = zip(*batch)

        batch_state = torch.FloatTensor(np.array(batch_state)).to(device)
        batch_next_state = torch.FloatTensor(np.array(batch_next_state)).to(device)
        batch_action = torch.FloatTensor(np.array(batch_action)).unsqueeze(1).to(device)
        batch_reward = torch.FloatTensor(np.array(batch_reward)).unsqueeze(1).to(device)
        batch_done = torch.FloatTensor(np.array(batch_done)).unsqueeze(1).to(device)

        with torch.no_grad():
            onlineQ_next = onlineQNetwork(batch_next_state)
            targetQ_next = targetQNetwork(batch_next_state)
            online_max_action = torch.argmax(onlineQ_next, dim=1, keepdim=True)
            y = batch_reward + (1 - batch_done) * GAMMA * targetQ_next.gather(1, online_max_action.long())
    # Loss Function
        loss = F.mse_loss(onlineQNetwork(batch_state).gather(1, batch_action.long()), y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    #  Decaying Epsilon
        if epsilon > FINAL_EPSILON:
            epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / EXPLORE

    state = next_state
episode_rewards.append(episode_reward)
# Checking the threshold condition to stop training
running_reward = 0.05 * episode_reward + (1 - 0.05)*running_reward
if running_reward> env.spec.reward_threshold:
    print("Environmnt Solved. Running Reward is now {}".format(running_reward))
    break
```

Figure 6: Inner loop of training function

Hyper-parameter Tuning

- Each experiment during an optimiser trials been run for a total of 500 episodes and the training is stopped when the reward threshold of -100 is crossed.
- For the final plot when each type is run for 5 seeds, the number of episodes per seed have been kept constant at 500.
- Hyper-parameters that have been tuned are initial epsilon,learning rate,batch size and replay size
- Discount factor γ has been kept constant at 0.99
- Optuna library has been used to maximise the mean rewards for 3 trials, the best of which is chosen
- Following are the tuned hyper-parameter values for Type-1 and Type-2:

```
initial_epsilon: 0.12123082109466547
lr: 5.454636296829481e-05
batch_size: 32
replay_size: 100000
```

Figure 7: Hyper-parameters for Type 1

```
initial_epsilon: 0.12696279595697205
lr: 8.97633979164655e-05
batch_size: 128
replay_size: 50000
```

Figure 8: Hyper-parameters for Type 2

Cartpole Environment

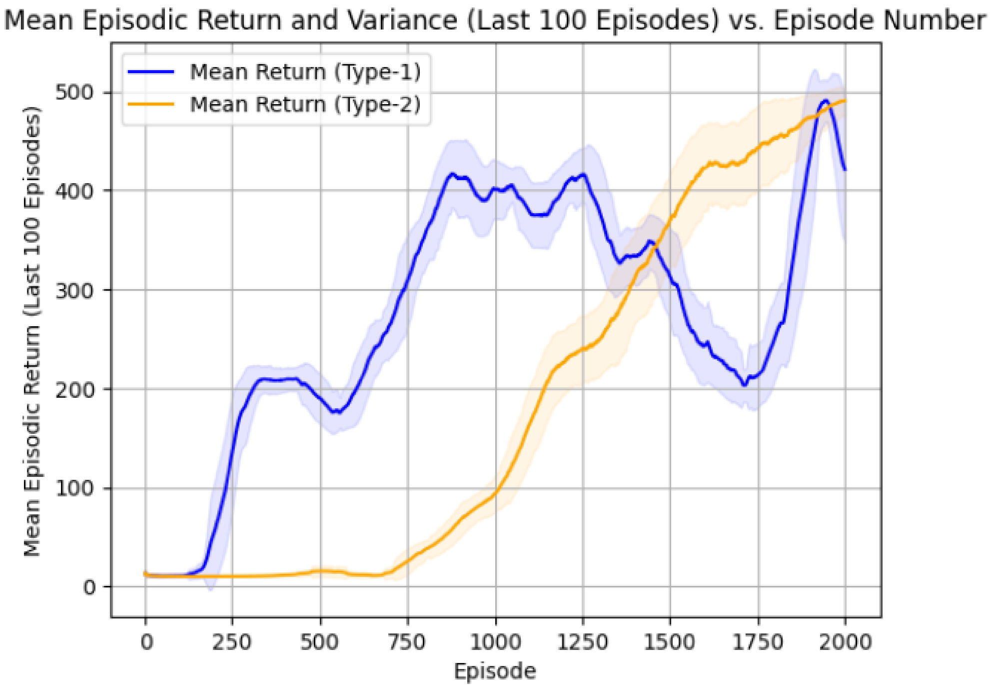


Figure 9: Moving Reward Curve for last 100 episodes

Observations

- Variance for Type1-DQN is comparatively higher than Type2-DQN.
- The number of episodes taken to reach the threshold is significanly lesser for Type1- DQN
- The learning curve for Type2-DQN shows a consistent but slow rise.
- Type-1 DQN shows a faster learning curve as the reward values increase faster in lesser number of episodes.
- A drop in the reward values can be seen for Type-1 around 1000 episode mark.This can be accounted to the environment stochasticity where the agent might have needed a larger number of episodes to train in one of the seeds. The agent relearns after that drop and the rewards increase after that.

Inferences

- By focusing on the maximum advantage value, Type-2 prioritizes the most advantageous action more strongly. This could lead to more consistent and reliable improvements in the agent's performance, resulting in a smoother reward curve.
- The Q value in this case is obtained by subtracting the maximum advantage value, and action is learned according to it. It could lead to lower variance which is similar to the previous case.

- The number of episodes to reach threshold is significantly lesser for average Advantage Q function which is also consistent with the optimal learning rate being higher.
- The average advantage Q function likely allows Type-1 DQN to maintain a balanced exploration-exploitation tradeoff throughout the learning process.
- Overall the average advantage Q-function performs better than max Advantage Q funtion in smaller state space environments.

Code Implementations

- The code implemetations are typically the same for both the environments, the only difference being the neural network dimensions.
- Cartpole has 4 possible states and 2 possible acions, whereas Acrobat has 6 possible states and 3 possible actions.

