CS6700
REINFORCEMENT LEARNING
PROGRAMMING ASSIGNMENT - 2
TEAM: Anirud N (CE21B014) , Leon Davis M S (ED21B038)

GITHUB REPOSITORY LINK FOR THE CODE:
(https://github.com/AnirudN/CS6700_CE21B014_ED21B038_PA2/tree/main)(Click here)

Objective:
This Assignment explores training two variants of each Dueling-DQN and Monte-Carlo REIN FORCE and assessing their comparative performance. This involves tuning of hyperparameters and plotting for Comparison.

Environment:
Acrobot-v1: The system consists of two links connected linearly to form a chain, with one end of the chain fixed. The joint between the two links is actuated. The goal is to apply torques on the actuated joint to swing the free end of the linear chain above a given height while starting from the initial state of hanging downwards.
CartPole-v1: A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

Algorithms :
DUELLING DQN :
Dueling-DQN (Cartpole)-

Code:

```
class QNetwork1(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_units=64, fc_value_units=256, fc_adv_units=256):
        super(QNetwork1, 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, action_size)

    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
```

The network takes a state tensor as input and processes it through layers to estimate both the value and advantage functions separately. These estimations are then combined to produce the Q-values for each action. By decoupling the value and advantage estimations, the dueling DQN architecture enhances the stability and efficiency of the learning process.

The forward pass has **2 variants** of the update equations:
1)

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

Implemented as-

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

return Q
```

2)

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \max_{a' \in |\mathcal{A}|} A(s,a';\theta) \right) \qquad \text{(Type-2)}$$

Implemented as-

```
max_adv = torch.max(adv, dim=1, keepdim=True)[0]
Q = value + adv - max_adv

return Q
```

```python
class ReplayBuffer:


    def __init__(self, action_size, buffer_size, batch_size, seed):

        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

The **ReplayBuffer** class serves as a container for storing experience tuples used in training reinforcement learning agents. These tuples encapsulate information about an agent's interactions with the environment, including the state observed, action taken, resulting reward, next state transitioned to, and whether the episode terminated after that transition. Key methods include add, which inserts new experiences into the buffer, sample, which randomly selects a batch of experiences for training, and __len__, which returns the current size of the buffer. By storing and randomly sampling experiences, the replay buffer facilitates more efficient and stable learning by breaking correlations between consecutive experiences and promoting more effective use of past experiences during training.

```python
def learn(self, experiences, gamma):
    """Perform a training iteration on a batch of data."""
    states, actions, rewards, next_states, dones = experiences

    # Get max predicted Q values (for next states) from target model
    Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

    # Compute Q targets for current states
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # Get expected Q values from local model
    Q_expected = self.qnetwork_local(states).gather(1, actions)

    # Compute loss
    loss = F.mse_loss(Q_expected, Q_targets)

    # Minimize the loss
    self.optimizer.zero_grad()
    loss.backward()

    # Gradient Clipping
    for param in self.qnetwork_local.parameters():
        param.grad.data.clamp_(-1, 1)

    self.optimizer.step()

    return loss.item()  # Convert loss to a scalar value for logging purposes
```

In the **learn** method, the agent performs a training iteration on a batch of experiences to update its Q-network parameters. It calculates the Q targets for the current states using the Bellman equation, incorporating rewards and estimated future rewards. Then, it computes the expected Q-values for the current states based on the actions taken. The method calculates the loss between the expected Q-values and the Q targets using mean squared error (MSE). After computing the loss, it performs backpropagation to update the Q-network's parameters, ensuring that the network approximates the Q-values more accurately. Additionally, gradient clipping is applied to stabilize training, and the method returns the loss as a scalar value for logging purposes.

```python
def dqn(agent,n_episodes=500, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):

    scores_window = deque(maxlen=100)
    ''' last 100 scores for checking if the avg is more than 195 '''

    eps = eps_start
    regret = 0
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break

        regret += 475-score
        scores_window.append(score)

        eps = max(eps_end, eps_decay*eps)
        ''' decrease epsilon '''

        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)), end="")

        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
        if np.mean(scores_window)>=195.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
            break
    return regret
```

If the environment is solved (average score >= 475), the function prints a message and exits the loop. The function also returns the total regret over the training period. Regret represents the difference between the maximum possible score (475 in this case) and the cumulative reward obtained during training, aiming to minimize this difference as the agent learns.

Optimization:

```python
import wandb

def main():
    wandb.init(project="DQN_avg_cart")
    agent = TutorialAgent(state_size=state_shape,action_size = action_shape,seed = 0, config=wandb.config)
    score = average_return_over_5_runs(agent)
    wandb.log({"score": score})

# 2: Define the search space
sweep_configuration = {
    "method": "bayes",
    "metric": {"goal": "minimize", "name": "score"},
    "parameters": {
        "BUFFER_SIZE": {"values": [1e5,1e6,1e4]},
        "BATCH_SIZE": {"max": 256, "min": 32},
        "GAMMA": {"values": [0.99]},
        "LR": {"max": 1e-3, "min": 1e-5},
        "UPDATE_EVERY": {"max": 100, "min": 10},
    },
}

# 3: Start the sweep
sweep_id = wandb.sweep(sweep=sweep_configuration, project="DQN_avg_cart")

wandb.agent(sweep_id, function=main, count=10)
```

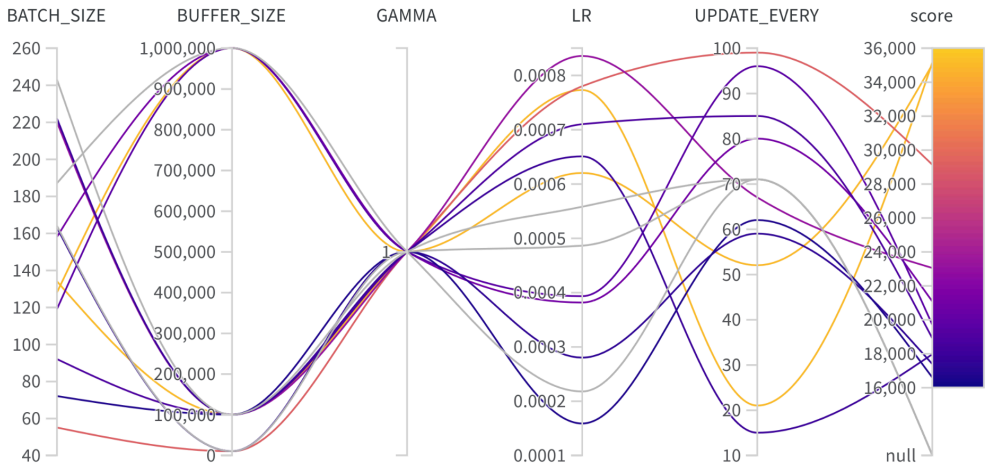
✓ 93m 41.3s

Hyperparameter tuning:

Wandb (https://wandb.ai/site) was used to visualize and perform the hyperparameter testing. The hyperparameters are