```
class QNetwork(nn.Module):
    def __init__(self):
        super(QNetwork, self).__init__()

        self.fc1 = nn.Linear(4, 64)
        self.relu = nn.ReLU()
        self.fc_value = nn.Linear(64, 256)
        self.fc_adv = nn.Linear(64, 256)

        self.value = nn.Linear(256, 1)
        self.adv = nn.Linear(256, 2)
```

Figure 10: Neural Network Cartpole

```
class QNetwork(nn.Module):
    def __init__(self):
        super(QNetwork, self).__init__()

        self.fc1 = nn.Linear(6, 64)
        self.relu = nn.ReLU()
        self.fc_value = nn.Linear(64, 256)
        self.fc_adv = nn.Linear(64, 256)

        self.value = nn.Linear(256, 1)
        self.adv = nn.Linear(256, 3)
```

Figure 11: Neural Network Acrobat

Hyper-parameter Tuning

- Each experiment during an optimiser trials been run for a total of 2000 episodes and the training is stopped when the reward threshold of 475 is crossed.

- For the final plot when each type is run for 5 seeds, the number of episodes per seed have been kept constant at 2000.
- Hyper-parameters that have been tuned are initial epsilon,learning rate,batch size and replay size
- Discount factor γ has been kept constant at 0.99
- Optuna library has been used to maximise the mean rewards for 3 trials, the best of which is chosen
- The optimal learning rate is higher for Type-1 DQN
- The optimal batch size and replay size are also larger for Type-1 DQN
- Following are the tuned hyper-parameter values for Type-1 and Type-2 Cartpole environments:

```
Params:
  initial_epsilon: 0.16758510311242736
  lr: 8.052945322072001e-05
  batch_size: 128
  replay_size: 75000
```

Figure 12: Hyper-parameters for Type 1

```
Params:
  initial_epsilon: 0.1374091586543872
  lr: 3.077690786239081e-05
  batch_size: 32
  replay_size: 50000
```

Figure 13: Hyper-parameters for Type 2

Episodic Returns for Cartpole and Acrobat DQN have been shown on the next page. These would give a better picture in comparing the variances.

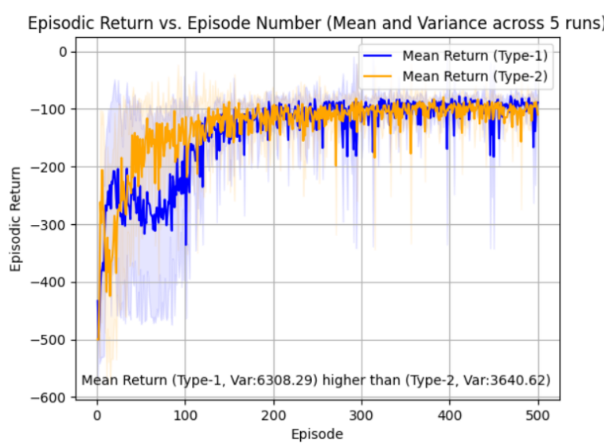


Figure 14: Episodic reward curve

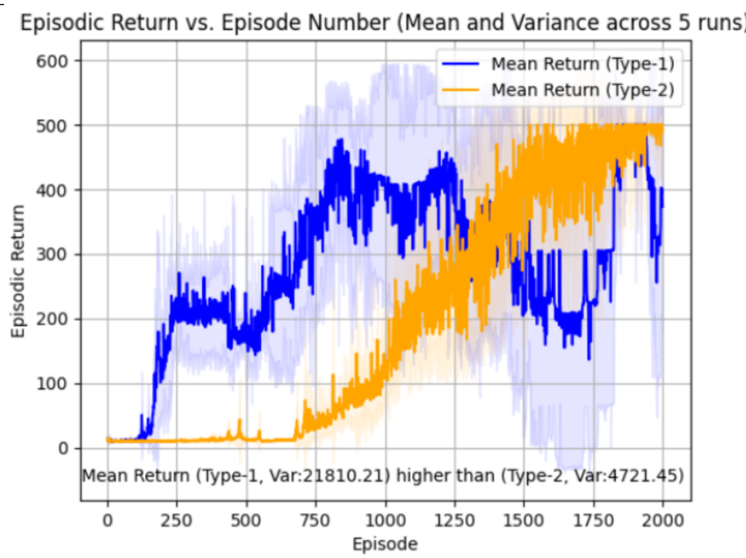


Figure 15: Episodic reward curve

Reinforce

Definitions

- **Without Baseline**:

$$\theta = \theta + \alpha G_t \nabla \pi(A_t|S_t, \theta)/\pi(A_t|S_t, \theta)$$

(3)

- **With Baseline** :

$$\theta = \theta + \alpha(G_t - V(S_t; \Phi))\nabla \pi(A_t|S_t, \theta)/\pi(A_t|S_t, \theta)$$

(4)

Code Implementations have been shown on the next page. The structure is the same for both the environments as in case of Dueling DQN.

```python
class Policy(nn.Module):
    """Policy for actor and critic
    """

    def __init__(self, state_size= env.observation_space.shape[0], action_size=env.action_space.n, hidden_size=128, dropout = 0.5):
        super(Policy, self).__init__()
        self.dropout = nn.Dropout(dropout)
        self.affine1 = nn.Linear(state_size, hidden_size)

        #actor's layer
        self.action_head = nn.Linear(hidden_size, action_size)

        # action & reward buffer
        self.saved_actions = []
        self.rewards = []


    def forward(self, state):
        x = self.affine1(state)
        # x = self.dropout(x)
        x = F.relu(x)
        action_prob = self.action_head(x)

        return action_prob
```

Figure 16: Network without baseline

```python
SavedAction = namedtuple('SavedAction', ['log_prob', 'value'])
class PolicyForBaseline(nn.Module):
    """
    Policy for actor and crtitic
    """

    def __init__(self, state_size = env.observation_space.shape[0], action_size = env.action_space.n, hidden_size  = 128, dropout = 0.5):
        super(PolicyForBaseline, self).__init__()
        self.dropout = nn.Dropout(dropout)

        #base layer
        self.affine1 = nn.Linear(state_size, hidden_size)

        #actor's layer
        self.action_head = nn.Linear(hidden_size, action_size)


        #critic's layer
        self.value_head = nn.Linear(hidden_size,1)

        # action & reward buffer
        self.saved_actions = []
        self.rewards = []

    def forward(self, state):
        x = self.affine1(state)
        x = F.relu(x)

        action_prob = self.action_head(x)

        state_values = self.value_head(x)

        return action_prob, state_values

    def select_action(self, state):
        state = torch.tensor(state).to(device)
        action_probs, state_values = self.forward(state)   #Forward pass for the current state
        action_probs = F.softmax(action_probs, dim=-1).to(device)