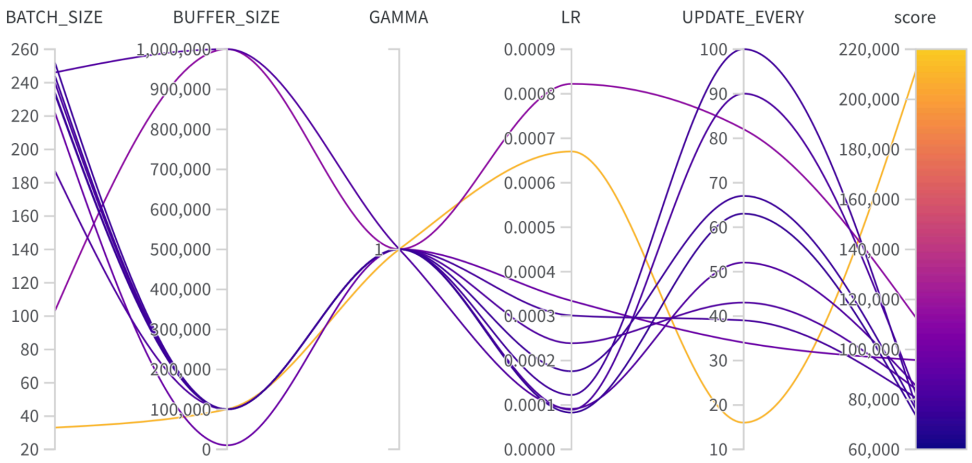
1) Learning Rate
2) Batch size
3) Buffer size
4) Gamma
5) Update every

Dueling DQN (Type-1):



LR = 0.00015
BUFFER_SIZE = 100000
BATCH_SIZE = 72
GAMMA = 0.99
UPDATE_EVERY = 62

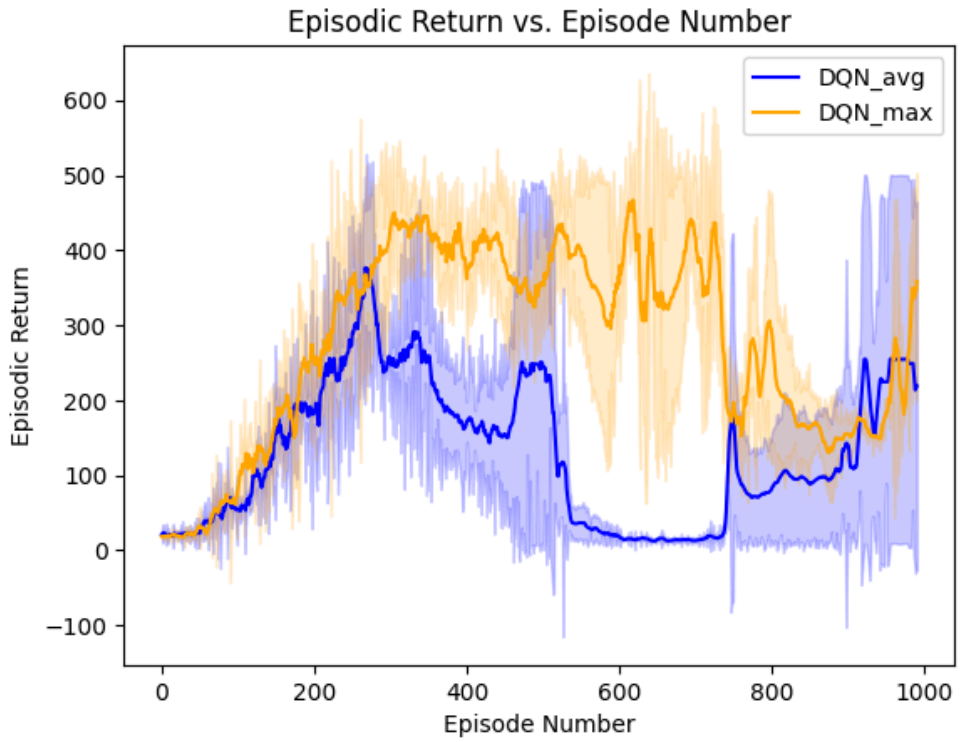
Dueling DQN (Type-2):



LR = 0.00008259
BUFFER_SIZE = 100000
BATCH_SIZE = 234
GAMMA = 0.99
UPDATE_EVERY = 63

Note that in the above figures, “score” must be a quantity that should be minimised.
At each episode, regret is defined as 475 - reward of the current episode.
A summation of all such regrets gives us “score” and we intend to minimise it to get the optimal hyperparameters.

Results:



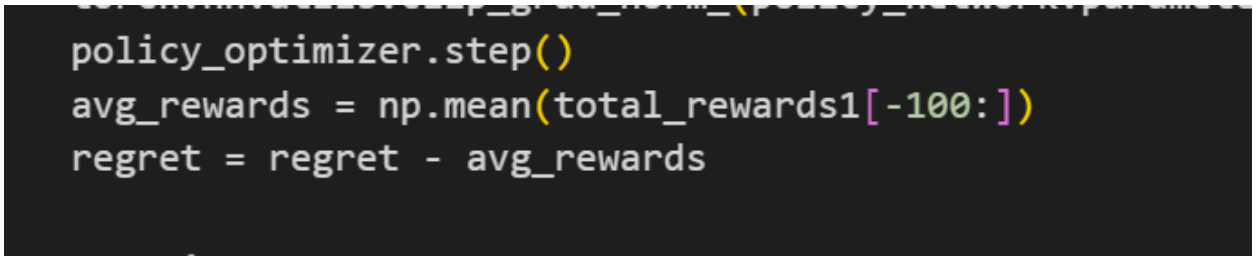
The above plot was generated by training for 1000 episodes. It is clearly inferred from the plot that the type-2 DQN performed better than type-1. The average regret values are also significantly lower for type-2 DQN. The returns for type-1 DQN plummets after around 300 episodes. Type-2 DQN performs better in the 500-800 episode region, but has high variance. The returns start to converge again at the 900 episode mark.

Reasons:
In the CartPole environment, the agent receives a reward of +1 for every timestep it keeps the pole balanced upright. Since the reward signal is sparse, it's crucial for the agent to learn quickly and efficiently which actions lead to longer pole-balancing durations. Emphasizing the maximum advantage can help the agent focus on the most promising actions that lead to extended periods of balancing, facilitating faster learning of the optimal policy. The CartPole environment has a discrete action space, consisting of only two actions: moving the cart left or right. With a limited set of actions, the advantage of each action can vary significantly, and there might be a clear "best" action to take in certain states. By prioritizing the maximum advantage, the agent can focus on exploiting these high-advantage actions more effectively, leading to quicker convergence to an optimal policy.

Dueling DQN - Acrobat:

```
env = gym.make('Acrobot-v1')
```

The code for acrobat is similar to that of cartepole-v1, as mentioned earlier.
The score for calculating regret - is different.
To optimize the hyperparameters, the criteria is that we wish to maximize the summation of average rewards. The idea is to get the maximum return in least possible interactions (and kind of maintain the performance). We have considered that the total number of episodes to train to be 5000.
At each episode we calculate the average reward of the last 100 episodes. And compute the negative of this reward.
Over all the 5000 episodes, we do the summation, i.e. summation of all the negatives of the rewards at each episode. So, to get to the optimal parameters , it means to minimize this summation.