        #sampling action
        dist = Categorical(action_probs)
        action = dist.sample()

        self.saved_actions.append(SavedAction(dist.log_prob(action), state_values))

        return action.item(), dist.log_prob(action)
```
0.0s

Figure 17: Network with baseline

```python
class Reinforce():
    def __init__(self, env, policy, optimizer,baseline: bool, break_at_threshold = False, seed = None, gamma = GAMMA, max_t = TIMESTEPS, print_every = PRINT_EVERY, reward_threshold = REWARD_THRESHOLD):
        self.env = env
        self.policy = policy
        self.optimizer = optimizer
        self.gamma = gamma
        self.reward_threshold = reward_threshold
        self.baseline = baseline
        self.seed = seed
        self.break_at_threshold = break_at_threshold

    def train(self,episode, max_t = 500 ):
        self.policy.train()
        saved_log_probs = []
        rewards = []
        done = False
        episode_reward = 0

        # Use the seed if provided
        state = self.env.reset(seed = self.seed)

        # state = self.env.reset()
        state = state[0]
        next_state = state

        for t in range(max_t):
            current_state = next_state

            #Sample action for current state
            action, log_prob = self.policy.select_action(current_state)
            saved_log_probs.append(log_prob)

            #Get next state, and the reward
            next_state, reward, done, _, _ = self.env.step(action)

            rewards.append(reward)
            self.policy.rewards.append(reward)

            episode_reward += reward
            if done:
                break

        #Calculate returns for the episode
        returns = self.calculate_returns(rewards, episode)

        #Optimise policy and value losses
        if self.baseline:
            self.optimize_policy_with_baseline(returns)
        else:
            self.optimise_policy_without_baseline(returns)

        return episode_reward
```

Figure 18: Training Class

```python
    def calculate_returns(self, rewards, episode, normalize = True):
        returns =[]
        R = 0
        for r in self.policy.rewards[::-1]:

            #TD(0) update step
            R = r + self.gamma*R
            returns.insert(0,R)
        returns = torch.tensor(returns).float()

        if normalize:
            returns = (returns - returns.mean()) / (returns.std() + episode)
        return returns
```

Figure 19: Return Calculation

```python
def optimize_policy_with_baseline(self, returns):
    policy_losses = []
    value_losses = []
    saved_actions = self.policy.saved_actions

    for (log_prob, value), R in zip(saved_actions, returns):
        #TD error
        delta = R - value.item()

        #calculate actor(policy) loss
        policy_losses.append(-log_prob*delta.item())

        #calculate critic(value) loss
        value_losses.append(F.smooth_l1_loss(value, torch.tensor([R]).to(device)))

    #Calculating gradients and backpropagation
    torch.autograd.set_detect_anomaly(True)
    self.optimizer.zero_grad()
    loss = torch.stack(policy_losses).sum() + torch.stack(value_losses).sum()
    loss.backward(retain_graph=True)
    self.optimizer.step()

    # reset rewards and action buffer
    del self.policy.rewards[:]
    del self.policy.saved_actions[:]

def optimise_policy_without_baseline(self, returns):

    policy_losses = []
    saved_actions = self.policy.saved_actions

    for log_prob, R in zip(saved_actions, returns):
        policy_losses.append(-log_prob*R)

    if policy_losses:
        policy_loss = torch.stack(policy_losses).sum()
    else:
        policy_loss = torch.tensor(0.0)

    self.optimizer.zero_grad()
    policy_loss.backward()
    self.optimizer.step()

    # reset rewards and action buffer
    del self.policy.rewards[:]
    del self.policy.saved_actions[:]
```

Figure 20: Optimising Losses

```python
#HYPERPARAMETER TUNING FOR LEARNING RATE  - WITHOUT BASELINE

from bohb import BOHB
import bohb.configspace as cs

SEED = 123

if __name__ == '__main__':
    np.random.seed(SEED)

    #defining objective function
    def evaluate(params,budget):


        print("budget: ", budget)

        policy = Policy().to(device)
        optimizer = torch.optim.Adam(policy.parameters(), lr=params['lr'])
        agent = Reinforce(env, policy, optimizer, baseline=False, seed=1234, break_at_threshold=True)
        episode_rewards, _ = agent.episode_run(n_episodes=budget, print_every=100)

        #returning negative rewards as loss and minimizing the loss (or maximizing the mean rewards for all episodes)
        return -np.mean(episode_rewards)

    learning_rate = cs.UniformHyperparameter('lr', 1e-4, 1e-1, log=True)
    configspace = cs.ConfigurationSpace([learning_rate])

    print(f"Learning rate: {learning_rate}")

    opt = BOHB(configspace, evaluate, max_budget=2000, min_budget=1, n_proc=1)



    logs = opt.optimize()
    print(logs)
```

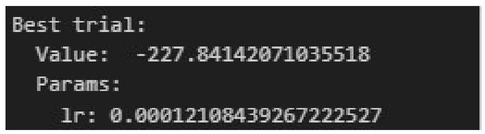
Figure 21: Hyperparameter tuning

Hyperparameter tuning

- Each experiment during an optimiser trials been run for a total of 2000 episodes and the training is stopped when the reward threshold of -100 is crossed.
- For the final plot when each type is run for 5 seeds, the number of episodes per seed have been kept constant at 2000.
- Hyper-parameter that have been tuned is learning rate.
- Discount factor γ has been kept constant at 0.99
- BOHB (Bayesian Optimisation HyperBand) method is used to learn the hyperparameters. It combines both Bayesian as well HyperBand approach for tuning in an efficient way.
- BOHB is a multi-fidelity optimisation method which uses a budget for performing optimisation. In our code, budget refers to number of episodes running in each iteration. We have used a min and max budget of 100 and 2000 respectively, while tuning. Refer fig-21.
- The optimal learning rate is higher for baseline case
- Following are the tuned hyper-parameter values for Type-1 and Type-2 Acrobat environments:

```
Best Hyperparameter Configuration:
Budget: 222.0
Loss: 189.52941176470588
Configuration:
    Name: lr | Value: 0.010575225442685743
```

Figure 22: Hyper-parameters for Type 1(With Baseline)-Acrobat