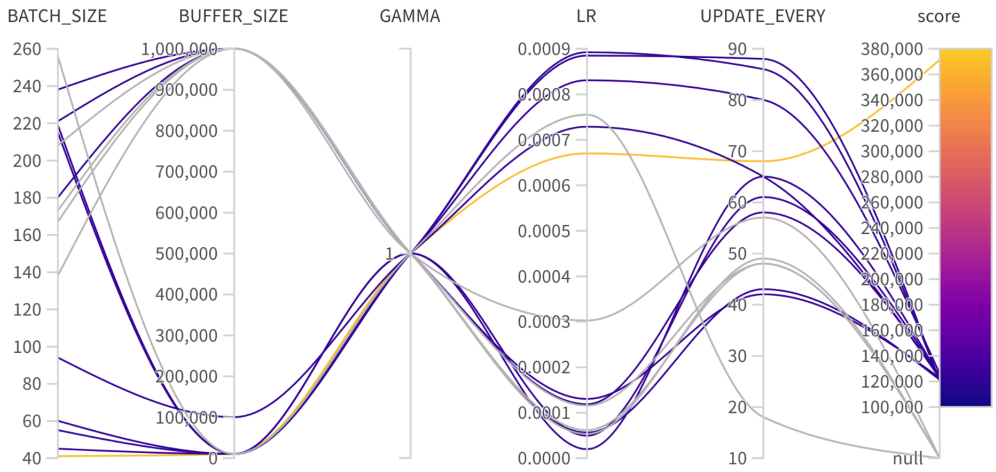
The other parts of the code is similar to the case of Cartepole-v1.

Hyperparameter tuning:

Wandb (https://wandb.ai/site) was used to visualize and perform the hyperparameter testing.
The hyperparameters are

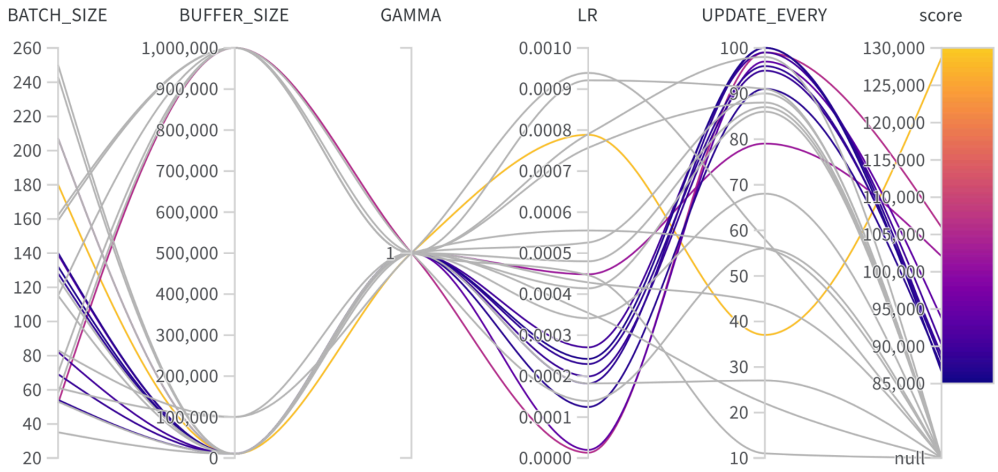
6) Learning Rate
7) Batch size
8) Buffer size
9) Gamma
10) Update every

Dueling DQN (Type-1):



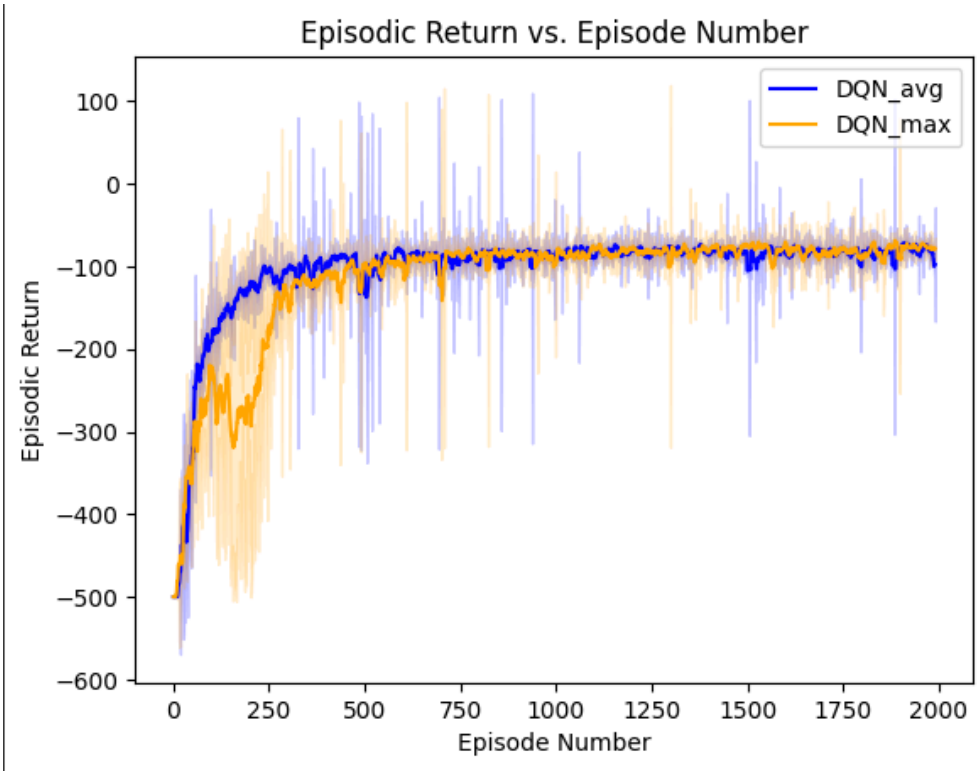
LR = 0.00005615
BUFFER_SIZE = 1000000
BATCH_SIZE = 221
GAMMA = 0.99
UPDATE_EVERY = 43

Dueling DQN (Type-2):



LR = 0.0002415
BUFFER_SIZE = 10000
BATCH_SIZE = 132
GAMMA = 0.99
UPDATE_EVERY = 100

Results:



Clearly, from the plot, we can see that type-1 DQN converges faster than type-2 DQN. Type-2 DQN also has much higher variance than type-1. However, after around 500 episodes, they both converge close to the optimal return.

Reasons:

During the initial exploration phase of learning, the agent's estimates of action values are noisy or inaccurate. In the max advantage method (type-2), the action with the highest advantage value is selected, which can lead to more volatile updates to the Q-values. The max advantage method is susceptible to overestimation bias, where certain actions' advantage values are overestimated. This makes sense when we consider a complex environment like Acrobat.

MC REINFORCE

The reinforce algorithm without baseline case

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

But then this poses a few issues - The algorithm is slowed down due to high variance in the estimate. Larger the state space, greater the variance.
So we use the MC Reinforce algorithm with a baseline:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big(G_t - b(S_t)\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}$$

This is the update rule.
Note that b(s) must be a function of states and it should not depend on the action taken.
This does not change the expected value but reduces the variance.

1) CARTPOLE: (click here)

```
DISCOUNT_FACTOR = 0.99
gamma = 0.99
#number of episodes to run
NUM_EPISODES = 5000

#max steps per episode
MAX_STEPS = 10000

#score agent needs for environment to be solved
SOLVED_SCORE = 475

#device to run model on
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
```

The **discount factor** - gamma was set to **0.99.**
The **number of episodes** needed for training were set to **5000.**
The **solved score** for CARTPOLE V-1 is **475.**

No Baseline case:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}$$

```
class PolicyNetwork(nn.Module):
  def __init__(self, observation_space, action_space,seed,layer_size):
        super(PolicyNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.input_layer = nn.Linear(observation_space, layer_size)
        self.output_layer = nn.Linear(layer_size, action_space)
  def forward(self, x):
        x = self.input_layer(x)
        x = F.relu(x)
        actions = self.output_layer(x)
        action_probs = F.softmax(actions, dim=-1)
        return action_probs
```

The above is the code for defining the policy network using pytorch. As seen here, layer size is the hyper parameter. Seed is used to provide random initialization of the weights of the policy network.
Softmax is used to compute the action probabilities from the values.

```
#Make environment
#env = gym.make('Acrobot-v1')
env = gym.make('CartPole-v1')
#Init network
'''
policy_network = PolicyNetwork(env.observation_space.shape[0], env.action_space.n)
stateval_network = StateValueNetwork(env.observation_space.shape[0])'''


#Init optimizer
```

Initialise the cartpole environment.

```
[ ]  def reset_weights(model):
        for layer in model.children():
            if hasattr(layer, 'reset_parameters'):
                layer.reset_parameters()
```

This function is used to reset the weights after every iteration.

```
seed_list = [1,42,30,25,17]
```

These are the seeds over which we would be running our 5 iterations to average for plots and for hyperparameter tuning.
Hyperparameter tuning:
Wandb (https://wandb.ai/site) was used to visualize and perform the hyperparameter testing.
The hyperparameters are

1) **Learning Rate (range was set from 1e-5 to 1e-2)**
2) **The parameter of network(layer_size) (range was set as integers 64,128,256)**
3) **Gamma (discount factor) is set to 0.99**

The **objective** function returns the average regret over 5 iterations(different seed initialisations of the policy network weights).
The objective function takes in parameters in the format :

```
params = {"LR": integer_for_learning_rate , "network_size":
integer_for_layer_size}
```

The first snippet of the **objective** function :

```
def objective(params):

    regret_avg = 0
    print(params,"Its just getting started ")
    for i in range(5):
      seed = seed_list[i]
      policy_network = PolicyNetwork(env.observation_space.shape[0], env.action_space.n,seed,params["network_size"])
      #stateval_network = StateValueNetwork(env.observation_space.shape[0],seed,params["network_size"])
      reset_weights(policy_network)
      #reset_weights(stateval_network)
      policy_optimizer = optim.Adam(policy_network.parameters(), params["LR"])
      #stateval_optimizer = optim.Adam(stateval_network.parameters(), params["LR"])
      ep = 0
      action_space = np.arange(env.action_space.n)
      total_rewards = []
      regret = 0
```

This just involved starting , and initializing the network and resetting the weights after every iteration. Regret records the total regret for each iteration and ep is the number of episodes for each iteration.

```
while ep < NUM_EPISODES:
  state = env.reset()
  states = []
  rewards = []
  actions = []
  done = False
  while done == False:
    state = torch.from_numpy(state).float().unsqueeze(0)[0]
    action_probability = policy_network.forward(state).detach().numpy()
    if np.isnan(action_probability).any():
      return -float('inf')
    else:
      action = np.random.choice(action_space,p=action_probability)
      state.detach()
      next_state,r,done,_ = env.step(action)
      states.append(state)
      rewards.append(r)
      actions.append(action)
      if done :
        break
      state = next_state

  total_rewards.append(sum(rewards))
  G = process_rewards(rewards,gamma)
  G = torch.FloatTensor(G)

  rewards = torch.FloatTensor(rewards)

  policy_optimizer.zero_grad()
  deltas = [gt for gt in zip(G)]
  deltas = torch.tensor(deltas)
  logprob = [torch.log(policy_network.forward(states[i])) for i in range(len(deltas))]
  policy_loss = []
  for i in range(len(deltas)):

        d = deltas[i]

        lp = logprob[i][actions[i]]

        policy_loss.append(-d * lp)
  policy_optimizer.zero_grad()
  #print(policy_loss,len(actions))
  sum(policy_loss).backward()
  policy_optimizer.step()
  avg_rewards = np.mean(total_rewards[-100:])
```

Loop till we reach the max number of episodes allowed.
Initialize variables to hold values of states visited, rewards for each action and the actions taken at each state. NUM_EPISODES is set to 5000
For each state, choose an action based on the policy network prediction, and reach the next state. This is continued until we get done = True, ie when the episode is ended.
We save all the reward values in the list and then pass it to the **process_rewards** function for computing the future rewards using the discounted factor. **process_rewards** is used to find the discounted rewards for all the future rewards using the discount factor(gamma) .

```python
def process_rewards(rewards,gamma):
    G = []
    total_r = 0
    for r in reversed(rewards):
        total_r = r + total_r * DISCOUNT_FACTOR
        G.insert(0, total_r)
    G = torch.tensor(G)
    #G = (G - G.mean())/G.std()
    return G
```

Log probabilities are computed from the policy network predictions.

```python
total_rewards.append(sum(rewards))
G = process_rewards(rewards,gamma)
G = torch.FloatTensor(G)

rewards = torch.FloatTensor(rewards)

policy_optimizer.zero_grad()
deltas = [gt for gt in zip(G)]
deltas = torch.tensor(deltas)
logprob = [torch.log(policy_network.forward(states[i])) for i in range(len(deltas))]
policy_loss = []
for i in range(len(deltas)):

      d = deltas[i]

      lp = logprob[i][actions[i]]

      policy_loss.append(-d * lp)
policy_optimizer.zero_grad()
#print(policy_loss,len(actions))
sum(policy_loss).backward()
policy_optimizer.step()
avg_rewards = np.mean(total_rewards[-100:])
```

Then, the update is done based on the equation:

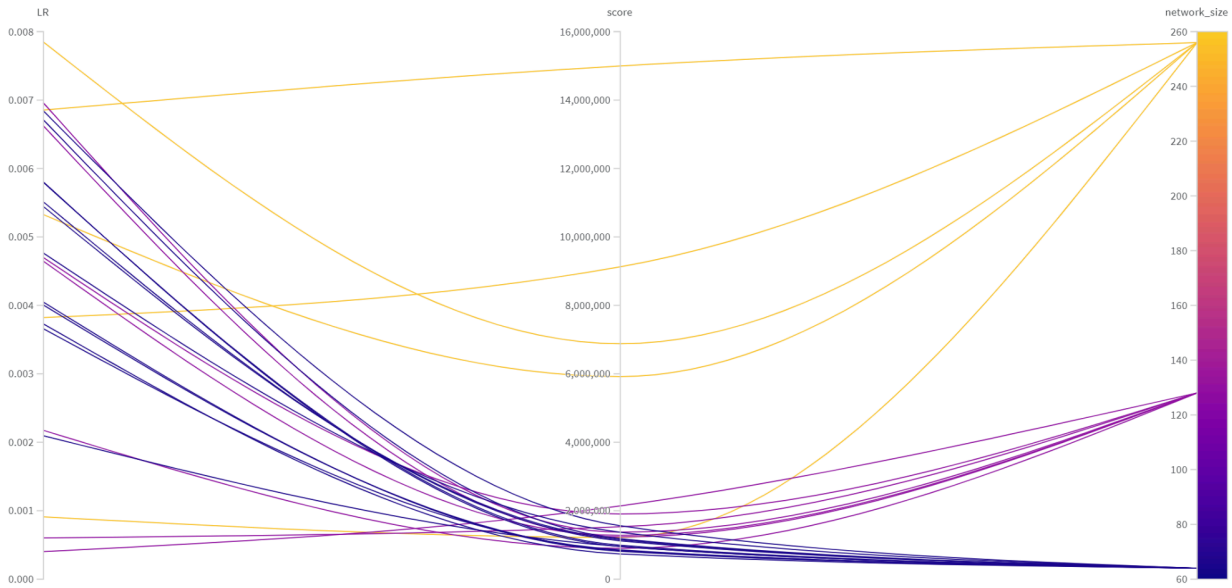
$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}$$

The average reward of the last 100 episodes is calculated. When the average reward reaches a value of 475, the environment is said to be solved.

```
        policy_optimizer.zero_grad()
        #print(policy_loss,len(actions))
        sum(policy_loss).backward()
        policy_optimizer.step()
        avg_rewards = np.mean(total_rewards[-100:])
        ep +=1
        if ep % 400  == 0:
          print("Ep:",ep,"last 100 episodes reward is  :",avg_rewards, end="\n")
        if avg_rewards > 475:
            print("problem solved at episode",ep)
            break
        regret += 475 - avg_rewards
      print(params,regret)
      regret_avg += regret
    print("regret_avg:",regret_avg,"for",params)
    return regret_avg
```

```
import wandb

def main():
    wandb.init(project="RLA2cartepole-wobase")
    #agent = TutorialAgent(state_size=state_shape,action_size = action_shape,seed = 0, config=wandb.config)
    score = objective(wandb.config)
    wandb.log({"score": score})

# 2: Define the search space
sweep_configuration = {
    "method": "bayes",

    "metric": {"goal": "minimize", "name": "score"},
    "parameters": {
        "LR": {"max": 1e-2, "min": 1e-5},
        "network_size": {"values": [64,256,128]},
    },
}

# 3: Start the sweep
sweep_id = wandb.sweep(sweep=sweep_configuration, project="RLA2cartepole-wobase")

wandb.agent(sweep_id, function=main, count=10)
```

As shown above, **wandb** is used to find the optimal hyperparameters.
Bayesian optimization is used.



Note that in the above figures, “**score**” must be a quantity that should be minimized.
At each episode, **regret** is defined as **475 - reward** of the last 100 episodes from the current episode. The **objective** function returns this **regret** value

```
    avg_rewards = np.mean(total_rewards[-100:])
    ep +=1
    if ep % 400  == 0:
      print("Ep:",ep,"last 100 episodes reward is  :",avg_rewards, end="\n")
    if avg_rewards > 475:
        print("problem solved at episode",ep)
        break
    regret += 475 - avg_rewards
  print(params,regret)
  regret_avg += regret
print("regret_avg:",regret_avg,"for",params)
return regret_avg
```

A summation of all such regrets gives us a “**score**” and we intend to minimize it to get the optimal hyperparameters.

From the above, it was decided that the best set of hyperparameters are :

```
params_without_baseline = {"LR": 0.00373 , "network_size": 64}
```

Learning rate : 0.00373
Layer size : 64
Similarly, the same procedure is followed to determine the optimal hyperparameters for the “with baseline” case.

After determining the optimal parameters, we use these parameters to plot the results.

Plot function code:([click here])

```python
# Define the objective function
def objective(params):
    eps_list = []
    regret_avg = 0
    reward_per_ep= [0 for i in range(5000)]
    reward_per_ep_list = [[475,475,475,475,475] for i in range(5000)]
    min_ep = 5000
    print(params,"Its just getting started ")
    for iteration in range(5):
      #c= 0
      seed = seed_list[iteration]
      policy_network = PolicyNetwork(env.observation_space.shape[0], env.action_space.n,seed,params["network_size"])
      stateval_network = StateValueNetwork(env.observation_space.shape[0],seed,params["network_size"])
      reset_weights(policy_network)
      reset_weights(stateval_network)
      policy_optimizer = optim.Adam(policy_network.parameters(), params["LR"])
      stateval_optimizer = optim.Adam(stateval_network.parameters(), params["LR"])
      ep = 0
      action_space = np.arange(env.action_space.n)
      total_rewards = []
      avg_rewards_list = []
      regret = 0
      while ep < NUM_EPISODES:
        state = env.reset()
        states = []
        rewards = []
        actions = []
        done = False
        while done == False:
          state = torch.from_numpy(state).float().unsqueeze(0)[0]
          action_probability = policy_network.forward(state).detach().numpy()
          if np.isnan(action_probability).any():
            return float('inf')
          else:
```

```python
  while done == False:
    state = torch.from_numpy(state).float().unsqueeze(0)[0]
    action_probability = policy_network.forward(state).detach().numpy()
    if np.isnan(action_probability).any():
      return float('inf')
    else:
      action = np.random.choice(action_space,p=action_probability)
      state.detach()
      next_state,r,done,_ = env.step(action)
      states.append(state)
      rewards.append(r)
      actions.append(action)
      if done :
        break
      state = next_state
  total_rewards.append(sum(rewards))
  G = process_rewards(rewards,gamma)
  G = torch.FloatTensor(G)
  rewards = torch.FloatTensor(rewards)
  policy_optimizer.zero_grad()
  deltas = [gt for gt in zip(G)]
  deltas = torch.tensor(deltas)
  logprob = [torch.log(policy_network.forward(states[i])) for i in range(len(deltas))]
  policy_loss = []
  for i in range(len(deltas)):
        d = deltas[i]
        lp = logprob[i][actions[i]]
        policy_loss.append(-d * lp)
  policy_optimizer.zero_grad()
  #print(policy_loss,len(actions))
  sum(policy_loss).backward()
  policy_optimizer.step()
  avg_rewards = np.mean(total_rewards[-100:])
  avg_rewards_list.append(avg_rewards)
```

```
    policy_optimizer.step()
    avg_rewards = np.mean(total_rewards[-100:])
    avg_rewards_list.append(avg_rewards)
    ep +=1
    if ep % 400  == 0:
      print("Ep:",ep,"last 100 episodes reward is  :",avg_rewards, end="\n")
    if avg_rewards > 475 :
        #c= 1
        print("problem solved at episode",ep)
        eps_list.append(ep)
        if ep < min_ep:
          min_ep = ep
        break
    regret += 475 - avg_rewards
  for j in range(len(avg_rewards_list)):
        #print(reward_per_ep_list[j])
        reward_per_ep_list[j][iteration] = avg_rewards_list[j]
  print(iteration,params,regret)
  regret_avg += regret
print("regret_avg:",regret_avg,"for",params)
for i in range(len(reward_per_ep_list)):
  if len(reward_per_ep_list[i]) == 5:
    #print(reward_per_ep_list[i],"oohoo")
    reward_per_ep[i] = sum(reward_per_ep_list[i])/5
  else:
    k = 5 - len(reward_per_ep_list[i])
    for h in range(k):
      reward_per_ep_list.append(475)
    reward_per_ep[i] = sum(reward_per_ep_list[i])/5
    #print(reward_per_ep[i],'ohoooo')
return regret_avg/5 , reward_per_ep, min_ep,eps_list,reward_per_ep_list
```