```
Best trial:
  Value:  -227.84142071035518
  Params:
    lr: 0.00012108439267222527
```

Figure 23: Hyper-parameters for Type2 (Without Baseline)-Acrobat

Acrobot

Observations

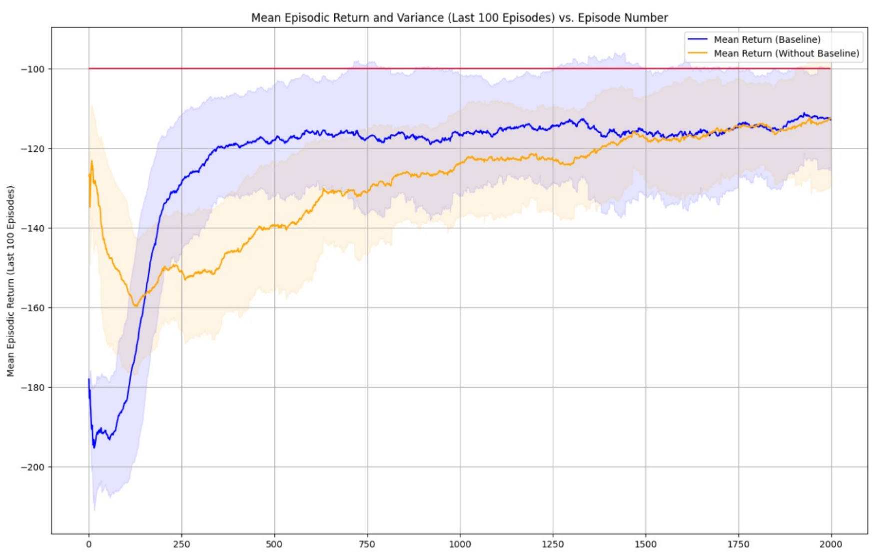


Figure 24: Rewards vs Episodes Plot- Acrobot - Reinforce With/Without Baseline

- Variance for Type2-(Without Baseline) is significantly higher than Type1-(With Baseline).
- The number of episodes taken to reach the threshold (-100 rewards) is significantly lesser for Type1 (Baseline) Model which is approx. 750 episodes as compared to approx. 1500 episodes for Type2 (Without Baseline).

- The learning curve for Type2 - Without Baseline shows a consistent but slow rise.
- Type-1-Baseline shows a faster learning curve as the reward values increase faster in lesser number of episodes.
- The overshoot in rewards at the beginning observed in without baseline case is due to stochasticity taken into account while running 5 experiments.

Inferences

- The higher Variance for Type2-(Without Baseline) as compared to Type1-(With Baseline) suggests that it is more inconsistent during learning.
- This is because the baseline provides a stable reference point, which reduces the variability and providing an unbiased estimate as compared to not having a baseline.
- For problems with large state spaces like Acrobat, the variance becomes unacceptably high.
- Variance is related to the "recurrence time" or the episode length which is thus higher in case of without baseline.
- The initial overshoot indicates that training without baseline might not be reliable as the stability gets affected.
- Faster learning process in with baseline variant is also consistent with the optimally tuned learning rate being higher in that case.

Cartpole

Figure 25: Cartpole- Hyperparameters with baseline

Figure 26: Cartpole- Hyperparameters without baseline

Observations

- Variance for Type2-(Without Baseline) is comparatively higher than Type1-(With Baseline).