As seen from the above code snippets, the code for the objective function is similar, but this time this function returns more values that provide information about the trained model. It averages over 5 runs. The list **`L_without_baseline = objective(params_without_baseline)`**
Stores the values of average regret , average reward per episode,list of the number of episodes under different seeds, and the list "**reward_per_ep_list"** is a list of 5000 arrays, each array contains 5 values, ie the reward value at the **i th episode** in each of the 5 iterations.
This is used to plot the final plots for comparison.

```
means_without_baseline = [np.mean(sublist) for sublist in
without_baseline_list]
std_devs_without_baseline = [np.std(sublist) for sublist in
without_baseline_list]
means_with_baseline = [np.mean(sublist) for sublist in
L_with_baseline[-1]]
std_devs_with_baseline = [np.std(sublist) for sublist in
L_with_baseline[-1]]
plt.plot(means_without_baseline, label='without_baseline')
plt.fill_between(range(len(means_without_baseline)),
np.subtract(means_without_baseline, std_devs_without_baseline),
```

```
np.add(means_without_baseline, std_devs_without_baseline), alpha=0.3,
label='Mean ± Std Dev')
plt.plot(means_with_baseline, label='with baseline')
plt.fill_between(range(len(means_with_baseline)),
np.subtract(means_with_baseline, std_devs_with_baseline),
np.add(means_with_baseline, std_devs_with_baseline), alpha=0.3,
label='Mean ± Std Dev')
plt.xlabel('Episode number')
plt.ylabel('Average episodic return')
#plt.title('Mean and Standard Deviation of Sublists')
plt.legend()
plt.grid(True)
plt.show()
```

MC REINFORCE With BASELINE:(click here)

```
class StateValueNetwork(nn.Module):
    def __init__(self, observation_space,seed,layer_size):
        super(StateValueNetwork, self).__init__()
        self.input_layer = nn.Linear(observation_space, 128)
        self.output_layer = nn.Linear(128, 1)
    def forward(self, x):
        x = self.input_layer(x)
        x = F.relu(x)
        state_value = self.output_layer(x)
        return state_value
```

This code explains the initialisation of pytorch network for state values for performing Reinforce algorithm with baseline V(s). The baseline V (s) is updated by TD(0) method.

```
def train_value(G, state_vals, optimizer):
    val_loss = F.mse_loss(state_vals, G)
    optimizer.zero_grad()
    val_loss.backward()
    optimizer.step()
```

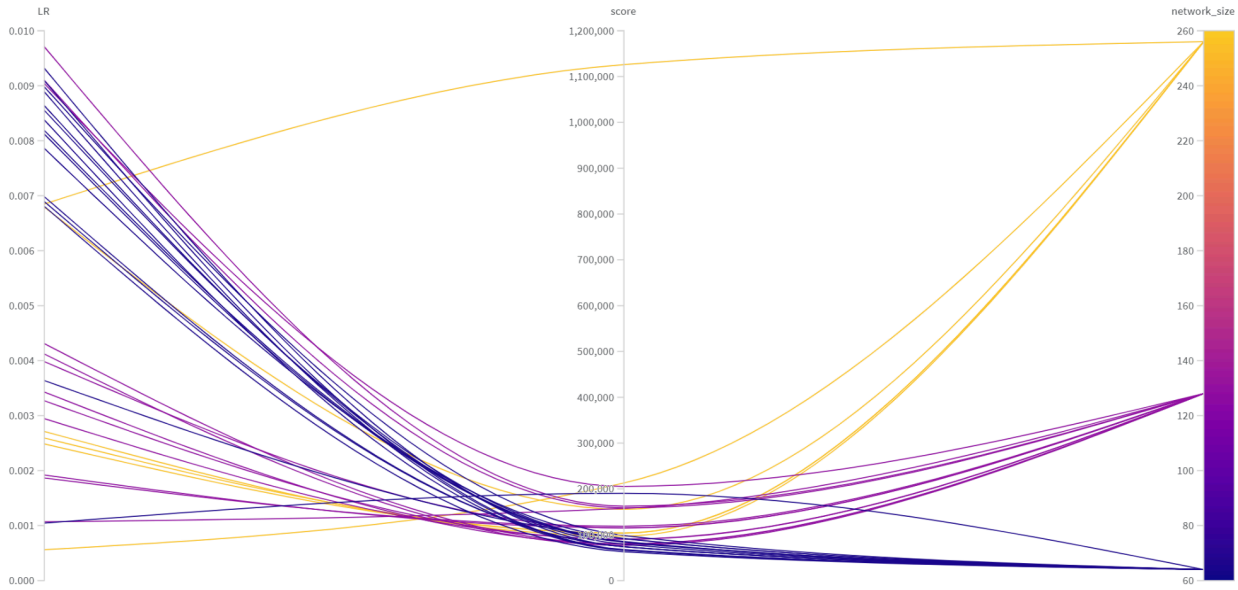
This strip of code explains the function that helps us optimize the weights of the state value network by minimizing the mse values between the predicted state values and the state values

from TD(0) update.

```
train_value(G, state_vals, stateval_optimizer)

deltas = [gt - val for gt, val in zip(G, state_vals)]
deltas = torch.tensor(deltas)
logprob = [torch.log(policy_network.forward(states[i])) for i in range(len(deltas))]
policy_loss = []
for i in range(len(deltas)):

      d = deltas[i]

      lp = logprob[i][actions[i]]

      policy_loss.append(-d * lp)
policy_optimizer.zero_grad()
#print(policy_loss,len(actions))
sum(policy_loss).backward()
policy_optimizer.step()
avg_rewards = np.mean(total_rewards1[-100:])
regret = regret  + 475 - avg_rewards

ep +=1
if ep%400 == 0:
    print("Ep:",ep,"last 100 episodes reward is  :",avg_rewards, end="\n")
if avg_rewards > 475:
```

Then we perform updates as:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha(G_t - V(S_t; \boldsymbol{\Phi}))\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}$$



The hyperparameters are

4) **Learning Rate (range was set from 1e-5 to 1e-2)**
5) **The parameter of network(layer_size) (range was set as integers 64,128,256)**
6) **Gamma (discount factor) is set to 0.99**

Note that in the above figures, “score” must be a quantity that should be minimized.
At each episode, regret is defined as 475 - reward of the current episode.
A summation of all such regrets gives us “score” and we intend to minimize it to get the optimal hyperparameters.

```
params_with_baseline = {"LR": 0.0002264 , "network_size": 64}
```

Learning rate : 0.0002264
Network size : 64
After determining the optimal parameters, we use these parameters to plot the results.
Plot function code:(click here)
The plot function code for this case is similar to the previous case.
The list **`L_with_baseline = average_over_5_runs(params_with_baseline)`**
Stores the values of average regret , average reward per episode,list of the number of episodes under different seeds, and the list “**reward_per_ep_list”** is a list of 5000 arrays, each array contains 5 values, ie the reward value at the **i th episode** in each of the 5 iterations.

ACROBOT:

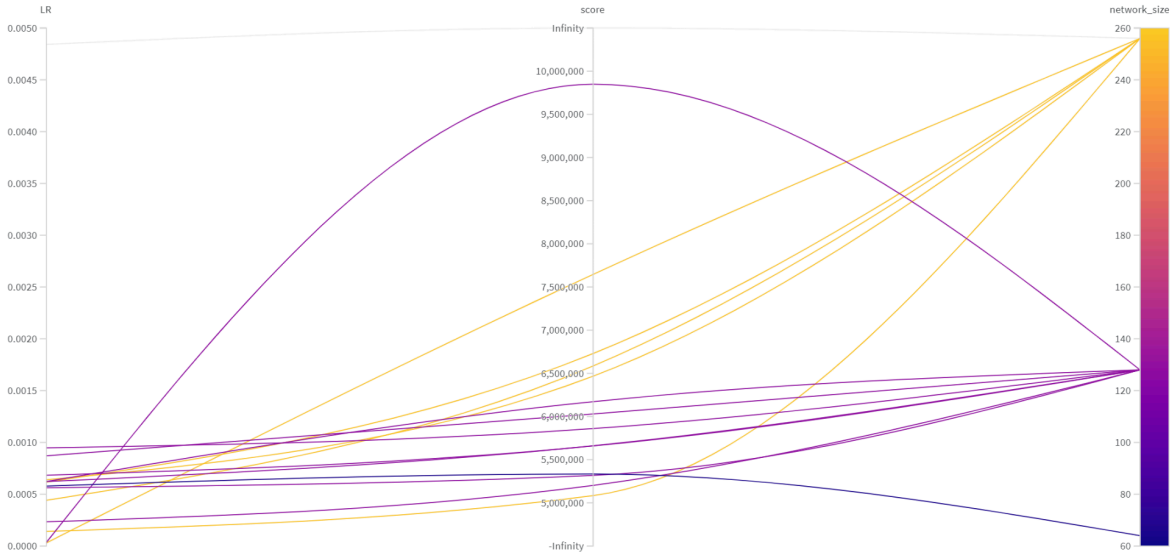
Code:
with baseline: (click here)
Without baseline : (click here)

```
env = gym.make('Acrobot-v1')
```

The code for acrobat is similar to that of cartepole-v1, as mentioned earlier.
The score for calculating regret - is different.
To optimize the hyperparameters, the criteria is that we wish to maximize the summation of average rewards. The idea is to get the maximum return in least possible interactions (and kind of maintain the performance). We have considered that the total number of episodes to train to be 5000.
At each episode we calculate the average reward of the last 100 episodes. And compute the negative of this reward.
Over all the 5000 episodes, we do the summation, i.e. summation of all the negatives of the rewards at each episode. So, to get to the optimal parameters , it means to minimize this summation.

```
policy_optimizer.step()
avg_rewards = np.mean(total_rewards1[-100:])
regret = regret - avg_rewards
```

The other parts of the code is similar to the case of Cartepole-v1.
Without baseline : Hyperparameter tuning:
(using Wandb)



The above is the plot for hyperparameter tuning without baseline for Acrobot
From the above:

```
params_without_baseline = {"LR": 0.0001407 , "network_size": 256}
```

Learning rate : 0.0001407
Layer size : 256

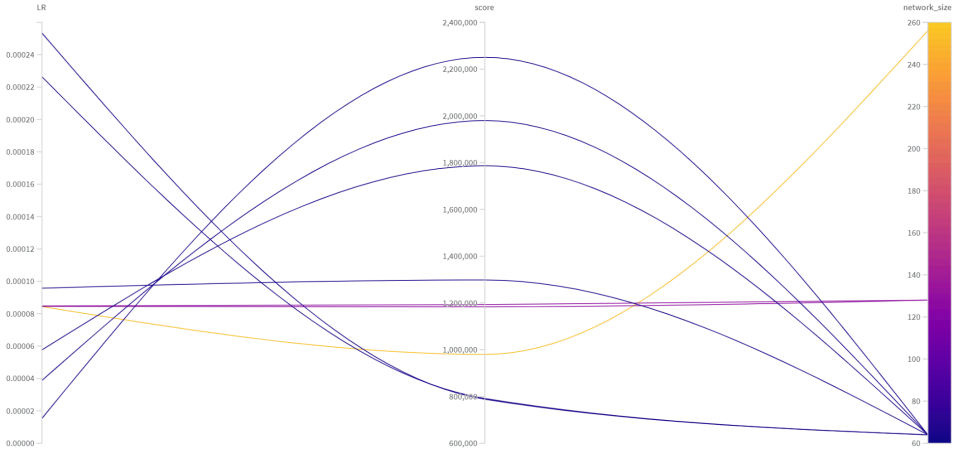
With baseline : Hyperparameter tuning:

(using Wandb)

```
params_with_baseline = {"LR": 0.0002264 , "network_size": 64}
```

Learning rate : 0.0002264

Layer size : 64



The above is the plot for hyperparameter tuning with a baseline case for Acrobot. The hyperparameters are