- The number of episodes taken to reach the threshold (475 rewards) is significantly lesser for Type1 (Baseline) Model which is approx. 1600 episodes as compared to approx. 2000+ episodes for Type2 (Without Baseline).

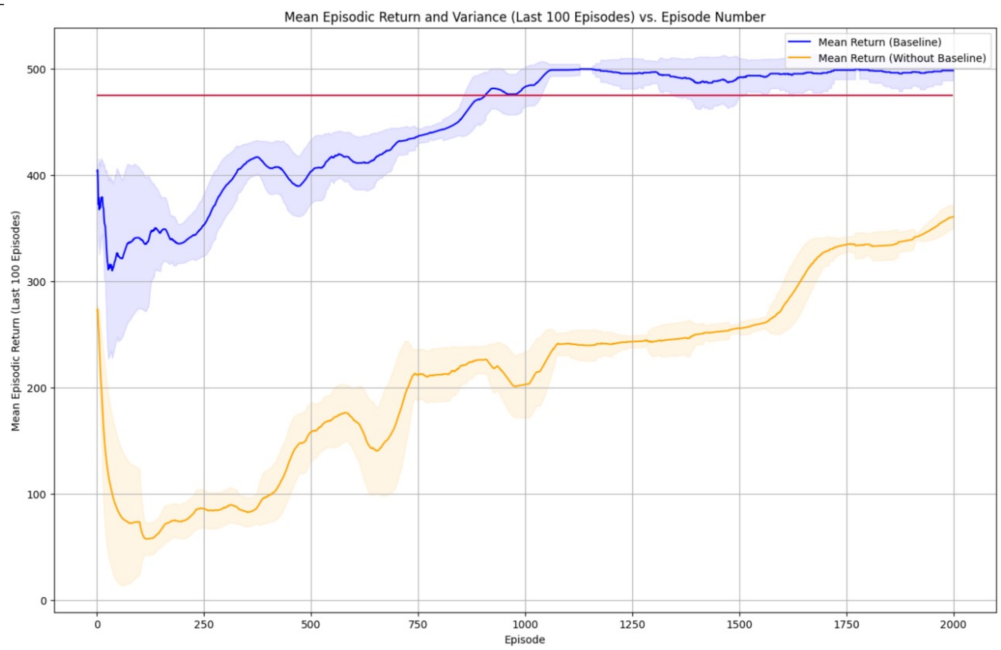


Figure 27: Rewards vs Episodes Plot- Cartpole - Reinforce With/Without Baseline

- The learning curve for Type2 - Without Baseline shows a consistent but slow rise.
- Type-1-Baseline shows a faster learning curve as the reward values increase faster in lesser number of episodes and crosses the threshold.
- Their is an initial overshoot for without baseline case which is followed by a deep trough. The observed drop is significantly smaller for with baseline case
- Variance progressively decreases in Cartpole with baseline as compared to the without baseline case where it remains high throughout.

Inferences

- The higher Variance for Type2-(Without Baseline) as compared to Type1-(With Baseline) suggests that it is more inconsistent during learning.
- This is because the baseline provides a stable reference point, which reduces the variability and providing an unbiased estimate as compared to not having a baseline.
- The variance for without baseline curve is not as high as the previous Acrobat case as the action space is smaller.
- Variance is related to the "recurrence time" or the episode length which is thus higher in case of without baseline.
- The initial overshoot followed by the drop indicates that training without baseline might not be reliable as the stability gets affected.
- Faster learning process in with baseline variant is also consistent with the optimally tuned learning rate being higher in that case.

Episodic Returns for Cartpole and Acrobat DQN have been shown on the next page. These would give a better picture in comparing the variances.

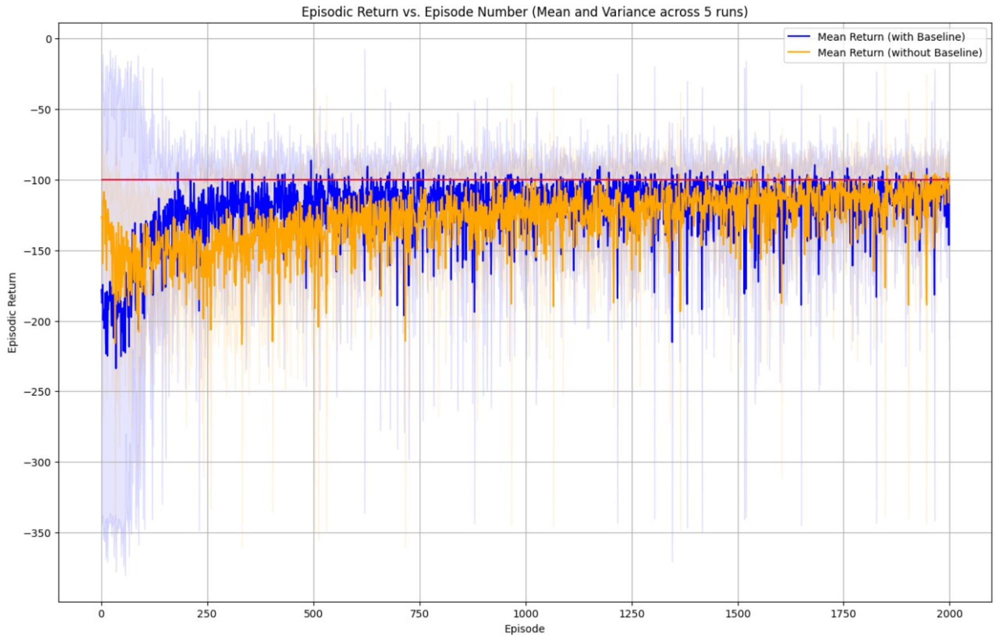


Figure 28: Episodic rewards plot for Acrobat

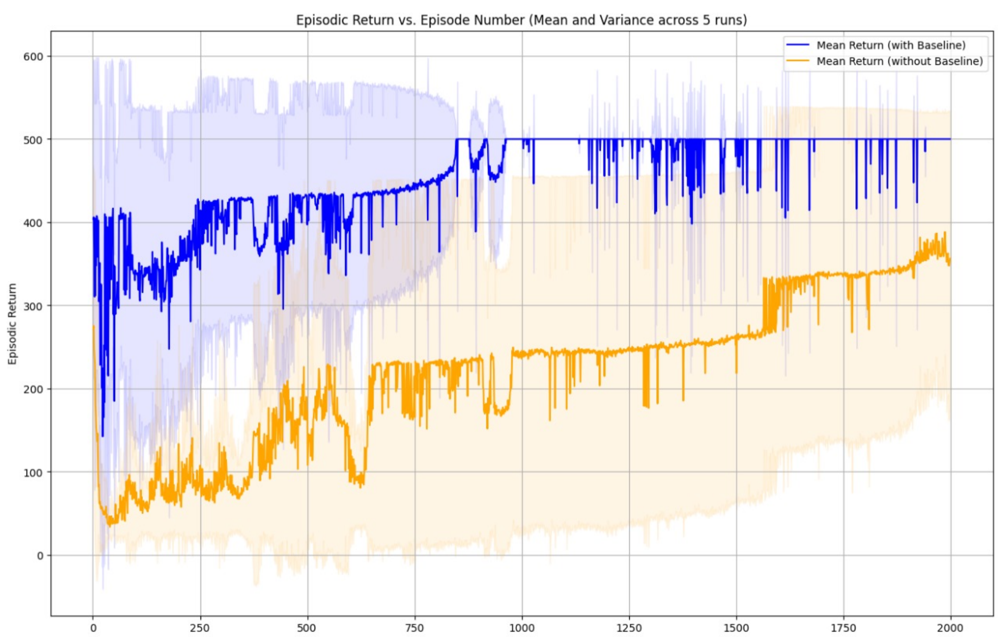


Figure 29: Episodic rewards plot for Carpole