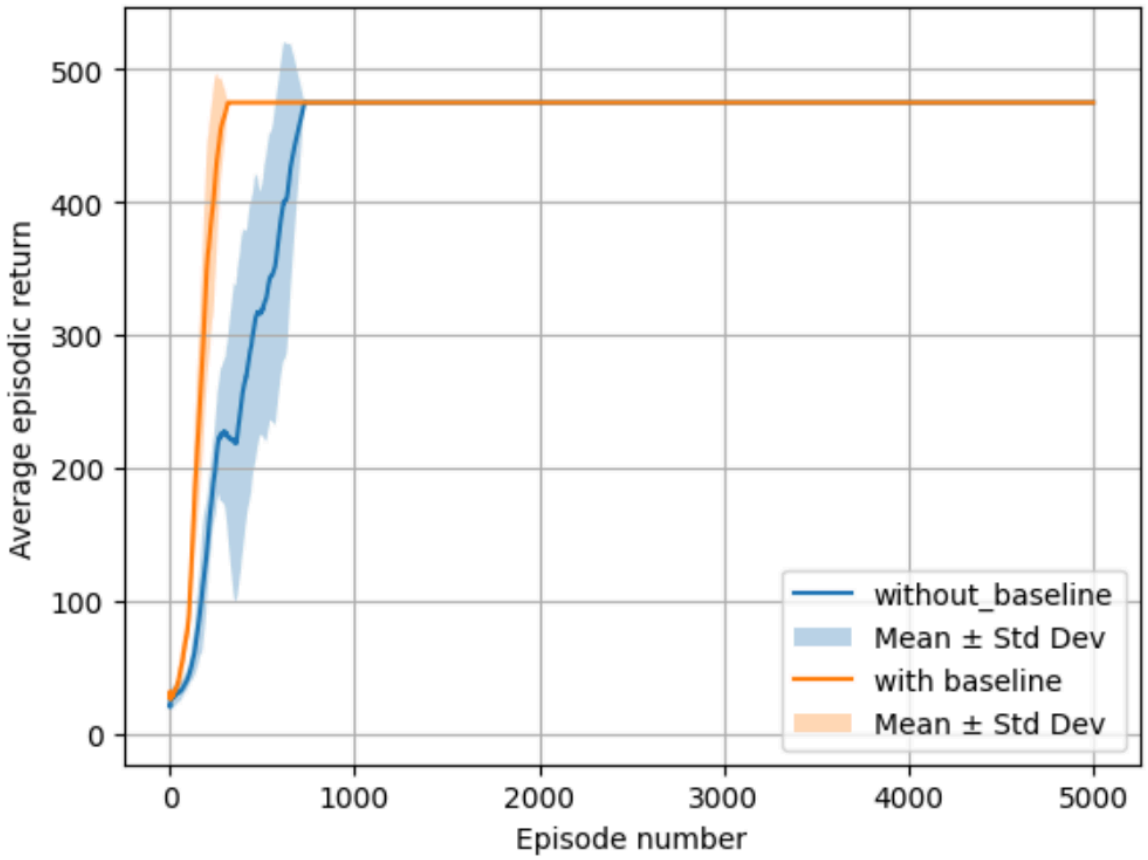
7) **Learning Rate (range was set from 1e-5 to 1e-2)**
8) **The parameter of network(layer_size) (range was set as integers 64,128,256)**
9) **Gamma (discount factor) is set to 0.99**

The plot and inference functions for these case is exactly similar to the functions in cartpole case.
Plot Code : (click here)

INFERENCE - MC REINFORCE

Cartpole-v1:

Plots comparing the performance of with and without baseline :

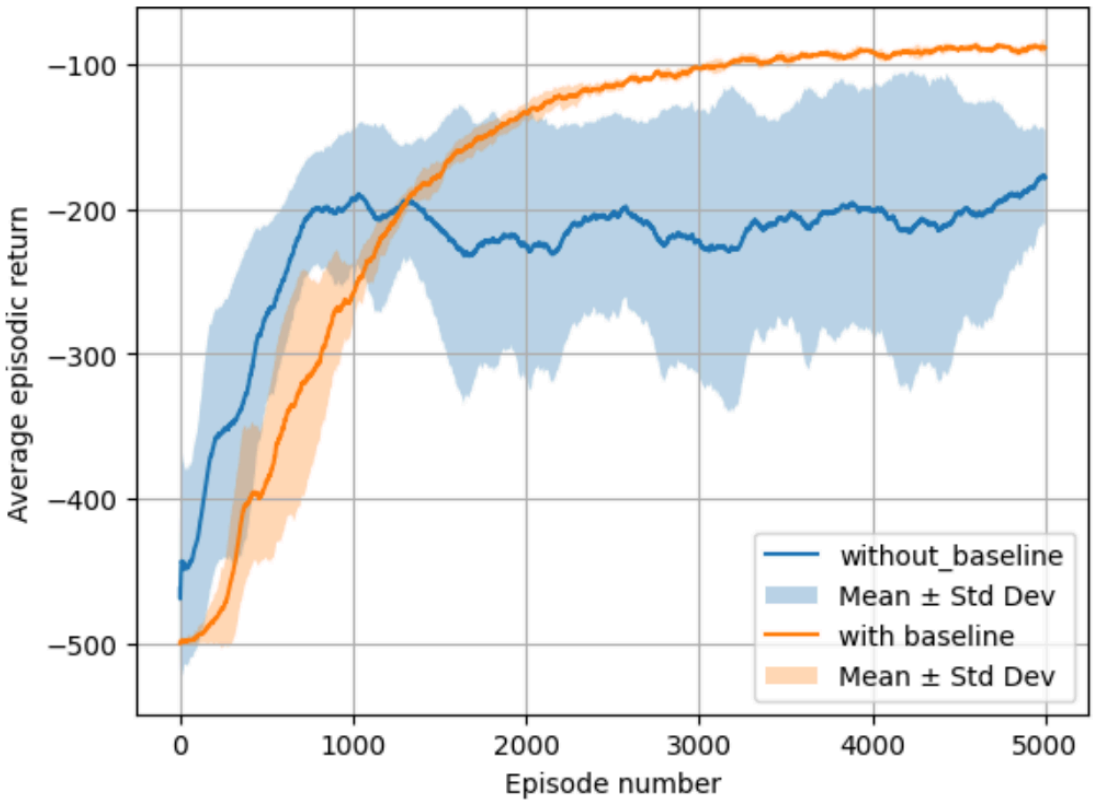


As you can see, the **baseline case performs better** than the case without baseline. This can be seen in terms of both average regret : (**the total regret for the "with baseline" case is much lesser than the without regret case.)** . **The number of episodes required by the "with baseline" case is also lesser than the number of episodes required by "without baseline" case for reaching average reward for the last 100 episodes to be 475.** From the plot, it is also evident that the without baseline case has a **higher band of [mean - standard deviation , mean + standard deviation], which means it has a higher variance, as compared to the without baseline case, where the band is narrower.**

(note that these plots have a constant 475 ,because in cartepole-v1, once the average reward of the last 100 episodes is greater than 475, the environment is said to be solved. So, using the assumption that once the environment is solved, the reward value average is always 475.
In reality this may not always be the case, but we don't care about the agent once the environment is solved, so it is fine to make such an assumption i.e. once the env is solved, the average reward is always 475, and go ahead to make it computationally more efficient.

Acrobot-v1:

Plot for comparison of with baseline vs without baseline:



As observed in the graph, **it is evident that the case of with baseline works better than without baseline**. For the same number of episodes for training i.e. **5000** : we can see that with baseline gets a **better average reward**(last **100** episodes) than without baseline case.
Note that we had run for a larger number of episodes - 5000, because, if we had run for a smaller number of episodes, say 1000, it would have led to wrong observation. This means that for a lesser number of episodes obtained for training, without baseline performs better than with the baseline method. **But as we have more data and more episodes to train the policy, we find that the baseline performs better and its performance increases.**
From the plot, it is also evident that the without baseline case has a higher band of [mean - standard deviation , mean + standard deviation], which means it has a higher variance, as compared to the without baseline case, where the band is narrower.

Reason :

Without baseline, the original Reinforce(Policy Gradient Algorithm) shows high variance due to empirical returns. To reduce a baseline, we subtract the G(t) by a function that depends only on the state and not on any actions. The value function is one such example, it doesn't introduce any bias to the policy gradient.
This is why we observe that having a baseline performs better.
The high variance can be validated by the plots above.

Github link:

(https://github.com/AnirudN/CS6700_CE21B014_ED21B038_PA2/tree/main)(Click here)